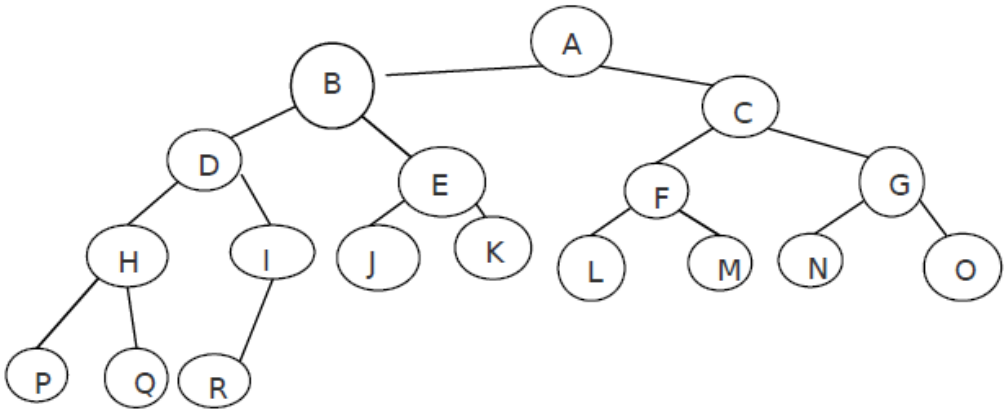
PROGRAMACIÓN 2 - Unidad 5: Árboles
Ejercicios

1. Considérese el árbol siguiente



   a. ¿Cuál es su profundidad?
   b. ¿Es un árbol binario? ¿Es completo o casi-completo? Justifique la respuesta.
   c. ¿Cuál es el **padre** del nodo R?
   d. ¿Cuáles son los **antecesores** del nodo J?
   e. Recorra el árbol anterior según los algoritmos de preorden, orden medio, postorden y en anchura. Para cada uno de los algoritmos dibuja el árbol e indica en cada nodo el orden de visita; y muestra al final los nombres de los nodos según el orden de visita.

a. 4
b. Es un árbol binario porque ningún nodo tiene más de dos hijos. No es completo porque el último nivel no está lleno. Es cuasi completo porque todos los demás niveles sí lo están. (Obsérvese que, sin embargo, NO es un árbol binario **de búsqueda**; por ejemplo, E > A está en el subárbol izquierdo de A)
c. I
d. A, B y E
e. Preorden: A B D H P Q I R E J K C F L M G N O
   En orden medio: P H Q D R I B J E K A L F M C N G O
   Postorden: P Q H R i D J K E B L M F N O G C A
   En anchura: A B C D E F G H I J K L M N O P Q R
   Los dibujos deberán corresponderse con estos órdenes, por ejemplo, en preorden: A-1, B-2, D-3, H-4, P-5, Q-6, I-7, …

2. Escriba el pseudocódigo de los algoritmos apropiados para determinar:
   a. La cantidad de nodos de un árbol binario.
   b. El número de hojas de un árbol binario.
   c. La suma del contenido de los nodos de un árbol binario de elementos de tipo entero.
   d. La profundidad de un árbol binario.

   Para los cuatro problemas, debe pensar cómo depende el valor a calcular del propio nodo y del valor correspondiente a los subárboles que lo tienen como raíz.

```
// a. La cantidad de nodos de un árbol binario.
int bt_numNodes (BinaryTree T)
  if bt_isEmpty(T) == TRUE: return 0
  return 1 + bt_numNodes(left(T)) + bt_numNodes(right(T))

// b. El número de hojas de un árbol binario.
int bt_numLeafs (BinaryTree T)
  if bt_isEmpty(T) == TRUE: return 0
  if isLeaf(T) == TRUE: return 1
  return bt_numLeafs(left(T)) + bt_numLeafs(right(T))

Bool isLeaf (BinaryTree T)
 if bt_isEmpty(left(T)) == TRUE AND bt_isEmpty(right(T)) == TRUE: return TRUE
 return FALSE

// c. La suma del contenido de los nodos de un árbol binario de elementos de tipo
entero.
int  bt_sumInfo (BinaryTree T)
  if bt_isEmpty(T) == TRUE: return 0
  return info(T) + bt_sumInfo(left(T)) + bt_sumInfo(right(T))

// d. La profundidad de un árbol binario.
int  bt_depth (BinaryTree T)
  if bt_isEmpty(T) == TRUE: return -1
  l = bt_depth(left(T))
  r = bt_depth(right(t))
  return 1 + maxValue(l,r)

int  maxValue (int a, int b)
  if a > b: return a
  return b
```

