Estructuras de Datos y Algoritmos 1

Estructuras Arborescentes

Carlos Luna

Definición

La recursión puede ser utilizada para la definición de estructuras realmente sofisticadas.

Una estructura *árbol* con tipo base T es,

1. O bien la estructura vacía
2. O bien un nodo de tipo T junto con un número finito de estructuras de árbol, de tipo base T, disjuntas, llamadas *subárboles*

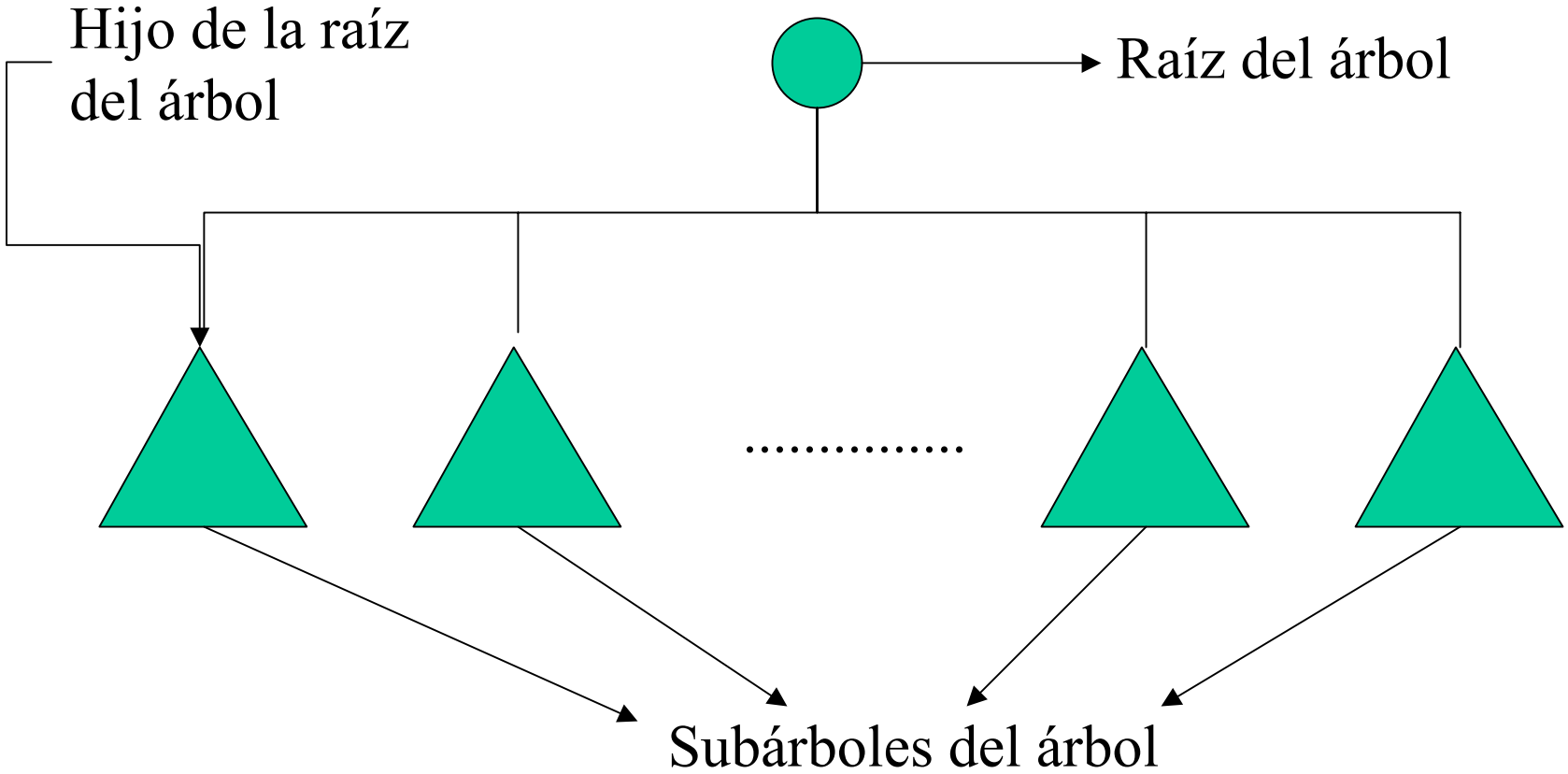
Conceptos básicos

Resulta evidente, a partir de la definición de secuencias (listas) y estructura árbol, que la lista es una estructura árbol en la cual cada nodo tiene a lo sumo un subárbol.

