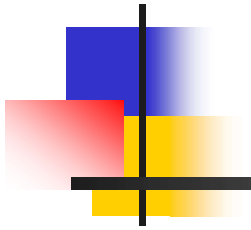
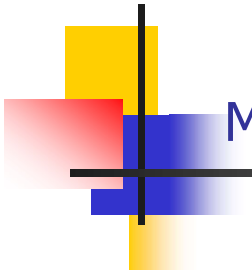


Tema 3
Diseño e Implementación de TADS Arborescentes

Estructura de datos y algoritmos
Facultad de Informática - Universidad Complutense de Madrid

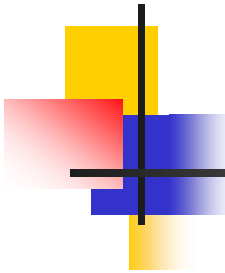
Transparencias de los Profs.:
Mercedes Gómez Albarrán y José Luis Sierra Rodríguez



Motivación

- Deseamos implementar un programa que pida al usuario que piense un animal y trate de adivinarlo haciendo distintas preguntas a las que el usuario sólo puede responder sí o no

```
Tiene cuatro patas?(si/no)
no
Vive en el agua?(si/no)
no
Mmmmmm, dejame que piense.....
Creo que el animal en que estabas pensando era...:Serpiente!
Espero haber acertado... Gracias por jugar conmigo
Presione una tecla para continuar . . .
```

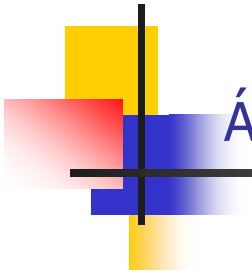


Motivación

- Y después ampliar su funcionalidad de forma que si el programa no "adivina" el animal en el que está pensando el usuario le pida que introduzca de qué animal se trata, así como una pregunta que le permita discriminar con el animal conjeturado

```
Tiene cuatro patas?(si/no)
no
Vive en el agua?(si/no)
no
Mmmmm, dejame que piense.....
Creo que el animal en que estabas pensando era...:Serpiente!
Acerte? (si/no):
no
De que animal se trata?
Avestruz
Que pregunta tendria que hacer para distinguirlo de Serpiente?
Es reptil?
En este caso, la respuesta seria Serpiente? (si/no)
si
Estupendo! Gracias por jugar conmigo.
Otra partida? (si/no)
si
Tiene cuatro patas?(si/no)
no
Vive en el agua?(si/no)
no
Es reptil?(si/no)
no
Mmmmm, dejame que piense.....
Creo que el animal en que estabas pensando era...:Avestruz!
Acerte? (si/no):
_
```

- ¿Cómo implementaríamos este programa?



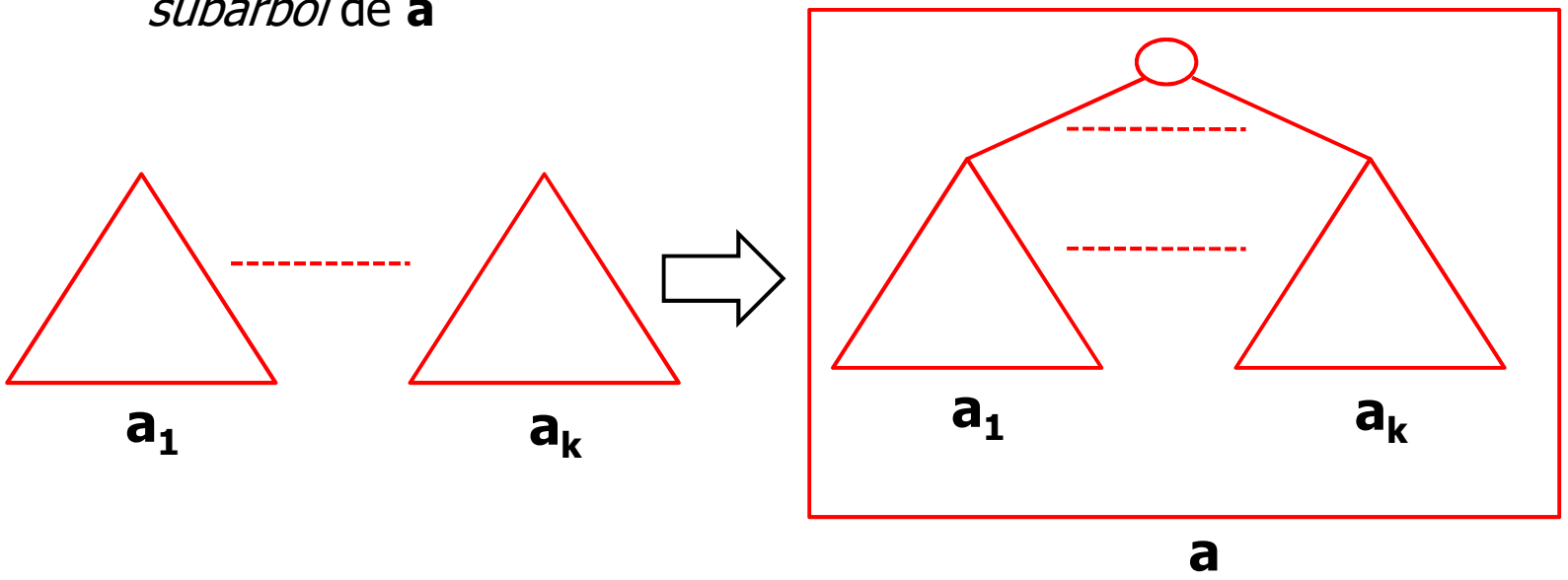
Árboles

- Para organizar la información del programa de ejemplo, así como de otros muchos programas, necesitamos **árboles**.
- Al contrario que los TADs lineales que hemos visto en el capítulo anterior, los **árboles** son estructuras **no lineales**: dado un elemento de información, tenemos múltiples ramificaciones
- Este tipo de estructuras jerárquicas surgen de manera natural dentro y fuera de la informática:
  - Árboles genealógicos.
  - Organización de un libro en capítulos, secciones, etc.
  - Estructura de directorios y archivos de un sistema operativo.
  - Árboles de análisis sintáctico.

Árboles: Modelo matemático

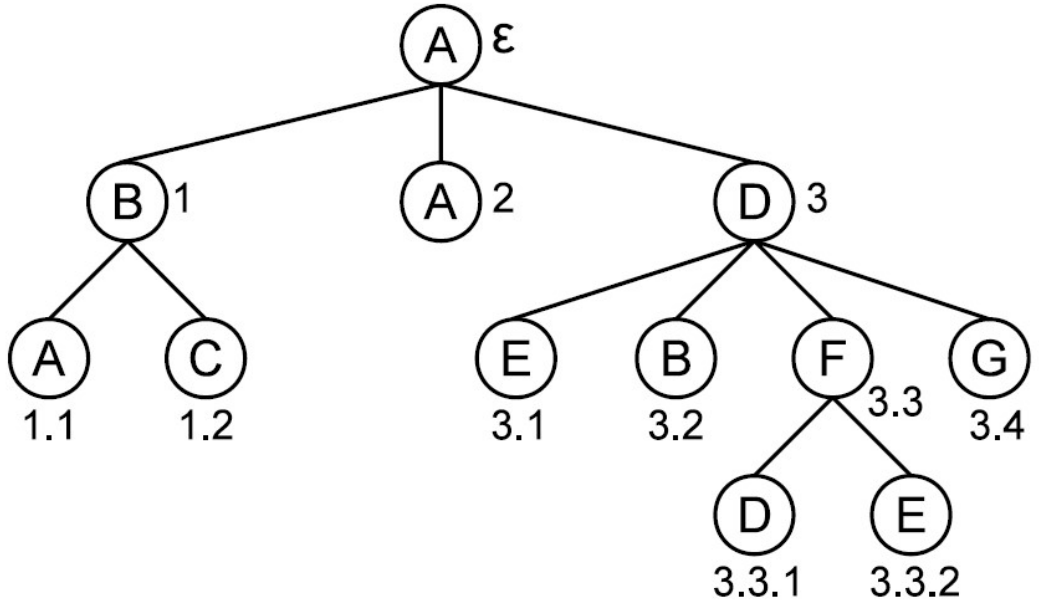
- Estructuras formadas por **nodos** construidas de manera inductiva:
  - Un solo nodo es ya un árbol **a**. El nodo es la *raíz* de dicho árbol

    a
  - Dado *n* árboles $\mathbf{a}_1$ … $\mathbf{a}_n$ es posible construir un nuevo árbol **a** añadiendo un nuevo nodo como raíz y conectando dicho nodo con las raíces de cada uno de los árboles $\mathbf{a}_i$. Se dice que cada $\mathbf{a}_i$ es un *subárbol* de **a**