3. Escriba el código C para implementar los pseudocódigos de los algoritmos del problema anterior. Para ello, puedes asumir las siguientes estructuras de datos y definiciones:

```
typedef enum {FALSE = 0, TRUE=1} Bool;
typedef enum {ERROR = 0, OK=1} Status;

#define info(pnode) ((pnode)->info)
#define left(pnode) ((pnode)->left)
#define right(pnode) ((pnode)->right)
#define root(pt) ((pt)->root)

struct _BTNode {
    Element *info;
    struct _BTNode *left;
    struct _BTNode *right;
};
typedef struct _BTNode BTNode;

struct _BinaryTree {
   BTNode *root;    // un árbol es el puntero a su nodo raíz
};
typedef struct _BinaryTree BinaryTree;

// a. La cantidad de nodos de un árbol binario.
int  bt_numNodes(const BinaryTree *pt) {
  if (!pt) return 0;
  return bt_numNodes_rec(root(pt));
}
```

```
int  bt_numNodes_rec(const BTNode *pn) {
  if (pn == NULL) return 0;
  return 1 + bt_numNodes_rec(left(pn)) + bt_numNodes_rec(right(pn))
}

// b. El número de hojas de un árbol binario.
int bt_numLeafs(const BinaryTree *pt) {
  if (!pt) return 0;
  return bt_numLeafs_rec(root(pt));
}

int  bt_numLeafs_rec(const BTNode *pn) {
  if (pn == NULL) return 0;
  if (isLeaf(pn)) return 1;
  return bt_numLeafs(left(pn)) + bt_numLeafs(right(pn))

Bool isLeaf(const BTNode *pn) {
 if (left(pn) == NULL && right(pn) == NULL) return TRUE;
 return FALSE;
}

// c. La suma del contenido de los nodos de un árbol binario de elementos de tipo
entero.
int bt_sumInfo(const BinaryTree *pt) {
  if (!pt) return 0;
  return bt_sumInfo(root(pt));
}

int bt_sumInfo_rec(const BTNode *pn) {
  if (!pn) return 0;
  return *(int *)(info(pn)) + bt_sumInfo_rec(left(pn)) + bt_sumInfo(right(pn));
}

// d. La profundidad de un árbol binario.
#define max(a,b) ((a) > (b) ? (a) : (b))

int bt_depth (const BinaryTree *pt) {
  if (!pt || bt_isEmpty(pt)) return -1;
  return bt_depthRec(pt->root);
}

int bt_depthRec(const BTNode *pn) {
  int left_depth, right_depth;
  if (pn == NULL) return -1;
  left_depth = bt_depthRec(left(T));
  right_depth = bt_depthRec(right(t));
  return 1 + max(left_depth,right_depth);
}
```

4. Escriba el pseudocódigo de un algoritmo para determinar si un árbol binario es un árbol binario de búsqueda.

Solución **incorrecta**: comprobar solo que todos los nodos N satisfacen info(left(N)) < info(N) < info(right(N)). Esto no es suficiente, ver página 48 de la presentación. Por ejemplo, puede ocurrir que right(left(T)) > T incluso si left(T) < T < right(T) y left(left(T)) < left(T) < right(left(T)), es decir, incluso si los nodos T y left(T) cumplen individualmente la condición.

La idea del siguiente pseudocódigo es pasar como argumento el límite mínimo y máximo que se puede permitir en la parte del árbol en la que nos encontramos. Estos límites se inicializan a INT_MIN e INT_MAX para simplificar el pseudocódigo. La raíz tiene que estar entre estos límites, **cualquier** nodo de su subárbol izquierdo entre INT_MIN e info(root(T)) – 1, **cualquier** nodo de su subárbol derecho entre info(root(T)) + 1 e INT_MAX, y según se desciende estos límites se van estrechando.
(¿Sabrías decir para qué nodos del árbol se siguen usando INT_MIX o INT_MAX como límites según descendemos?)

```
Bool bt_isBST (BinaryTree T)
   return isBSTminmax(root(T), INT_MIN, INT_MAX)

Bool isBSTminmax (BinaryTree T, int min, int max)
   if bt_isEmpty(T) == TRUE: return TRUE

   // falso si viola la restricción min/max
   if info(T) < min OR info(T) > max:
      return FALSE

   if isBSTminmax(left(a), min, info(T) - 1)
        AND isBSTminmax(right(a), info(T) + 1, max):
      return TRUE
   return FALSE
```

El problema con la solución anterior es que solo sirve para ints. Y aunque limits.h define valores mínimos y máximos para algunos otros tipos de datos, no siempre podemos definir estos límites. Por ejemplo: ¿cuál sería la "cadena máxima" si no existe un límite superior para su longitud?

Una solución **genérica**, válida para cualquier tipo de elemento, se puede obtener usando como límites iniciales los valores más a la izquierda y más a la derecha del árbol (ver ejercicio 17). En un ABdB, estos serían respectivamente los valores mínimo y máximo del árbol, y por tanto pueden usarse como límites inferior y superior para el algoritmo. Si, por otro lado, los nodos más a la izquierda y más a la derecha no fueran el mínimo y/o el máximo, es fácil ver que el procedimiento devolvería correctamente que el árbol no es una ABdB.

```
Bool bt_isBST(const BSTree *tree) {
   BTNode *min, *max;

   if (tree_isEmpty(tree)) return TRUE;

   min = tree_minValueNode(tree->root); // nodo más a la izquierda
```

```
    max = tree_maxValueNode(tree->root); // nodo más a la derecha
    return bt_isBSTRec(tree->root, min->info, max->info, tree);
}


Bool bt_isBSTRec(const BTNode *node, const void *min_elem, const void *max_elem,
const BSTree *tree) {

    if (node == NULL)
       return TRUE; // el árbol vacío es un ABdB

    if (tree->cmp_element(node->info, min_elem) < 0 ||
        tree->cmp_element(node->info, max_elem) > 0)
        return FALSE; // viola restricción min/max

   // ambos subárboles han de ser ABdBs:
   return bt_isBSTRec(node->left, min_elem, node->info, tree) &&
           bt_isBSTRec(node->right, node->info, max_elem, tree);
}
```