La lista es por ello también conocida por el nombre de árbol degenerado.

Veremos a continuación un poco de nomenclatura:

Conceptos básicos (cont.)



Los elementos se llaman habitualmente nodos del árbol.

Arboles binarios

La descripción de la noción de árbol dada antes es casi una definición inductiva.

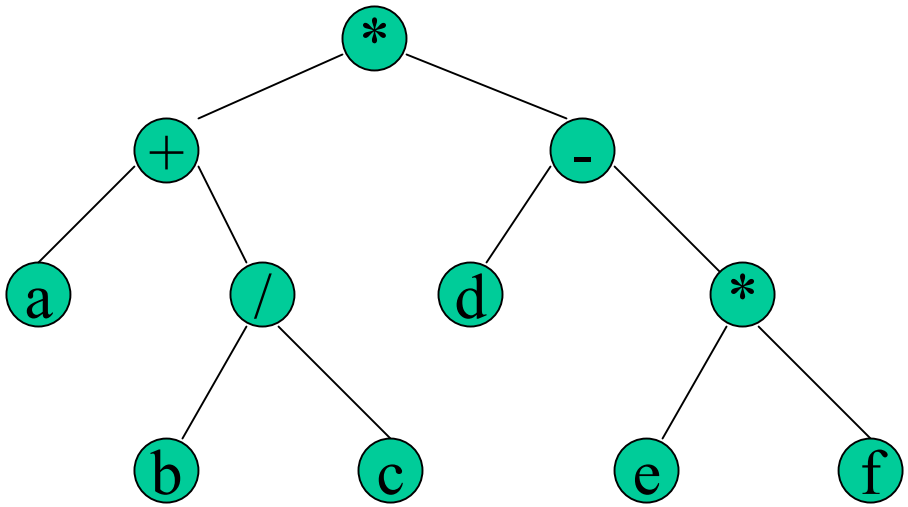
Falta precisar qué se quiere decir con “un número finito ...” en la segunda cláusula de construcción de árboles.

Existe un caso particular de árbol en que esta cláusula se hace precisa fácilmente:

“... junto con exactamente 2 (dos) subárboles binarios.”

Ejemplo: árbol de expresiones

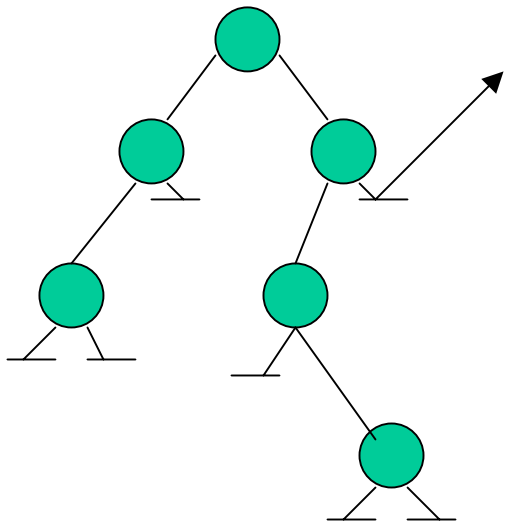
Representación no ambigua de expresiones aritméticas.



Representación arborescente de la fórmula:

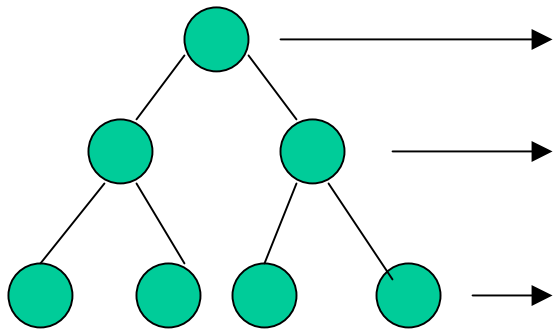
(a + b / c) * (d - e * f)

Hojas



Def: *Hojas* son los nodos cuyos (ambos) subárboles son vacíos

Niveles y profundidad

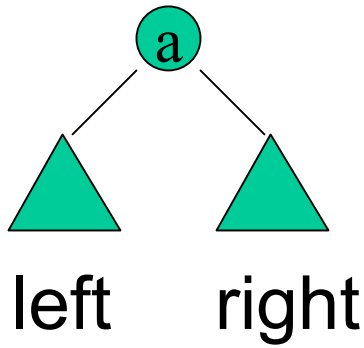


Def.
La *profundidad* de un árbol es:
la cantidad de niveles que tiene, o
la cantidad de nodos en el camino más largo de la raíz a una hoja.