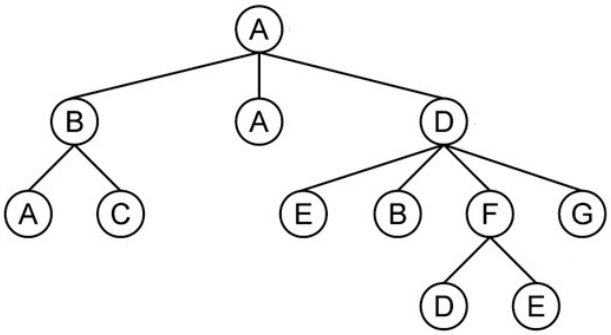
Árboles: Modelo matemático

- Los nodos de un árbol pueden posicionarse mediante **cadenas de localización**
  - La raíz tiene asignada la cadena vacía (la denotaremos por ε)
  - Si un nodo tiene asignada la cadena de localización α, el hijo i-ésimo de dicho nodo tendrá asignada la cadena de localización α**.i**



Árboles: Modelo matemático

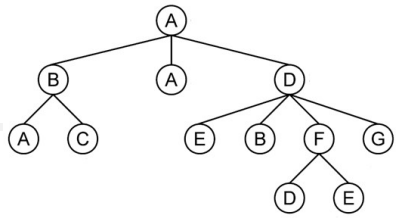
- Un árbol puede describirse mediante una función **a**: **N** $\rightarrow$ **V** donde:
  - **N** es el conjunto de cadenas de localización
  - **V** es el conjunto de contenidos de los nodos



N = {ε,1,2,3,1.1,1.2,3.1,3.2,3.3,3.4,3.3.1,3.3.2}
V = {A,B,C,D,E,F,G}

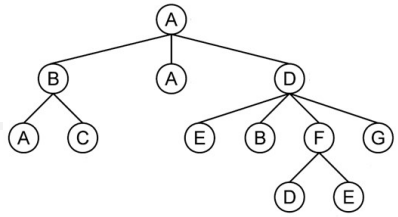
a(ε)=A
a(1)=B
a(2)=A
a(3)=D
a(1.1)=A
a(1.2)=C
a(3.1)=E
a(3.2)=B
a(3.3)=F
a(3.4)=G
a(3.3.1)=D
a(3.3.2)=E

Árboles: Terminología

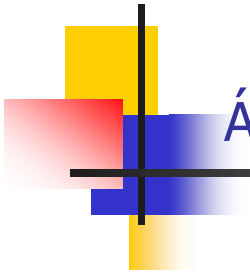


- Dado un árbol **a**: **N** $\rightarrow$ **V**
  - La **raíz** es el nodo con cadena de localización ε
  - Las **hojas** son aquellos nodos con cadenas de localización α tales que no existe ningún nodo con cadena de localización α.i
  - Los **nodos internos** son los nodos que no son hojas
  - Si el nodo **n** tiene como cadena de localización α y el nodo **n'** tiene como cadena de localización α.i, entonces **n** es el **padre** de **n'** y **n'** es un **hijo** de **n**
  - Dos nodos de posiciones α.i y α.j (i$\neq$j) se llaman **hermanos**
  - Un **camino** es una secuencia de nodos $\mathbf{n_1}$ $\mathbf{n_2}$ … $\mathbf{n_k}$ en el que cada nodo $\mathbf{n_i}$ ($1 \leq i < k$) es padre del nodo $\mathbf{n_{i+1}}$.
  - Una **rama** es un camino que comienza en la raíz y termina en una hoja
  - La **longitud de un camino** es el número de nodos que este contiene.
  - El **nivel** o **profundidad** de un nodo es la longitud del camino que va desde la raíz hasta dicho nodo (en particular, el nivel de la raíz es 1)

Árboles: Terminología

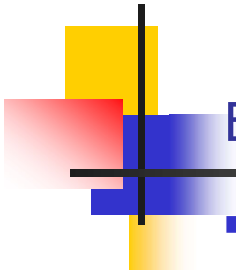


- La **talla** o **altura** de un árbol es el máximo de los niveles de todos los nodos del árbol
- El **grado** o **aridad** de un nodo es el número de hijos que tiene dicho nodo.
- El **grado** o **aridad** de un árbol es el máximo de los grados de los nodos de dicho árbol
- El nodo **n** es un *antepasado* (respectivamente, *descendiente*) de un nodo **n'** si existe un camino que va desde **n** a **n'**
- Cada nodo **n** de **a** determina un subárbol $\mathbf{a}_0$ con raíz en dicho nodo
- Dado un nodo **n**, los subárboles que cuelgan de él se llaman **árboles hijo** de **n**



Árboles: Tipos de árboles

- **Árboles ordenados**: un árbol es **ordenado** si el orden de los hijos de cada nodo es relevante
- **Árboles n-arios**: Un árbol es **n-ario** si el máximo número de hijos de cada nodo es **n**
- **Árboles generales**: Un árbol es **general** si no existe una limitación al número máximo de hijos de cada nodo
- Un tipo de árboles especialmente relevante es el de los árboles **2-arios**. Dichos árboles se denominarán **árboles binarios**.