5. Escriba el pseudocódigo de un algoritmo para determinar si un árbol binario es:
   a. Estrictamente binario (es decir, todo nodo no-terminal tiene **exactamente** dos hijos)
   b. Completo.

```
Bool bt_isFullBinaryTree (BinaryTree T)
  if bt_isEmpty(T) == TRUE: return TRUE
  // dos hijos vacíos
  if bt_isEmpty(left(T)) == TRUE AND bt_isEmpty(right(T)) == TRUE:
      return TRUE
  // un solo hijo vacío
  if bt_isEmpty(left(T)) == TRUE OR bt_isEmpty(right(a)) == TRUE:
      return FALSE
  // dos hijos, ninguno vacío: comprobar subárboles
  return bt_isFullBinaryTree(left(T)) AND bt_isFullBinaryTree(right(T))
```

```
Bool bt_isCompleteTree(BinaryTree T)
   if bt_isEmpty(T) == TRUE: return TRUE;
   num_nodes = bt_numNodes(T)
   depth = bt_depth(T)
   return (num_nodes == 2^(depth+1)-1)
```

6. Escriba el código C, sin control de errores, de una función para determinar si un árbol binario es estrictamente cuasi-completo (es decir, cuasi-completo, y además todos los nodos del último nivel ocurren a la izquierda). Puede utilizar los tipos y estructuras de datos y macros en C del ejercicio 3.
Sugerencia: Utilice una variación de la búsqueda en anchura.

Recorremos el árbol en anchura (por niveles), usando una cola. Usando este recorrido, el árbol será estrictamente cuasi-completo solo si no hay ningún nodo nulo después del primer nodo nulo.

Usamos un flag **null_found** que haremos verdadero en el momento que veamos el primer nodo nulo. Al sacar un elemento de la cola, si su hijo izquierdo no es nulo y ya hemos visto algún nodo nulo, devolvemos FALSE; y si no, insertamos este hijo en la cola. De la misma forma, si su hijo derecho no es nulo y ya hemos visto algún nodo nulo, devolvemos FALSE; y si no insertamos este hijo en la cola. Si al completar el bucle, no hemos visto ningún nodo nulo después de uno nulo, el árbol es estrictamente cuasi-completo.

(Nota: la terminología puede variar, en algunos sitios se dice que un árbol binario es “completo” para referirse a lo que aquí llamamos “estrictamente cuasi completo”)

```
Bool bt_isAlmostComplete (const BinaryTree *pt) {
    Queue *Q;
    BTNode *node;
    Bool null_found = FALSE;

    // un árbol vacío es completo
    if (root(pt)) == NULL) return TRUE;

    pc = queue_init(); // cola de nodos

    queue_insert(Q, root(pt));
    while (queue_isEmpty(Q) == FALSE) {
         node = queue_extract(Q);

        if (left(node)) {
            if (null_found) {
                queue_free(Q);
                return FALSE;
            }
            queue_insert(Q, left(node));
        } else {   null_found = TRUE;  }

        if (right(node)) {
            if (null_found) {
                queue_free(Q);
                return FALSE;
            }
            queue_insert(Q, right(node));
        } else {   null_found = TRUE;  }

      }
```

```
    // si hemos llegado hasta aquí, no hemos encontrado
    // ningún nodo nulo después de un nodo nulo
    queue_free(q);
    return TRUE;
}
```

7. a) Escribir una función en C que reciba un árbol binario (pero no necesariamente un ABdD) y un elemento y devuelva el nivel del árbol en que se encuentra, o -1 si no se encuentra en el árbol. Puedes utilizar las estructuras de datos y macros del ejercicio 3.
   b) Si el árbol tuviese nodos repetidos debería devolver el nodo más profundo.

Suponed que se dispone de una función que permite comparar dos elementos con el prototipo:

```
int element_compare (Element *ele1, Element *ele2)
```

tal como se ha descrito en la presentación del tema para ABdDs. (Es decir, devuelve 0 si ele1 == ele2, un número negativo si ele1 < ele2, y un número positivo si ele1 > ele2).

```
#define max(a,b) ((a) > (b) ? (a) : (b))

int bt_findNodeLevel(const Binary Tree *T, Element *ele) {
    if (bt_isEmpty(T)) return -1;
    return bt_findNodeLevelRec(root(T), ele, 0); // 0 = profundidad de la raíz
}

int bt_findNodeLevelRec(const BTNode *pn, const Element *ele, int depth) {
    int found_left, found_right, found = -1;
    if (pn == NULL) return -1;              //elemento no encontrado
    if (element_compare(info(pn),ele) == 0) //elemento encontrado
       found = depth;
   // buscar recursivamente en AMBOS subárboles
   found_left = bt_findNodeLevelRec(left(pn),ele,depth+1);
   found_right = bt_findNodeLevelRec(right(pn),ele,depth+1);
   return max(found,max(found_left,found_right));
}
```

8. Repetir el problema anterior suponiendo que el árbol es un árbol binario de búsqueda (ABdB). ¿El algoritmo del ejercicio 7 sería eficiente para un ABdB? ¿Por qué?