La profundidad del árbol binario vacío es 0.

Profundidad (cont.)

La profundidad de un árbol de la forma:



es 1 + max(p_i, p_d), donde p_i es la profundidad de left y p_d es la profundidad de right.

Más formalmente:

Profundidad (cont.)

Definición Inductiva de Árboles Binarios

$$\frac{}{() : \text{ArbBin}} \qquad \frac{\text{left} : \text{ArbBin} \quad t : T \quad \text{right} : \text{ArbBin}}{(\text{left}, t, \text{right}) : \text{ArbBin}}$$

```
prof (()) = 0
prof ((left,a,right)) =
          1 + max(prof(left), prof(right))
```

Recorridas de árboles binarios

- En general, son procedimientos que visitan todos los nodos de un árbol binario efectuando cierta acción sobre cada uno de ellos
- La forma de estos procedimientos depende del orden que se elija para visitar los nodos
- Obviamente recorrer un árbol vacío es trivial

Recorridas de árboles binarios (cont.)

Para recorrer un árbol no vacío hay tres órdenes naturales, según la raíz sea visitada:

- antes que los subárboles
  (PreOrden - preorder)
- entre las recorridas de los subárboles
  (EnOrden - inorder)
- después de recorrer los subárboles
  (PostOrden - postorder)

Ejemplo

Formar la lista de los elementos de un árbol binario que se obtiene al recorrerlo en orden (linealización, secuenciamiento, listado):

```
enOrden (()) = []
enOrden ((left,a,right))
    = enOrden(left) ++ (a . enOrden(right))

    = enOrden(left) ++ [a] ++ enOrden(right)
```

Implementación en C++

Ahora presentaremos una posible implementación del tipo de dato árbol binario (ArbBin) en C++. Asumimos que los elementos del árbol son de tipo T.

```
struct arbbinNode {
    T info;
    struct arbbinNode *left, *right;
};
typedef struct arbbinNode ABNode;
typedef ABNode* ABNodePtr;
```

Recorridas

Formularemos ahora los 3 métodos de recorrida de árbol previamente presentados como procedimientos C++.

Usaremos un parámetro explícito t, que denotará el árbol sobre el cual se efectuará la recorrida.

Usaremos también un parámetro implícito P, que denotará la operación a ser aplicada sobre los elementos del árbol.

Procedimiento preOrden

```
void preOrden (ABNodePtr t){
      if (t != NULL)
         { P(t);
           preOrden(t -> left);
           preOrden(t -> right);
          };
};
```

Procedimiento enOrden

```
void enOrden (ABNodePtr t){
      if (t != NULL)
         { enOrden(t -> left);
           P(t);
           enOrden(t -> right);
          };
};
```

Procedimiento postOrden

```
void postOrden (ABNodePtr t){
      if (t != NULL)
         { postOrden(t -> left);
           postOrden(t -> right);
           P(t);
          };
};
```

Cantidad de nodos de un árbol

```
cantNodos (()) = 0
cantNodos ((left,a,right)) =
      1 + cantNodos(left) + cantNodos(right)
```

```
int cantNodos (ABNodePtr t){
      if (t == NULL) return 0;
      else
         return 1 + cantNodos(t -> left)
                  + cantNodos(t -> right);
};
```

Espejo

```
espejo (()) = ()
espejo ((left,a,right)) =
  (espejo(right),a,espejo(left))

ABNodePtr espejo (ABNodePtr t){
    if (t == NULL) return NULL;
    else
       { ABNodePtr rt = new ABNode;
         rt -> info = t -> info;
         rt -> left  = espejo (t -> right);
         rt -> right = espejo (t -> left);
         return rt;
       };
};
```