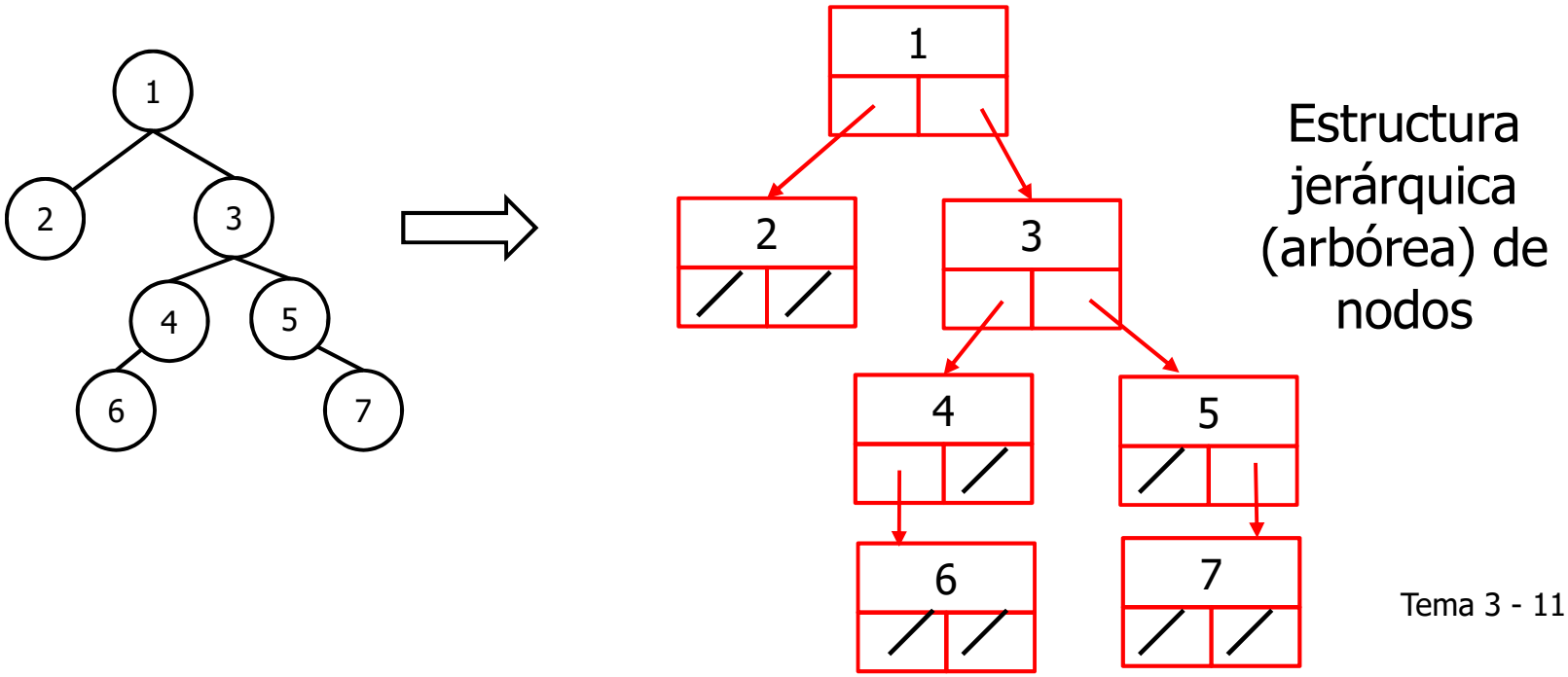


El TAD de los árboles binarios

- Un **árbol binario** es una estructura recursiva cuyos nodos tienen, a lo sumo, dos árboles hijo (el hijo izquierdo, y el hijo derecho)
- **TAD de los árboles binarios**: `Arbin`
- Operaciones básicas de `Arbin`:
  - Crear un árbol vacío (sin nodos): **arbolVacio**: --> Arbin. Generadora
  - Crear un árbol con un único nodo cuyo valor es `Elm`: **arbolSimple**: Elm --> Arbin. Generadora
  - Crear un árbol a partir del elemento `Elm` que se almacenará en la raíz y dos árboles que actúan como hijo izquierdo `Iz` y derecho `Der` de la raíz: **cons**: Iz, Elem, Der --> Arbin. Generadora
  - Obtener el elemento almacenado en la raíz: **raiz**: Arbin--> Elm. Observadora parcial (no definida en árboles vacíos).
  - Obtener el hijo izquierdo de la raíz: **hijoIz**: Arbin--> Arbin. Observadora parcial (no definida en árboles vacíos).
  - Obtener el hijo derecho de la raíz: **hijoDer**: Arbin--> Arbin. Observadora parcial (no definida en árboles vacíos).
  - Averiguar si un árbol es vacío: **esVacio**: Arbin--> Bool. Observadora.

Implementación de árboles binarios mediante nodos

- Cuando no hay restricciones adicionales sobre la estructura del árbol binario, la implementación más razonable se basa en **nodos** creados en **memoria dinámica**.
- Cada nodo tendrá:
  - El elemento de información del árbol
  - Un puntero al nodo raíz del subárbol izquierdo
  - Otro puntero al nodo raíz del subárbol derecho

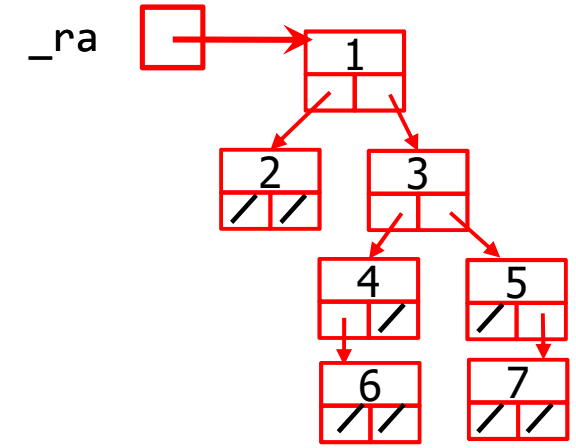


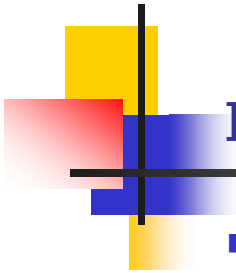
Estructura jerárquica (arbórea) de nodos

Implementación de árboles binarios mediante nodos

- **Tipos representantes**: Un puntero a nodo que contiene la raíz del árbol. Estructura jerárquica de nodos

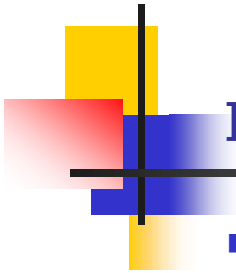
```
template <class T>
class Arbin {
   private:
      class Nodo {
      public:
       Nodo() : _iz(NULL), _dr(NULL) {}
       Nodo(const T &elem) : _elem(elem),_iz(NULL),_dr(NULL) {}
       Nodo(Nodo *iz, const T &elem, Nodo *dr) :
            _elem(elem), _iz(iz), _dr(dr) {}
       T _elem;
       Nodo* _iz;
       Nodo* _dr;
      };
      Nodo* _ra;
  public:
   ...
};
```





Implementación de árboles binarios mediante nodos

- **Invariante de la representación**
  - El árbol está vacío
  - O bien, para cada nodo **n** hay **un único camino** que lleva desde la raíz hasta **n**
- **Relación de equivalencia**
  - Dos árboles son iguales si ambos son vacíos
  - O bien si: (i) ambos no son vacíos, (ii) los contenidos de sus raíces son iguales, (iii) los hijos izquierdos son iguales, y (iv) los hijos derechos son iguales



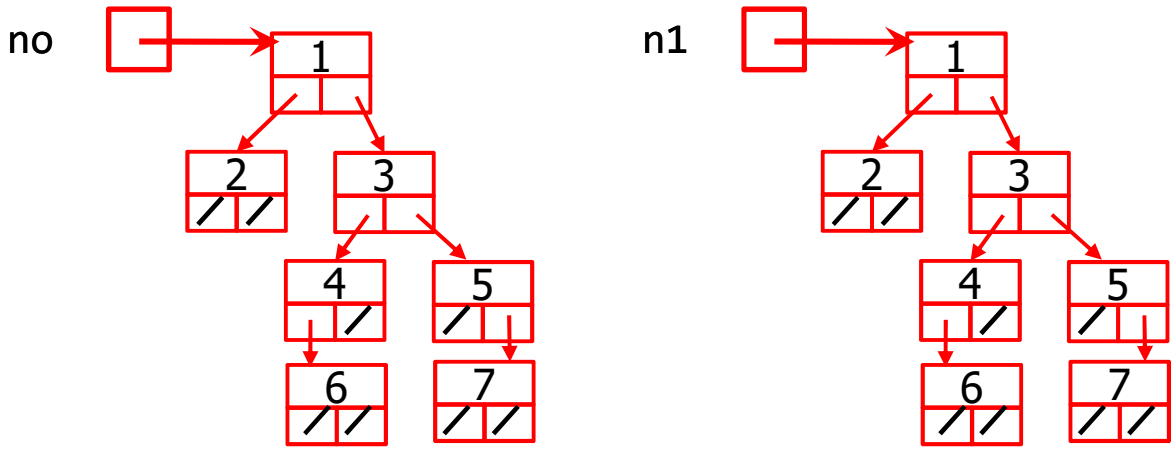
Implementación de árboles binarios mediante nodos

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Comparan dos estructuras?
  - Genera una copia de una estructura?
  - Libera la memoria ocupada por una estructura?

Implementación de árboles binarios mediante nodos

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Comparan dos estructuras?

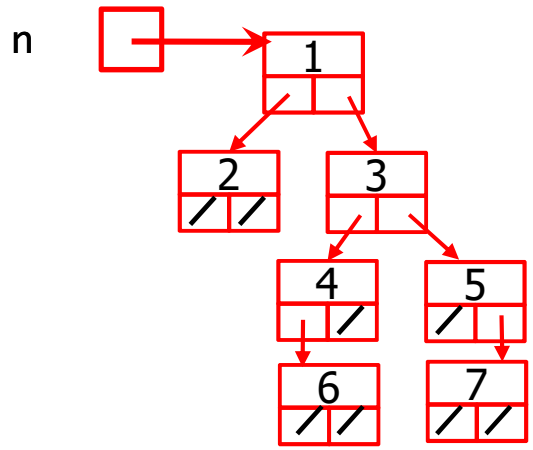
```
static bool sonIguales(const Nodo *no, const Nodo* n1) {
  return        no == n1 // mismo árbol (incluidos los vacíos)
                      ||
                no != NULL && n1 != NULL &&
                no->_elem == n1 ->_elem &&
                sonIguales(no->_iz, n1->_iz) &&
                sonIguales(no->_dr, n1->_dr);
}
```



Implementación de árboles binarios mediante nodos

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Genera una copia de una estructura?