Asumimos que no hay duplicados en un ABdB, así que basta con pasar un parámetro adicional con la profundidad a la que nos encontramos en el procedimiento estándar de búsqueda de un ABdB.

```
int bst_findNode(const Binary Tree *T, const Element *ele) {
    if (!T || !ele || bt_isEmpty(T)) return -1;
   return bst_findNodeRec(root(T), ele, 0); // 0 = profundidad de la raíz
}

int bst_findNodeRec(const BTNode *pn, const Element *ele, int depth) {
   int cmp;
   if (pn == NULL) return -1;           //elemento no encontrado
   cmp = element_compare(info(pn),ele);
   if (cmp == 0) return depth;          //elemento encontrado
```

```
   // buscar recursivamente en el subárbol que corresponda
   if (cmp < 0) return bst_findNodeRec(left(pn),ele,depth+1);
   return bst_findNodeRec(right(pn),ele,depth+1); // cmp > 0
}
```

La complejidad de este código es proporcional a la profundidad del ABdB. El algoritmo del ejercicio anterior, por otro lado, requiere considerar, en el caso peor, todos los nodos del árbol, y es por tanto mucho menos eficiente.

9. Proporcione el código C de una función de prototipo

`Status tree_printFromLevel (BinaryTree *pa, int level, FILE *pf)`

que imprima en un flujo de salida pf el campo info de todos los nodos del árbol binario completo `pa` que estén en un nivel menor o igual al nivel `level` que se le pasa como argumento. Si la profundidad del árbol fuese inferior al nivel, la función devolverá ERROR y no imprimirá nada. Puedes utilizar las estructuras de datos y macros en C del ejercicio 3.

Nota: Probablemente tree_printToLevel habría sido un nombre mejor para esta función, puesto que se pide imprimir los niveles por encima de level.

```
Status tree_printFromLevel (const BinaryTree *pa, int max_level, FILE *pf) {
   if (pa == NULL || pf == NULL || max_level < 0) return ERROR;

   return tree_printFromLevelRec(root(pa), max_level, 0, pf); //0: profundidad de la
raíz
}

Status tree_printFromLevelRec (const BT_Node *pn, int max_level, int
current_depth, FILE *pf) {
   Status l, r;
   // dos condiciones de parada de la recursión
   // 1. he llegado al final de un sub-árbol
   if (pn == NULL) return ERROR;
   // 2. he llegado a la profundidad deseada
   if (current_depth == max_level) {
      element_print(pf, info(pn));
      return OK;
   }
   // Caso general de la recursión
   // Nótese que estas llamadas solo se realizan con current_depth < max_level
   l = tree_printFromLevelRec (left(pn), max_level, current_depth+1, pf);
   r = tree_printFromLevelRec (right(pn), max_level, current_depth+1, pf);

   // el nodo está a una profundidad menor que el nivel (parámetro de entrada)
   // pero el árbol tiene una profundidad igual o superior
   if (l==OK || r==OK)  {
      element_print(pf, info(pn));
      return OK;
   }
}
```

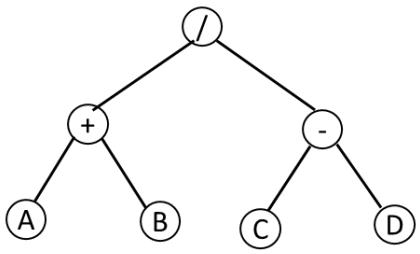
10. Escriba el código en C para recorrer un árbol binario en orden previo y en orden posterior. Puedes utilizar las estructuras de datos y macros en C del ejercicio 3, y el pseudocódigo de la presentación del tema.

```c
/*Recorre un árbol en orden previo.*/
void tree_preOrder(FILE *f, const BinaryTree *tree) {
  if (!f || !tree) return;
  tree_preOrderRec(f, tree->root);
}

void tree_preOrderRec(FILE *f, BTNode *pn) {
  if (!pn) return;
  element_print(f, pn->info);
  tree_preOrderRec(f, pn->left);
  tree_preOrderRec(f, pn->right);
}

/*Recorre un árbol en orden posterior.*/
void tree_postOrder(FILE *f, const BinaryTree *tree) {
  if (!f || !tree) return;
  tree_postOrderRec(f, tree->root);
}

void tree_postOrderRec(FILE *f, BTNode *pn) {
  if (!pn) return;
  tree_postOrderRec(f, pn->left);
  tree_postOrderRec(f, pn->right);
  element_print(f, pn->info);
}
```

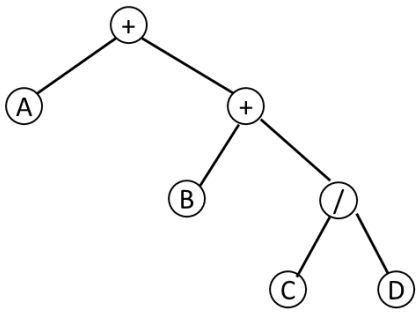
11. Dibujar los árboles de expresión que representan las siguientes expresiones

(A + B) / (C - D);
A + B + C / D;
A – (B - (C - D) / (E + F));
(A + B) * ((C + D) / (E + F));