Hojas de un árbol

```
hojas (()) = []

hojas ((),a,()) = [a]

hojas ((left,a,right)) =
                  hojas[left] ++ hojas[right],
                  Si left≠() o right≠()
```

```
PtrNodoLista hojas (ABNodePtr t) {
PtrNodoLista p;
 if (t == NULL) return NULL;
 else
    if ((t -> left == NULL)
        && (t -> right == NULL))
       { p = new NodoLista;
         p -> info = t -> info;
         p -> sig  = NULL;
         return p;
       };
    else return Concat(hojas(t -> left),
                       hojas(t -> right));
}; //
```

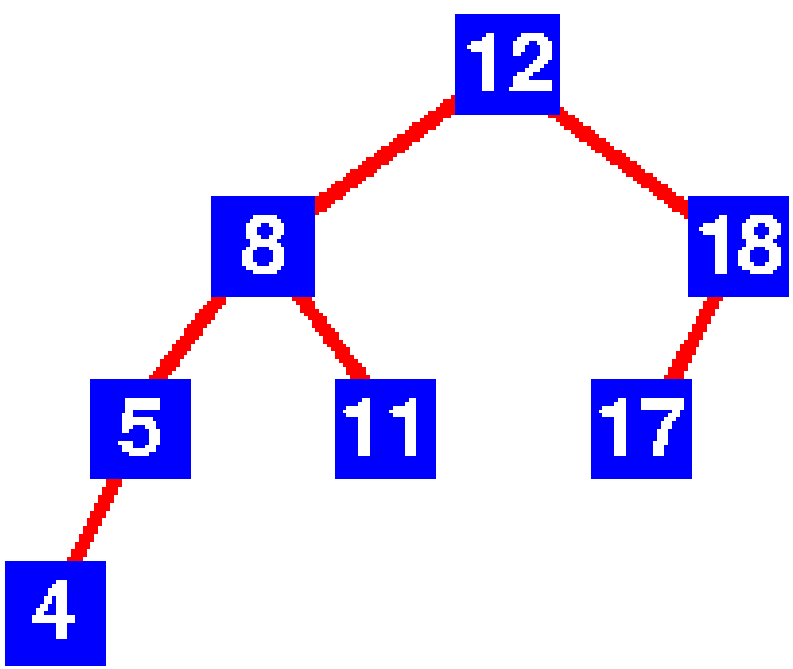
Nota: **Concat** es aquí una función (no un proced.)

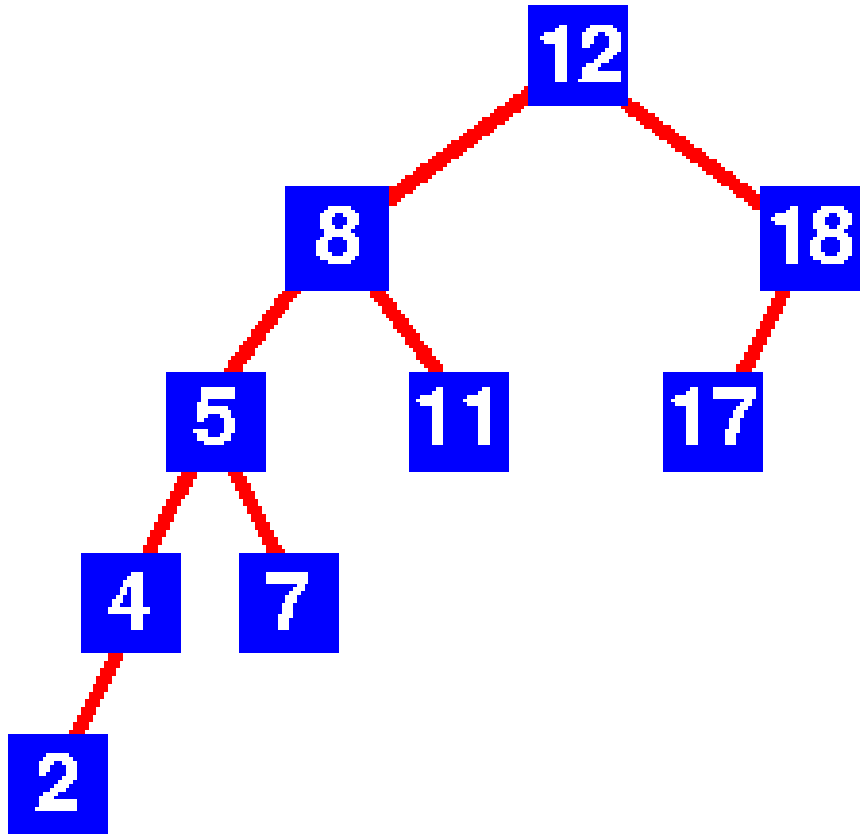
Arbol binario de búsqueda (ABB)