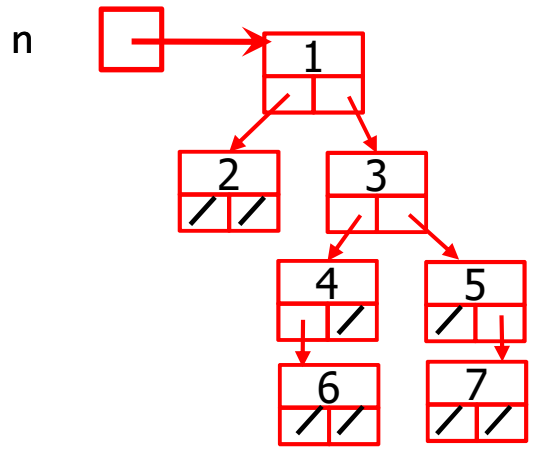
```
static Nodo* copia(const Nodo *n) {
  if (n == NULL) {
        return NULL;
  }
  else {
        return new Nodo(copia(n->_iz), n->_elem, copia(n->_dr));
  }
}
```



Implementación de árboles binarios mediante nodos

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Libera la memoria ocupada por una estructura?

```
static void libera(Nodo *n) {
   if (n != NULL) {
        libera(n->_iz);
        libera(n->_dr);
        delete n;
   }
}
```



Implementación de árboles binarios mediante nodos: operaciones

- Implementación de la operación `arbolVacio`:

```
Arbin() {
   _ra = NULL;
};
```

- Implementación de la operación `arbolSimple`:

```
Arbin(const T &elem) {
   _ra = new Nodo(elem);
};
```

- Implementación de la operación `cons`:

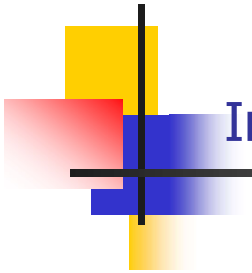
```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) {
    _ra = new Nodo(copia(iz._ra), elem, copia(dr._ra));
};
```

Implementación de árboles binarios mediante nodos: operaciones

- ¿Por qué no implementar la operación `cons` como

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) {
    _ra = new Nodo(iz._ra , elem, dr._ra);
};
```

?



Implementación de árboles binarios mediante nodos: operaciones

- Implementación de `esVacio`

```
bool esVacio() const {
   return _ra == NULL;
}
```

- Implementación de `raiz`

```
const T & raiz() const {
   if (_ra == NULL) {
      throw EArbolVacio(); // excepción de árbol vacío
   }
   return _ra->_elem;
}
```

Implementación de árboles binarios mediante nodos: operaciones

- Implementación `hijoIz` e `hijoDer`

```
Arbin hijoIz() const {
    if(_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_iz);
}

Arbin hijoDer() const {
    if(_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_dr);
}
```

Hacen uso de constructor privado que genera un árbol a partir de una estructura arborescente de nodos:

```
Arbin (Nodo* ra) {
   _ra = copia(ra);
}
```

Implementación de árboles binarios mediante nodos: operaciones

- Implementación de la relación de equivalencia: sobrecarga del operador ==

```
bool operator==(const Arbin& a) const {
        return sonIguales(_ra, a._ra);
}
```

- Sobrecarga del operador =

```
Arbin& operator=(const Arbin& a) {
        if (this != &a) {
                libera(_ra);
                _ra = copia(a._ra);
        }
        return *this;
}
```

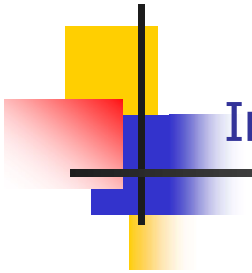
Implementación de árboles binarios mediante nodos: operaciones

- Constructor de copia

```
Arbin(const Arbin& a) {
        _ra = copia(a._ra);
}
```

- Destructor

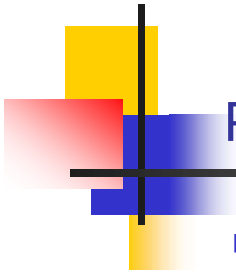
```
~Arbin() {
        libera(_ra);
}
```



Implementación de árboles binarios mediante nodos: operaciones

Operación	Coste (*)
arbolVacio	O(1)
arbolSimple	O(1)
cons	O(n), con *n* la suma de los nodos de los árboles dados como argumentos
esVacio	O(1)
raiz	O(1)
hijoIz, hijoDer	O(n), con *n* el número de nodos en el árbol resultante

(*) Asumimos que el tipo de los elementos tiene operaciones destrucción y copia O(1)



Recorridos de árboles binarios

- Un procesamiento típico de los árboles consiste en recorrer sus nodos en un determinado orden, realizando operaciones sobre los valores durante dicho recorrido. Cada nodo se visita exactamente una vez.
- Recorridos más habituales:
  - Recorrido en **preorden**: (i) Se visita la raíz, (ii) se recorre en **preorden** el subárbol izquierdo, (ii) se recorre en **preorden** el subárbol derecho
  - Recorrido en **postorden**: (i) Se recorre en **postorden** el árbol izquierdo, (ii) se recorre en **postorden** el árbol derecho, (iii) se visita la raíz
  - Recorrido en **inorden**: (i) Se recorre en **inorden** el árbol izquierdo, (ii) se visita la raíz, (iii) se recorre en **inorden** el árbol derecho
  - Recorrido **por niveles** (o **en anchura**): Se comienza procesando los nodos de nivel 1, después los de nivel 2 (de izquierda a derecha), después los de nivel 3 (de izquierda a derecha) … y así sucesivamente.

Recorridos de árboles binarios

- Teniendo en cuenta la definición recursiva de un árbol
  - Un árbol es vacío, o
  - Tiene una raíz, un subárbol derecho y un subárbol izquierdo

se puede construir un algoritmo recursivo de recorrido en preorden como sigue (y equivalentes en postorden e inorden):

- Si el árbol es vacío, entonces no hacer nada

  sino

  Visitar la raíz

  Recorrer el subárbol izdo

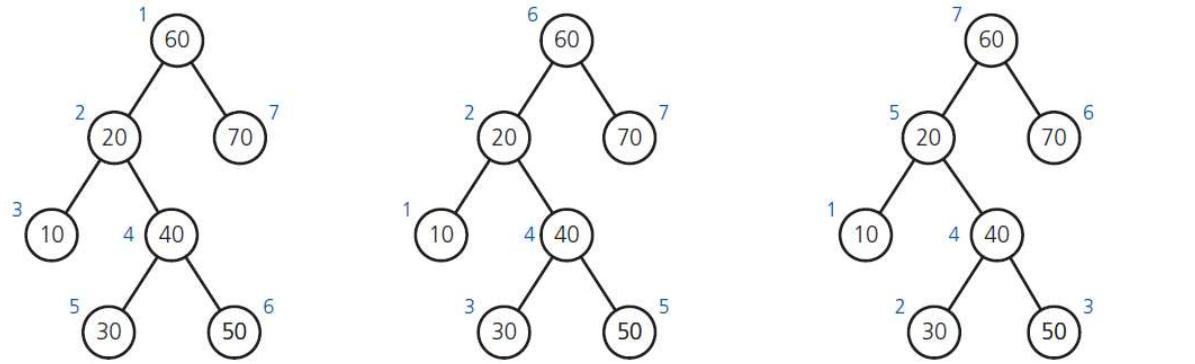
  Recorrer el subárbol dcho

caso base

paso inductivo

Recorridos de árboles binarios

- Se va a enriquecer el TAD `Arbin` con las siguientes operaciones adicionales:
  - `preorden`: Devuelve una lista con los valores de los nodos del árbol listados en preorden
  - `postorden`: Lista con los valores listados en postorden
  - `inorden`: Lista con los valores listados en inorden

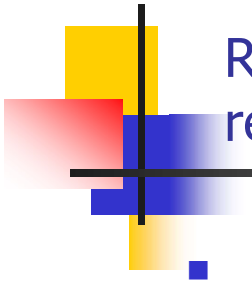


(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

- `niveles`: Lista con los valores listados por niveles

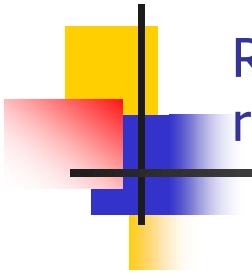
Figuras extraídas de Carrano, Data Abstraction and Problem Solving with C++: Walls and Mirrors



Recorridos de árboles binarios: implementación sobre la representación de nodos

- Implementación *naif* de `preorden`
  - Concatenar, a la lista que contiene la raíz, las listas que resultan de recorrer los subárboles izquierdo y derecho (en ese orden)
  - Dado que el TAD Lista no tiene una operación de concatenar, se implementa un método auxiliar que concatene listas