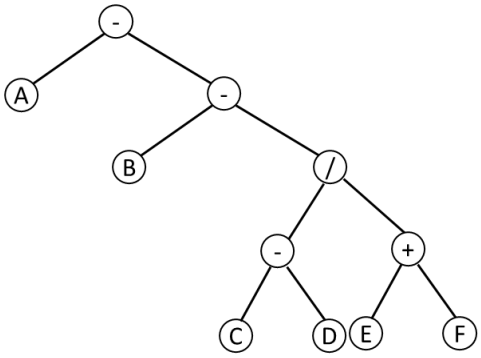
1



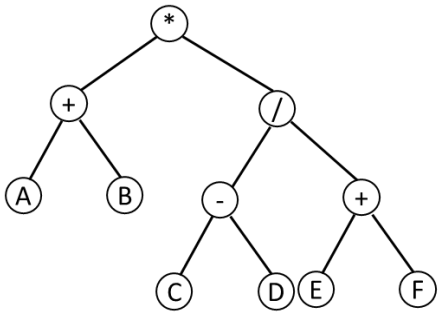
2



3



4



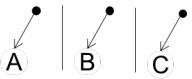
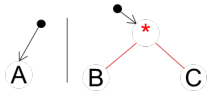
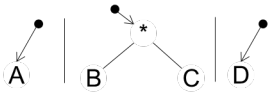
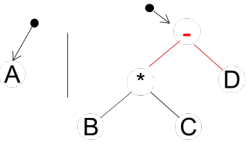
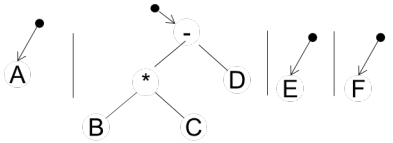
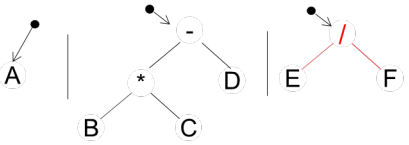
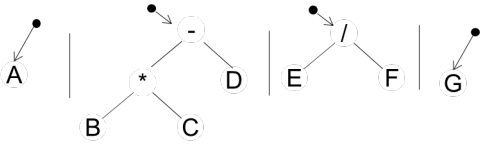
12. Construir algorítmicamente el árbol de expresión a partir de las siguientes expresiones sufijo:

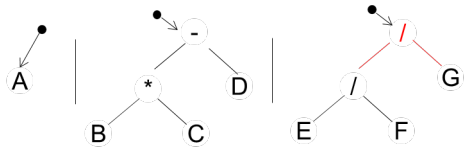
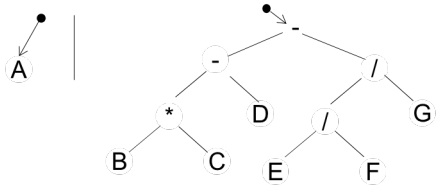
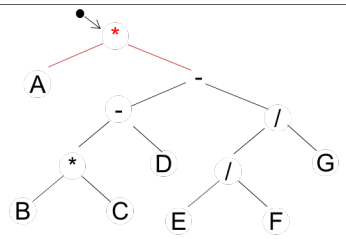
A B C * D - E F / G / - * ;

A B + C D - / E F * G - - ;

A B C D - - E F G + + * * ;

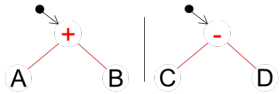
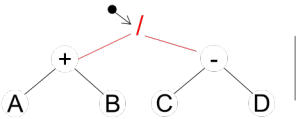
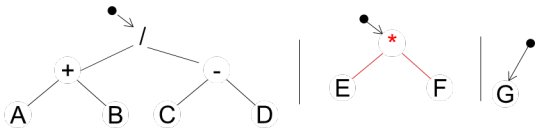
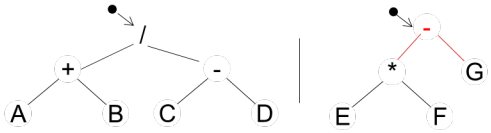
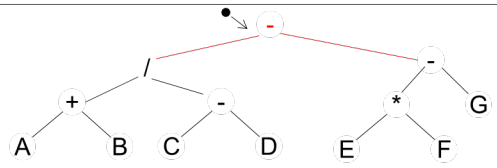
1. A B C * D - E F / G / - * ;

Símbolo	Pila (tope por la derecha)
A, B, C	
*	
D	
-	
E, F	
/	
G	

/	
-	
*	
;	Pop. Como la pila queda vacía, lo extraído es el árbol de expresión.

Cadena terminada con pila vacía => árbol de expresión completo.

2. A B + C D - / E F * G - - ;

Símbolo	Pila (tope por la derecha)
A, B, +, C, D, -	
/	
E, F, *, G	
-	
-	
;	Pop. Como la pila queda vacía, lo extraído es el árbol de expresión.

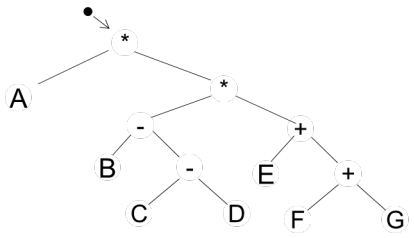
3. A B C D - - E F G + + * * ;

Sin hacer el dibujo completo, los estados de la pila que se van generando son:

Símbolos leídos	Estado de la pila
A, B, C, D	A, B, C, D
-	A, B, (C – D)
-	A, B – (C – D)
E, F, G	A, B – (C – D), E, F, G
+	A, B – (C – D), E, (F + G)
+	A, B – (C – D), E + (F + G)
*	A, (B – (C – D)) * (E + (F + G))
*	A * ((B – (C – D)) * (E + (F + G)))
;	Pop, pila vacía, el árbol extraído es la expresión final

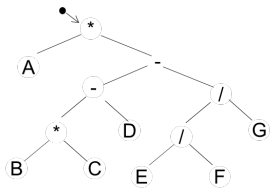
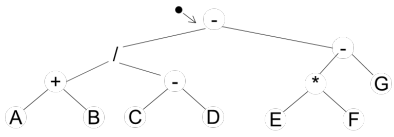
En la última expresión se podrían eliminar paréntesis innecesarios: A * (B – C – D) * (E + F + G). Pero los paréntesis realmente reflejan la forma en que la expresión sería evaluada.