Arboles binarios son frecuentemente usados para representar conjuntos de datos cuyos elementos pueden ser recuperables (y por lo tanto identificables) por medio de una clave única.

Si un árbol está organizado de forma tal que para cada nodo n_i, todas las claves en el subárbol izquierdo de n_i son menores que la clave de n_i, y todas las claves en el subárbol derecho de n_i son mayores que la clave de n_i, entonces este árbol es un *ABB*.

Arbol binario de búsqueda (ABB): Ejemplos





Implementación de ABB

La implementación del tipo *ABB* es muy similar a la de ArbBin. La diferencia es que ahora diferenciamos un campo, **`key`**, cuyo tipo es un ordinal (**`Ord`**), del resto de la información del nodo:

```
struct abbNodo {
    Ord key;
    T info;
    struct abbNodo *left, *right;
};
typedef struct abbNodo ABBNodo;
typedef ABBNodo *ABBNodoPtr;
```

Búsqueda binaria

En un ABB es posible localizar (encontrar) un nodo de clave arbitraria comenzando desde la raíz del árbol y recorriendo un camino de búsqueda orientado hacia el subárbol izquierdo o derecho de cada nodo del camino dependiendo solamente del valor de la clave del nodo.

Como esta búsqueda sigue un único camino desde la raíz, puede ser fácilmente implementada en forma imperativa, como veremos a continuación:

Find iterativo

```
ABBNodoPtr find(Ord x; ABBNodoPtr t) {
bool found;
  found = false;
  while ((t != NULL) && !found){
    if (t -> key == x)
       found = true;
    else
        if (t -> key > x)
           t = t -> left;
        else t = t -> right;
    };
  return t;
};
```

Find recursivo

```
ABBNodoPtr find(Ord x; ABBNodoPtr t){
ABBNodoPtr res;
  if (t == NULL) res = NULL;
  else
   if (x == t->key) res = t;
   else
     if (x < t->key) res = find(x,t -> left);
     else res = find(x,t->right);
  return res;
};
```

Find procedural y recursivo

```
void find(Ord x; ABBNodoPtr t;
          bool & found; ABBNodoPtr & r) {
  if (t == NULL)
     {found = false; r = NULL};
  else
     if (x == t->key)
        {found = true; r = t};
     else if (x < t->key)
           find(x,t->left,found,r);
          else find(x,t->right,found,r);
};
```

Pertenencia

Hemos dicho que ABB se usan para representar conjuntos, o colección de elementos que son identificados únicamente por su clave.

Otra de las funciones características definidas sobre este tipo de árboles es la función que determina si existe un nodo del árbol cuya clave sea igual a una arbitraria dada (si el elemento pertenece al conjunto).

La función Member

```
bool member (Ord x; ABBNodoPtr t){
  if (t == NULL) return false;
  else
     if (x == t->key)
       return true;
     else
       return (member(x,t->left)
            || member(x,t->right));
};
```

Cuál es el orden O de tiempo de ejecución ?

Podría optimizarse la función member ?

Ejercicios Propuestos

- **Defina un procedimiento C++ Insert que dados un elemento t:T, su clave x:Ord y un ABB, inserte el elemento en el árbol, generando un ABB.**
- **Defina un procedimiento C++ DeleteMin que elimine el mínimo elemento de un ABB. El resultado debe ser un ABB.**
- **Piense un procedimiento C++ Delete que elimine un elemento dado de un ABB. El resultado debe ser un ABB.**

Bibliografía Recomendada

Como Programar en C/C++

H.M. Deitel & P.J. Deitel; Prentice Hall, 1994.

(o la Versión 2 del *Deitel*)