```
static void concatena(Lista<T>& l1, const Lista<T>& l2) {
  typename Lista<T>::ConstIterator il2 = l2.cbegin();
  for(int i=0; i < l2.longitud(); i++) {
      l1.pon_final(il2.elem());
      il2.next();
   }
}
```

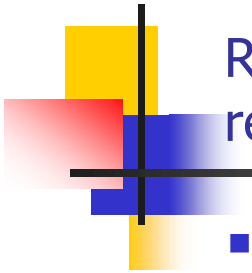


Recorridos de árboles binarios: implementación sobre la representación de nodos

- Se usa dicho método auxiliar para *sintetizar* el listado en preorden:

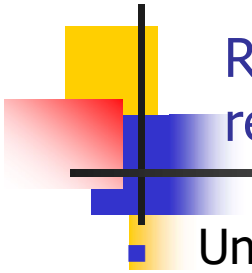
```
Lista<T> preordenNaif() const {
     return preordenNaifAux(_ra);
}

static Lista<T> preordenNaifAux(Nodo *n) {
  if (n == NULL)   return Lista<T>();
  else {
     Lista<T> resul;
     resul.pon_ppio(n->_elem);
     Lista<T> riz = preordenNaifAux(n->_iz);
     concatena(resul,riz);
     Lista<T> rdr = preordenNaifAux(n->_dr);
     concatena(resul,rdr);
     return resul;
  }
}
```



Recorridos de árboles binarios: implementación sobre la representación de nodos

- Problemas de la implementación *naif*:
  - Concatenar una lista a otra es costoso (se duplica la segunda lista: factor lineal) y además no es una operación disponible en el TAD Lista
  - Cuando se devuelve el resultado, éste se duplica a su vez (de nuevo otro factor lineal), y luego se destruye (otro factor lineal más) → tener gran cantidad de nodos temporales

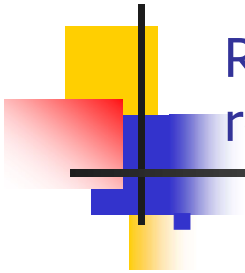


Recorridos de árboles binarios: implementación sobre la representación de nodos

- Una solución mejor: llevar la lista como un **parámetro de acumulación**, añadiendo los valores al final.

```
Lista<T> preorden() const {
      Lista<T> resul;
      preordenAux(_ra, resul);
      return resul;
}

static void preordenAux(Nodo *n, Lista<T>& resul) {
   if (n != NULL) {
         resul.pon_final(n->_elem);
         preordenAux(n->_iz, resul);
         preordenAux(n->_dr, resul);
   }
}
```



Recorridos de árboles binarios: implementación sobre la representación de nodos

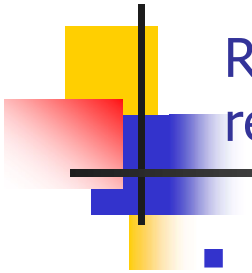
La implementación de `postorden` e `inorden` puede seguir la misma estrategia (únicamente varía el lugar en el que se añade la raíz a la lista)

```
Lista<T> postorden() const {
     Lista<T> resul;
     postordenAux(_ra, resul);
     return resul;
}

static void postordenAux(Nodo *n,
                         Lista<T>& resul) {
   if (n != NULL) {
      postordenAux(n->_iz,resul);
      postordenAux(n->_dr,resul);
      resul.pon_final(n->_elem);
   }
}
```

```
Lista<T> inorden() const {
     Lista<T> resul;
     inordenAux(_ra, resul);
     return resul;
}

static void inordenAux(Nodo *n,
                       Lista<T>& resul) {
   if (n != NULL) {
      inordenAux(n->_iz, resul);
      resul.pon_final(n->_elem);
      inordenAux(n->_dr, resul);
   }
}
```

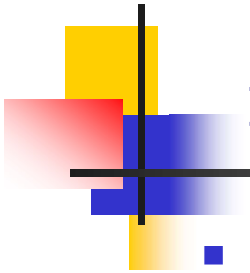


Recorridos de árboles binarios: implementación sobre la representación de nodos

- El recorrido por niveles sigue una estrategia diferente
- La idea básica es utilizar una **cola** de nodos a recorrer
- La cola se inicializa con el nodo raíz
- El algoritmo es iterativo:
  - El proceso termina cuando la cola de nodos a recorrer se vacía
  - En cada iteración:
    - Se extrae un nodo de la cola
    - Dicho nodo se procesa (en este caso, se añade a la lista)
    - Finalmente, se añaden a la cola los nodos hijo por orden (en este caso, primero el izquierdo, después el derecho, si es que existen)

Recorridos de árboles binarios: implementación sobre la representación de nodos

```
Lista<T> niveles() const {
   Lista<T> resul;
   Cola<Nodo*> pendientes;
   if (_ra != NULL) {  // si el arbol no es vacio
      pendientes.pon(_ra);
      while (! pendientes.esVacia()) {
         Nodo* actual = pendientes.primero();
         pendientes.quita();
         resul.pon_final(actual->_elem);
         if (actual->_iz != NULL) {
           pendientes.pon(actual->_iz);
         }
         if (actual->_dr != NULL) {
           pendientes.pon(actual->_dr);
         }
      }
   }
   return resul;
}
```

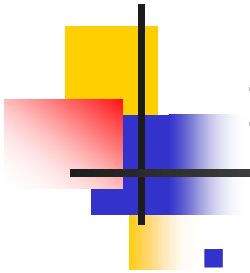


Implementación eficiente de los árboles binarios

- Con la implementación de `Arbin` descrita, ¿cuál es la complejidad de la siguiente función que halla la suma de los nodos de un árbol de enteros

```
int suma(const Arbin<int>&a) {
   if (a.esVacio()) return 0;
   else
     return a.raiz() +
            suma(a.hijoIz()) +
            suma(a.hijoDer());
}
```

?

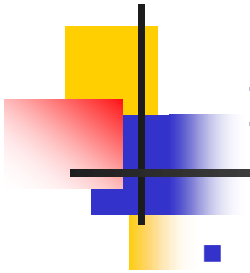


Implementación eficiente de los árboles binarios

- Volvamos a retomar una idea rechazada anteriormente…

  Dado que los árboles son objetos inmutables y que la devolución, p.e., del hijo izquierdo compartiendo nodos no es peligrosa en el sentido de que las modificaciones de un árbol afecten al contenido de otro, ¿por qué no compartir estructura?

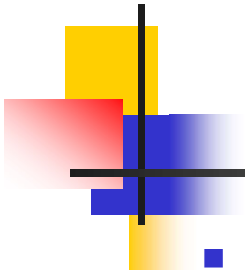
```
Arbin hijoIz() const {
    // NO VALIDO!!!!!!
   if (_ra == NULL) { throw EArbolVacio(); }
   Arbin iz;
   iz._ra = _ra->_iz;
   return iz;
}
```



Implementación eficiente de los árboles binarios

- Una alternativa es llevar en el nodo un **contador de referencias**: un entero que indica cuántos punteros lo referencian.