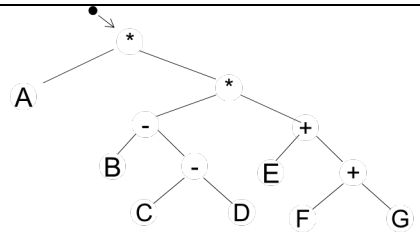
El árbol final es:



13. Utilizando los árboles obtenidos en el problema anterior, obtenga las expresiones prefijo e infijo correspondientes a las expresiones sufijo.

Recorremos los árboles en preorden y orden medio, respectivamente. Para infijo, abrimos paréntesis al visitar un operador no raíz, y lo cerramos cuando terminamos su subárbol. Cada profundidad de un operador la ilustramos con un color en las expresiones infijas: 0, 1, 2, 3.

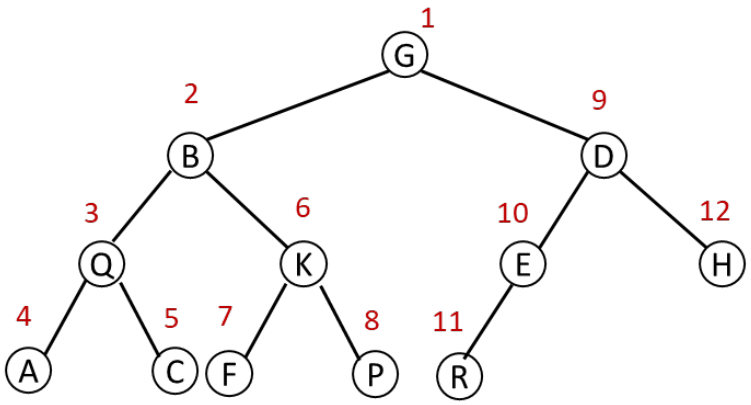
Árbol de expresión	Prefijo	Infijo
	* A – – * B C D / / E F G	A * [((B*C) – D) – ((E / F) / G)]
	– / + A B – C D – * E F G	[(A + B) / (C – D)] – [(E * F) – G]

	* A * – B – C D + E + F G	A * [(B – (C – D)) * (E + (F + G))] Equivalentemente A * (B – C - D) * (E + F + G).

14. Supóngase que al recorrer un árbol binario en preorden se obtiene la siguiente lista

    G B Q A C K F P D E R H

    Dibuje una posible estructura de nodos del árbol. ¿Hay otras estructuras posibles?



Hay muchas otras estructuras posibles. Por ejemplo, si R es hijo derecho de E en vez de su hijo izquierdo; o si H es hijo derecho de E. O bien C podría ser el hijo derecho de B, y tener a K, F, P en su subárbol.

Nótese que cualquiera de estos es un árbol binario, pero no un árbol binario de búsqueda: por ejemplo, Q > B y K > G.

15. Construir el árbol binario de búsqueda a partir de cada una de las listas siguientes:

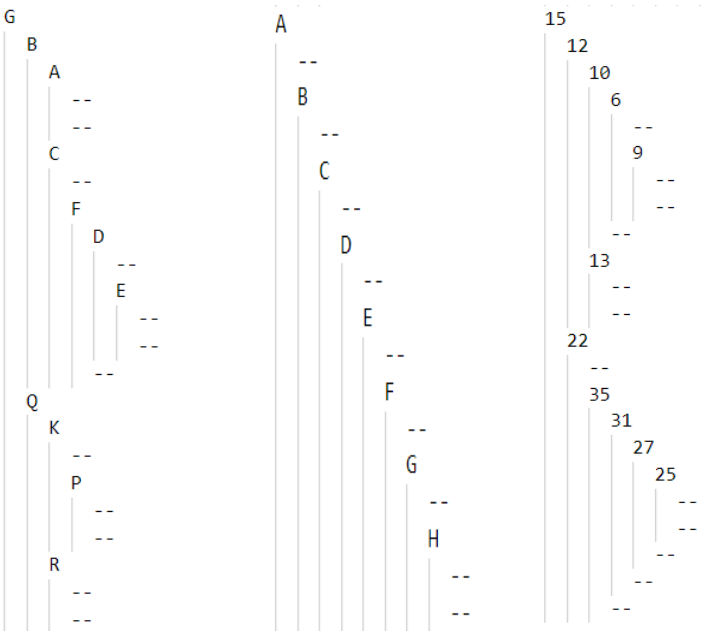
    G B Q A C K F P D E R

    A B C D E F G H

    15 22 12 35 31 13 10 27 6 9 25

En las siguientes imágenes, en un árbol cada nodo se imprime con una sangría proporcional a su profundidad, y los hermanos están unidos por líneas verticales; además, los nodos nulos se imprimen como '—'. Por ejemplo, en el primer árbol G es la raíz, y tiene dos hijos B y Q. A su vez, B tiene A y C como hijos; A no tiene hijos, y C solo tiene hijo derecho, F. Los hijos de Q son K y R.