```
class Nodo {
public:
   Nodo() : _iz(NULL), _dr(NULL), _nrefs(0) {}
   Nodo(const T &elem) :
     _elem(elem),_iz(NULL),_dr(NULL), _nrefs(0) {}
   Nodo(Nodo *iz, const T &elem, Nodo *dr) :
     _elem(elem), _iz(iz), _dr(dr), _nrefs(0) {
          if (iz != NULL) iz->addRef();
          if (dr != NULL) dr->addRef();
        }
   void addRef() {_nrefs++;};
   void rmRef()  {_nrefs--;}
   T _elem;          // valor del nodo
   Nodo* _iz;        // hijo izquierdo
   Nodo* _dr;        // hijo derecho
   int _nrefs;       // número de referencias al nodo
};
```



Implementación eficiente de los árboles binarios

- ¿Qué pasa con las operaciones para el manejo de la estructura de datos arbórea que vimos para…
  - Comparar dos estructuras? → se mantiene igual
  - Generar una copia de una estructura? → ¡¡ahora no hace falta!!
  - Liberar la memoria ocupada por una estructura?
    - El método de liberación decrementará en 1 el contador de referencias, y liberará el nodo únicamente cuando dicho contador se haga 0

```
static void libera(Nodo *n) {
    if (n != NULL) {
      n->rmRef();
      if (n->_nrefs == 0) {
         libera(n->_iz);
         libera(n->_dr);
         delete n;
      }
    }
}
```

```
static void libera(Nodo *n) {
    if (n != NULL) {
        libera(n->_iz);
        libera(n->_dr);
       delete n;
        }}}
```

Implementación eficiente de árboles binarios: operaciones

- Implementación de la operación `arbolVacio` → no cambia

```
Arbin() {
   _ra = NULL;
};
```

```
Arbin() {
     _ra = NULL;
};
```

- Implementación de la operación `arbolSimple` → tiene que incluir el incremento del contador de referencias del nodo recién creado

```
Arbin(const T &elem) {
    _ra = new Nodo(elem);
    _ra->addRef();
};
```

```
Arbin(const T &elem) {
     _ra = new Nodo(elem);
};
```

Implementación eficiente de árboles binarios: operaciones

- Implementación de la operación `cons` → se aplica la compartición de memoria (no es necesaria la copia profunda de la estructura) e incluye el incremento del contador de referencias del nodo recién creado

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) {
    _ra = new Nodo(iz._ra, elem, dr._ra);
    _ra->addRef();
};
```

```
Arbin(const Arbin &iz, const T &elem, const Arbin &dr) {
    _ra = new Nodo(copia(iz._ra), elem, copia(dr._ra));
};
```

Implementación eficiente de árboles binarios: operaciones

- Implementación de `esVacio` → no cambia

```
bool esVacio() const {
   return _ra == NULL;
}
```


```
bool esVacio() const {
   return _ra == NULL;
}
```


- Implementación de `raiz` → no cambia

```
const T & raiz() const {
   if(_ra == NULL) {
        throw EArbolVacio();
   }
   return _ra->_elem;
}
```


```
const T & raiz() const {
   if(_ra == NULL) {
        throw EArbolVacio();
   }
   return _ra->_elem;
}
```

Implementación eficiente de árboles binarios: operaciones

- Implementación de `hijoIz` e `hijoDer` → cambia el constructor privado en el que delegan ya que no necesitan copiar los subárboles (pero sí incrementar las referencias a los nodos compartidos)

```
Arbin hijoIz() const {
    if (_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_iz);
}

Arbin hijoDer() const {
    if (_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_dr);
}
```

Hacen uso de constructor privado que genera un árbol a partir de una estructura arborescente de nodos:

```
Arbin(Nodo* ra) {
  _ra = ra;
  if (_ra != NULL)
        _ra->addRef();
}
```

```
Arbin (Nodo* ra) {
    _ra = copia(ra);
}
```

```
Arbin hijoIz() const {
    if(_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_iz);
}
```

```
Arbin hijoDer() const {
    if(_ra == NULL) {
        throw EArbolVacio();
    }
    return Arbin(_ra->_dr);
```

Implementación eficiente de árboles binarios: operaciones

- Implementación de la relación de equivalencia: sobrecarga del operador == → no cambia

```
bool operator==(const Arbin& a) const {
        return sonIguales(_ra, a._ra);
}
```

- Sobrecarga del operador = → no necesita copiar el árbol (pero sí incrementar el contador de referencias del nodo apuntado)

```
Arbin& operator=(const Arbin& a) {
        if (this != &a) {
                libera(_ra);
                if (a._ra != NULL) a._ra->addRef();
                _ra = a._ra;
        }
        return *this;
}
```

```
Arbin& operator=(const Arbin& a) {
  if (this != &a) {
          libera(_ra);
          _ra = copia(a._ra);
  }
  return *this;
}
```

Implementación eficiente de árboles binarios: operaciones

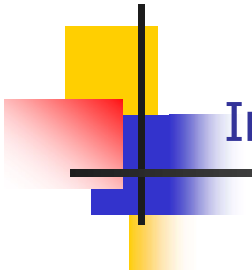
- Constructor de copia → no necesita copiar el árbol (pero sí incrementar el contador de referencias del nodo apuntado)

```
Arbin(const Arbin& a) {
    if (a._ra != NULL) a._ra->addRef();
    _ra = a._ra;
}
```

```
Arbin(const Arbin& a) {
  _ra = copia(a._ra);
}
```

- Destructor → no cambia (aunque sí cambió `libera`)

```
~Arbin() {
      libera(_ra);
}
```

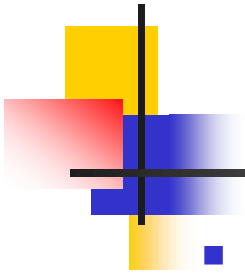


Implementación eficiente de árboles binarios: operaciones

Operación	Coste (*)
arbolVacio	O(1)
arbolSimple	O(1)
cons	O(1)
esVacio	O(1)
raiz	O(1)
hijoIz, hijoDer	O(1)

¡Y además, reducción en el consumo de memoria!

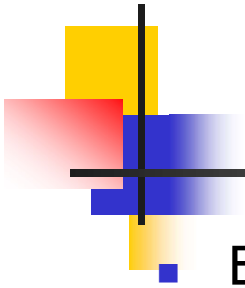
(*) Asumimos que el tipo de los elementos tiene operaciones destrucción y copia O(1)



Implementación eficiente de árboles binarios y recorridos

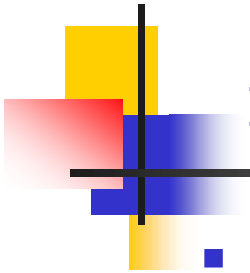
- Implementación de `preorden` como función externa, utilizando las operaciones del TAD
  - Con la implementación optimizada, la complejidad es O(n), frente a la complejidad $O(n^2)$ que tendría si se usara la implementación sin optimizar

```
template <class T>
Lista<T> preorden(const Arbin<T>&a) {
  Lista<T> resul;
  preordenAux(a, resul);
  return resul;
}
template <class T>
void preordenAux(const Arbin<T>&a, Lista<T>& resul) {
  if (!a.esVacio()) {
        resul.pon_final(a.raiz());
        preordenAux(a.hijoIz(), resul);
        preordenAux(a.hijoDer(), resul);
  }
}
```



Implementación eficiente de árboles binarios: mejoras

- El manejo explícito de los contadores de referencias puede ser propenso a errores
- En su lugar, pueden usarse **punteros inteligentes** que automatizan la gestión del conteo de referencias
  - En lugar de un puntero, se usa un objeto que encapsula el puntero, y que, además, cuando cambia de valor decrementa el contador de referencias del antiguo objeto apuntado, e incrementa el contador de referencias del nuevo objeto

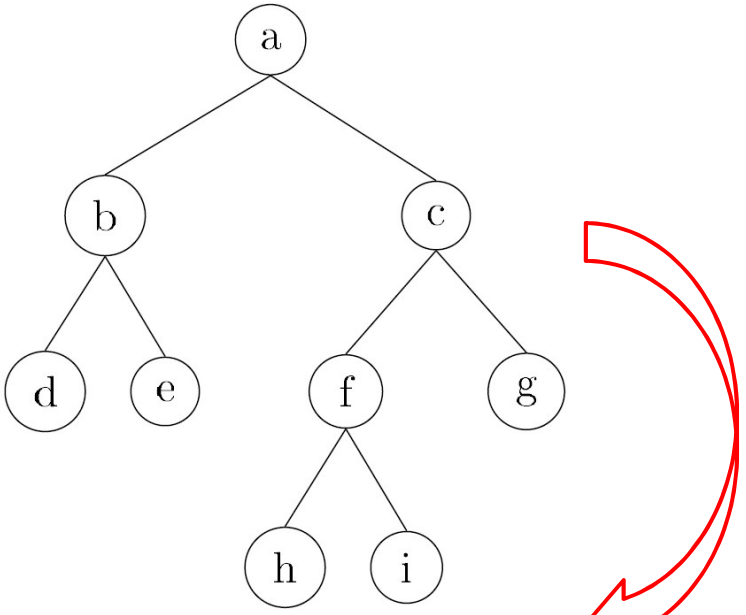


Implementación estática de árboles binarios

- La implementación anterior usando estructura arbórea de nodos con memoria compartida da lugar a árboles inmutables que se crean "de abajo a arriba"
- Una alternativa de implementación, que permitiese construir árboles "de arriba a abajo", añadiendo nodos a los ya existentes, se puede conseguir usando arrays
  - En lugar de utilizar memoria dinámica, los nodos del árbol se almacenan en un array
  - La ventaja adicional es que se evita el coste asociado a las operaciones de manejo de la memoria dinámica (**new** y **delete**)
- Cada elemento del array será un objeto del tipo:

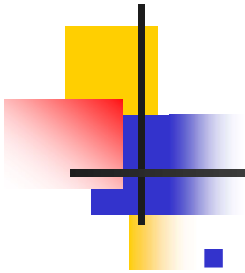
```
class InfoNode {
public:
...
private:
   T _elem;
   int indexIz; // -1 si no hay hijo izquierdo
   int indexDr; // -1 si no hay hijo derecho
};
```

Implementación estática de árboles binarios



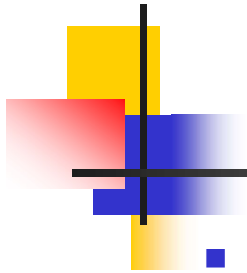
a		b		c		f		h		g		d		e		i		...
1	2	6	7	3	5	4	8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

- Se supone que el orden de inserción ha sido a, b, c, f, h, g, d, e, i.



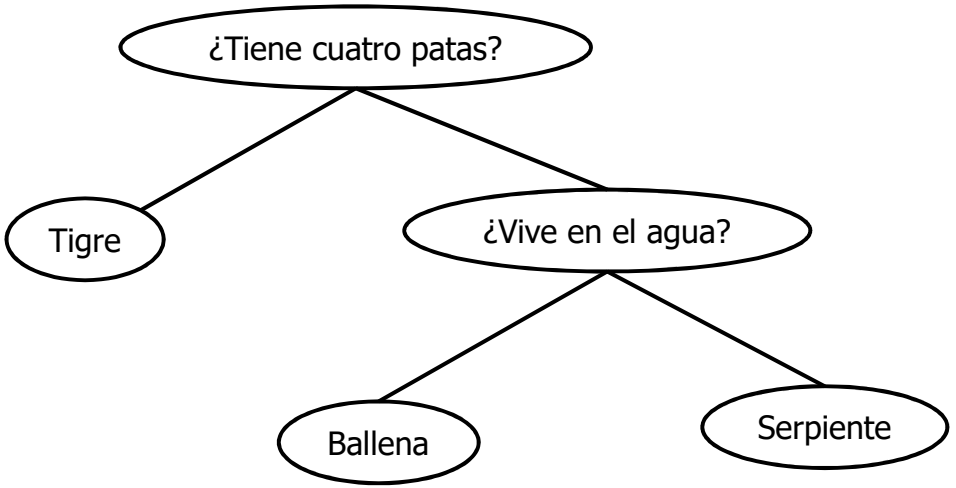
Árboles n-arios y generales

- Las representaciones son una generalización de las vistas para árboles binarios
- Es necesario disponer de una estructura secuencial que permita tener más de dos hijos (un array o una lista de hijos)
  - Para árboles **n-arios**, basta un array de tamaño n
  - Para árboles generales, en los que no se determina a priori un número máximo de hijos, puede usarse una lista de punteros a los hijos
  - Otra alternativa para ambos casos es que cada nodo apunte a su hermano (p.e., al derecho). De esta forma, basta con mantener únicamente un puntero al hijo (p.e., el de más a la izquierda)
    - Para acceder al hijo i-ésimo de un nodo se accede al hijo y se recorren i-1 punteros al hermano



Implementación del juego de las adivinanzas

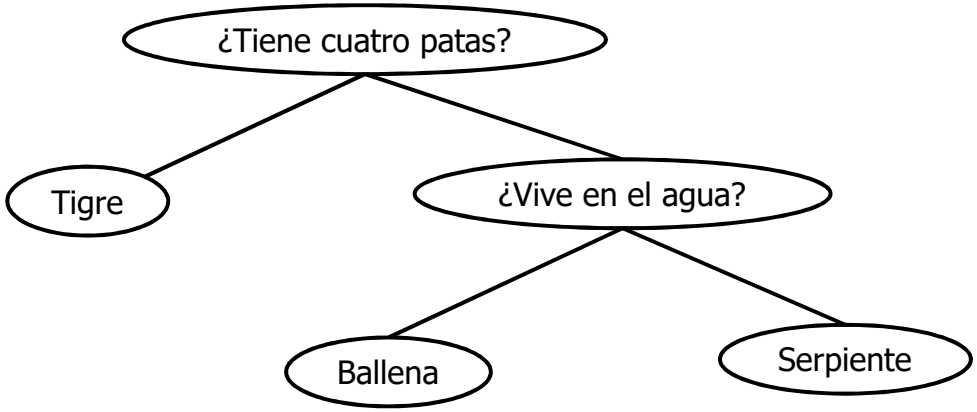
- El juego inicialmente planteado puede implementarse mediante un árbol binario de **string**s
  - Los nodos interiores serán las preguntas
  - Los nodos hoja se corresponderán con los animales
  - El subárbol izquierdo representa la interacción si la respuesta es positiva; el derecho si es negativa

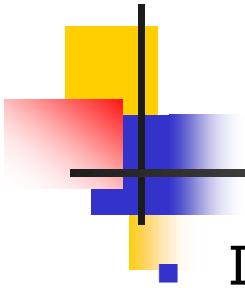


Implementación del juego de las adivinanzas

- Construcción del árbol

```
Arbin<string> bd =
    Arbin<string>(
            Arbin<string>("Tigre"), // subarbol izdo
            "Tiene cuatro patas?", // raíz del árbol
            Arbin<string>( // subarbol dcho
                    Arbin<string>("Ballena"),
                    "Vive en el agua?",
                    Arbin<string>("Serpiente")
            )
    );
```

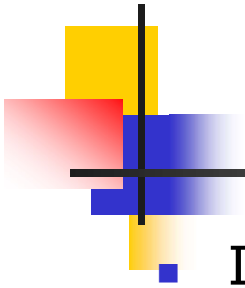




Implementación del juego de las adivinanzas

- Implementación de una partida "sin aprendizaje" (solución iterativa)

```
bool esHoja(const Arbin<string>& bd) {
   return bd.hijoIz().esVacio() && bd.hijoDer().esVacio();
}
void juegaIterativo(const Arbin<string>& bd){
   Arbin<string> auxiliar = bd;
   while (!esHoja(auxiliar)){
     cout << auxiliar.raiz() << "(si/no)" << endl;
     string linea;
     getline(cin, linea);
     if (linea == "si") {auxiliar = auxiliar.hijoIz();}
     else {auxiliar = auxiliar.hijoDer();}
   }
   cout << "Mmmmmm, dejame que piense.....\n";
   cout << "Creo que el animal en que estabas pensando era...:" <<
   auxiliar.raiz() << "!\n";
   cout << "Espero haber acertado... Gracias por jugar conmigo" << endl;
}
```