Obsérvese que el segundo árbol solo tiene hijos derechos, y es por tanto básicamente equivalente a una lista enlazada. Esto ocurrirá siempre que se crea un ABdB a partir de un conjunto de elementos ordenados.



16. En un árbol binario de búsqueda se define el predecesor de un dato D como el mayor dato D' del árbol tal que D' < D y su sucesor como el menor dato del árbol tal que D' > D. Sobre los arboles del problema anterior, encontrar el predecesor y el sucesor de cada uno de los elementos.

    Podemos encontrar los predecesores y sucesores de todos los nodos simplemente imprimiendo los árboles en orden medio: el predecesor de un elemento en el árbol es el elemento que le precede en el orden medio, y el sucesor el elemento que le sigue. El resultado es el siguiente:
    A B C D E F G K P Q R
    A B C D E F G H
    6 9 10 12 13 15 22 25 27 31 35

    Así, por ejemplo, en el primer árbol, el predecesor de C es B y su sucesor es D. Más en general:
    - El sucesor de un nodo N es el nodo más a la izquierda de su subárbol derecho, si este existe, puesto que este será el mínimo entre los nodos mayores que N.
    - Si no tiene hijo derecho, se asciende desde N hasta encontrar el padre de un hijo izquierdo.
    - El predecesor de un nodo N es el nodo más a la derecha de su subárbol izquierdo, si este existe, puesto que este será el máximo entre los nodos menores que N.
    - Si no tiene hijo izquierdo, se asciende desde N hasta encontrar el padre de un hijo derecho.
- Ver ejercicio 18 para más detalles sobre el sucesor.

17. Escribir el código de C de un algoritmo que devuelva un puntero al elemento mínimo de un árbol binario de búsqueda. Dar un procedimiento recursivo y otro no recursivo.

```
Element *minElement (BinaryTree *pa)
```

    El elemento mínimo es el que está más a la izquierda en el árbol.

Versión no recursiva

```
Element *minElement(const BinaryTree *pa) {
   if (!pa || bt_isEmpty(pa)) return NULL;
   for (pn = root(pa); left(pn) != NULL; pn = left(pn)); // bajar siempre a la izquierda
   return info(pn);
}
```

Versión recursiva

```
Element *minElement(const BinaryTree *pa){
  if (pa==NULL) return NULL;
  return minElementRec (pa->root);
}

Elemento *minElementRec (const BTNode  *pm) {
 if (pn==NULL) return NULL;
 if (left(pn)==NULL) return info(pn);

 return minElementRec (left(pn));
}
```

18. Dar el pseudocódigo y el código C de una función que reciba un nodo de un árbol binario de búsqueda y devuelva el campo info de su nodo sucesor. Para ello, considere que la estructura de datos utilizada para representar los nodos del árbol contiene un campo adicional `parent` que permite acceder al **padre** de un nodo dado.

Código:

```
Element *successor(const BTNode *pn)
  if (!pn) return NULL;
  //si el nodo tiene subárbol derecho devuelve el mínimo del subárbol derecho
  if (right(pn))
    return minElement(right(pn));

  // si no tiene subárbol derecho, asciende. El sucesor es el padre del primer
  // hijo izquierdo que se encuentra al ascender
  pt = parent(pn);
  // mientras pn sea el hijo derecho de su padre pt, ascender
  // (bucle para si padre nulo, o pn es hijo izquierdo de pt)

  while (pt && right(pt) == pn) {
     pn = pt;
     pt = parent(pn);
  }
  if (!pt) { // padre nulo significa que pn es ahora la raíz
  // hemos llegado hasta aquí sin ascender nunca de un hijo izquierdo
 // a su padre, por tanto el nodo inicial pn es el más a la derecha (máximo)
  // del árbol
     return NULL;
  }
  return info(pt);
}
```

19. Dos árboles binarios son topológicamente similares si ambos están vacíos o, si ambos no están vacíos, sus subárboles izquierdos son similares y sus subárboles derechos son similares. De un ejemplo de árboles similares de profundidad mayor o igual a dos.
    a. Escriba el pseudocódigo de un algoritmo para determinar si dos árboles son similares.
    b. Utilizando las estructuras de datos y macros en C del ejercicio 3 proporcione el código C de una función con el prototipo siguiente que implemente el algoritmo anterior:
    `Bool similarTrees (BinaryTree *T1, BinaryTree *T2)`

    Código:

```
Bool similarTrees(const BinaryTree  *pt1, const BinaryTree *pt2) {
    if (pt1==NULL || pt2==NULL) return FALSE;
    return similarTreesRec(root(pt1), root(pt2));
}

Bool similarTreesRec (const BTNode  *pn1, const BTNode *pn2) {
   Bool sim_left, sim_right;

   // Casos base de la recursión
   if (pn1==NULL && pn2==NULL) return TRUE;  // ambos nulos: similares
   if (pn1==NULL || pn2==NULL) return FALSE; // solo uno nulo: no similares

   // Caso general de la recursion: ninguno nulo
   sim_left = similarTreesRec(left(pn1), left(pn2)); // similares subárboles
   izqdos?
   sim_right = similarTreesRec(right(pn1), right(pn2)); // similares subárboles
   derechos?
   return (sim_left && sim_right);
}
```

20. Escriba el pseudocódigo de un algoritmo que reciba un árbol binario y cree un nuevo árbol binario copia del anterior. Dar, a continuación, el código C del algoritmo propuesto. Suponga, para ello, los tipos y estructuras de datos definidas en el ejercicio 3.