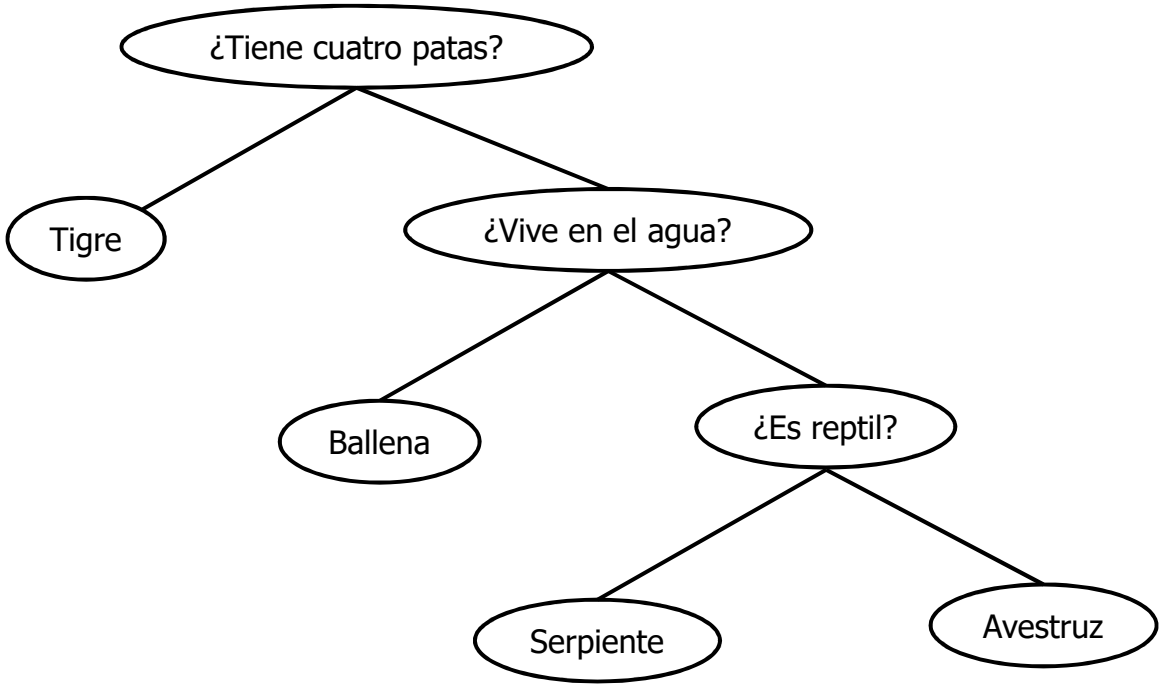
Implementación del juego de las adivinanzas

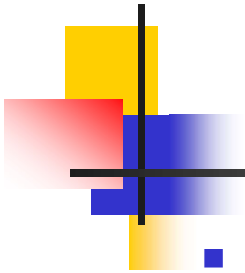
- Implementación de una partida "sin aprendizaje" (solución recursiva)

```
void juegaRecursivo(const Arbin<string>& bd) {
   if (esHoja(bd)) {
     cout << "Mmmmmm, dejame que piense.....\n";
     cout << "Creo que el animal en que estabas pensando era...:"
          << bd.raiz() << "!\n";
     cout << "Espero haber acertado... Gracias por jugar conmigo" << endl;
   } // Fin del tratamiento de los nodos hoja
   else {
     cout << bd.raiz() << "(si/no)" << endl;
     string linea;
     getline(cin, linea);
     if (linea == "si") { juegaRecursivo(bd.hijoIz()); }
     else { juegaRecursivo(bd.hijoDer()); }
   }
}
```

Implementación del juego de las adivinanzas

- Implementación de una partida "con aprendizaje"
  - La partida puede dar lugar a un nuevo árbol
  - El aprendizaje consiste en que, en el árbol resultante, la hoja con la que se ha terminado la partida debe "sustituirse" por un nodo interno con la pregunta que permite discriminar y dos hojas (uno con el animal conjeturado –la antigua hoja– y otro con el animal en el que realmente pensó el usuario)





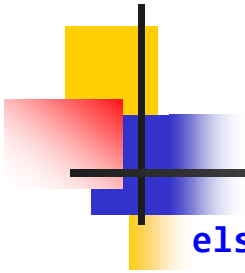
Implementación del juego de las adivinanzas

- Implementación de una partida "con aprendizaje"

```
void juegaAprendiendo(const Arbin<string>& bd,
                      Arbin<string>& nueva_bd, bool& bd_modificada) {
  Arbin<string> resul;
  bd_modificada = false;
  if (esHoja(bd)) {
    cout << "Mmmmmm, dejame que piense.....\n";
    cout << "Creo que el animal en que estabas pensando era...:"
         << bd.raiz() << "!\n";
    cout << "Acerte? (si/no):\n";
    string respuesta; getline(cin,respuesta);
```

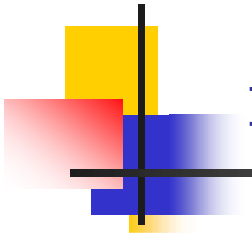
Implementación del juego de las adivinanzas

```
if (respuesta == "no") {  // comienza el aprendizaje
   cout << "De que animal se trata?" << endl;
   string animal; getline(cin,animal);
   cout << "Que pregunta tendria que hacer para "
           << "distinguirlo de " << bd.raiz() << "?" << endl;
   string pregunta; getline(cin,pregunta);
   cout << "En este caso, la respuesta seria " << bd.raiz()
          << "? (si/no)“  << endl;
   string respuesta; getline(cin,respuesta);
   if (respuesta == "si") {
      nueva_bd = Arbin<string>(bd, pregunta, Arbin<string>(animal));
   }
   else {
     nueva_bd = Arbin<string>(Arbin<string>(animal), pregunta, bd);
   }
   bd_modificada = true;
}
cout << "Estupendo! Gracias por jugar conmigo." << endl;
} // Fin del tratamiento de los nodos hoja
```



Implementación del juego de las adivinanzas

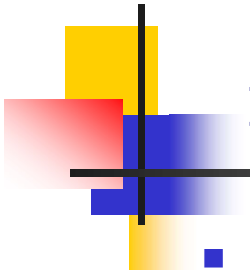
```
    else { // estamos en un nodo interno
            cout << bd.raiz() << "(si/no)" << endl;
            string linea; getline(cin,linea);
            if (linea == "si") {
               Arbin<string> nuevo_hijo_iz;
               juegaAprendiendo(bd.hijoIz(), nuevo_hijo_iz, bd_modificada);
               if (bd_modificada) {
                 nueva_bd = Arbin<string>(nuevo_hijo_iz, bd.raiz(), bd.hijoDer());
               }
            }
            else {
               Arbin<string> nuevo_hijo_der;
               juegaAprendiendo(bd.hijoDer(), nuevo_hijo_der, bd_modificada);
               if(bd_modificada) {
                 nueva_bd = Arbin<string>(bd.hijoIz(), bd.raiz(), nuevo_hijo_der);
               }
            }
    }
}
```



Implementación del juego de las adivinanzas

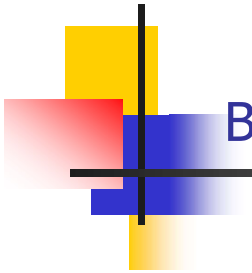
```
int main(){
     Arbin<string> bd =
          Arbin<string>(
                    Arbin<string>("Tigre"), // subarbol izdo
                    "Tiene cuatro patas?", // raíz del árbol
                    Arbin<string>( // subarbol dcho
                              Arbin<string>("Ballena"),
                              "Vive en el agua?",
                              Arbin<string>("Serpiente")));
     string respuesta;

     do {
          Arbin<string> nueva_bd;
          bool modificada;
          juegaAprendiendo(bd, nueva_bd, modificada);
          if (modificada) {bd = nueva_bd;}
          cout << "Otra partida? (si/no)" << endl;
          getline(cin, respuesta);
    } while (respuesta == "si");
    return 0;
}
```



Implementación del juego de las adivinanzas

- En caso de añadir un nuevo animal, es necesario construir un nuevo árbol
- Aunque la compartición amortigua el coste, se construirán tantos nodos como longitud tenga la rama seguida (en el peor caso, tantos nodos como tiene el árbol)
- La construcción explícita de un nuevo árbol con cada jugada revela una de las limitaciones del TAD: no permite modificar la estructura interna del árbol
- Una implementación más completa del TAD permitiría llevar a cabo dicha modificación directa de la estructura
- Si bien la complejidad asintótica no cambiaría, en la práctica la implementación sería bastante más eficiente



Bibliografía del tema

- Apuntes ISBN 978-84-697-0852-1
- Fundamentals of Data Structures in C++ / Horowitz, Sahni & Mehta / Computer Science Press / 1995
- Data abstraction & Problem Solving with C++: Walls and Mirrors, 6th edition / Frank Carrano & Timothy Henry / Pearson, 2013
- ADTs, Data Structures, and Problem Solving with C++, 2nd edition / Larry Nyhoff / Pearson – Prentice Hall, 2005
- Estructuras de datos y métodos algorítmicos: Ejercicios resueltos / Martí Oliet, Ortega Mallén y Verdejo López / Pearson – Prentice Hall, 2010

Acerca de Creative Commons

Licencia CC (Creative Commons)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

En http://es.creativecommons.org/ y http://creativecommons.org/ puedes saber más de Creative Commons.