    Pseudocódigo:

```
BinaryTree tree_copy(BinaryTree T)
   new_tree = bt_tree_new()
      root(new_tree) = tree_copyRec(root(T))

BTNode tree_copy_rec(BTNode N)
   if N == NULL: return NULL
      new_node = bt_node_new()
   info(new_node) = copy_element(info(T))
      left(new_node) = tree_copyRec(left(T))
   return new_node
```

Código:

```
BinaryTree *tree_copy(const BinaryTree *tree) {
     BinaryTree *new_tree = bt_new();

     if (!new_tree) return NULL;
     new_tree->root = tree_copyRec(tree->root);
     return new_tree;
   }

   BTNode *tree_copyRec(const BTNode *node) {
     BTNode *new_node = NULL;

     if (!node) return NULL;
     new_node = bt_node_new();
     if (!new_node) return NULL;

     new_node->info = element_copy(node->info);
     if (!new_node->info) {
       bt_node_free(new_node);
       return NULL;
     }

     new_node->left = tree_copyRec(node->left, tree);
     new_node->right = tree_copyRec(node->right, tree);
     return new_node;
   }
```

21. Escriba el pseudocódigo de un algoritmo que acepte un árbol binario y lo modifique de forma que sea la imagen refleja del original (es decir, un árbol en el que todos los subárboles derechos del árbol original son ahora subárboles izquierdos y viceversa). Dar, a continuación, el código C del algoritmo propuesto.

Pseudocódigo

```
   void bt_mirrorTree (BinaryTree T)
      if bt_isEmpty(T) == FALSE
           aux = left(T)
           left (T) = right(T)
           right(T) = aux

           bt_Mirror(left(T))
           bt_Mirror(right(T))
```

Código

```
   void bt_mirrorTree (BinaryTree *pt) {
       if (pt !=NULL)
          bt_mirrorTreeRec(root(pt));
   }
```

```
void bt_mirrorTreeRec (BTNode *pn) {
    BTNode *temp = NULL;

   if (pn == NULL) return;

   // intercambiar hijos
   temp = left(pn);
   left(pn) = right(pn);
   right(pn) = temp;

   // recursivamente computar el reflejo de los subárboles
   bt_mirrorTreeRec(left(pn));
   bt_mirrorTreeRec(right(pn));
}
```

22. Escriba el pseudocódigo y el código C de un algoritmo recursivo que devuelva como cadena de caracteres la representación prefijo de una expresión aritmética almacenada en un árbol de expresión.

La solución es una modificación del recorrido preorden.

Pseudocódigo

```
String prefix_exp(BinaryTree eT)
   if bt_isEmpty(eT): return
   añadir info(eT) a la cadena
   prefix_exp(left(T))
   prefix_exp(right(T))
   return cadena
```

Código

```
#define BUFFER_SIZE 512; // ¿cómo se podría hacer con memoria dinámica?

char *prefix_exp(const BinaryTree *eT) {
   char *str = NULL;

   if (!eT || bt_isEmpty(eT )) return NULL;

   str = malloc(BUFFER_SIZE * sizeof(char));
   if (!str) return NULL;
   buffer[0] = '\0'; // inicializar buffer a la cadena vacía

   return prefix_expRec(root(eT), str);
}

char *prefix_expRec(const BTNode *pn, char * str) {
   if (!pn || !pn->info) return str;

   // concatenar info(pn) al final de la cadena str
   // CdE: debería comprobarse que hay espacio suficiente !!
   // coste de concatenar es proporcional a la longitud de las cadenas,
   // podría optimizarse pasando un argumento adicional con la longitud de str
   strcat(str, info(pn))
   prefix_expRec(left(pn), str);
   prefix_expRec(right(pn), str);
   return str;
}
```

23. Se desea construir un algoritmo que, dado un árbol binario de búsqueda (ABdB), un elemento y un flujo de salida, busque dicho elemento en el árbol y si se encuentra imprima en el flujo de salida el camino desde el nodo raíz del árbol hasta el elemento. Es importante que sólo se imprima el camino si se encuentra el nodo en el árbol.

    Puede usar las funciones de la interfaz de cualquiera de los TADs vistos durante el curso, así como las estructuras y definiciones del ejercicio 3 y las primitivas que necesite para gestionar el contenido del campo info (element_print, element_free, etc.)

    El algoritmo sólo se considerará válido si tiene la menor complejidad posible.

    a) Proporcione el código C con control de errores de la función de prototipo *int tree_printPath (FILE *pf, AB *T, Element *ele)* que implementa el algoritmo anterior.

    b) ¿Cuál es la complejidad de la función desarrollada? Responda justificadamente

    a) Se puede usar una pila, cola o lista para ir acumulando los nodos visitados en la búsqueda del elemento. Escogemos una cola en esta solución porque nos permite imprimir el camino desde el origen, la raíz, al destino, con operaciones eficientes. Ejercicio: ¿cómo cambiaría la solución si se usara una pila o una lista? ¿Qué ventajas e inconvenientes tendrían?

    Importante: Esta solución funciona para un ABdB, porque tenemos garantizado que todos los nodos visitados están en el único camino desde la raíz hasta el elemento. Para árboles que no sean ABdBs, o grafos, sería necesario extraer de la cola (o pila o lista) los nodos visitados que no estén en el camino.

```
int tree_printPath (FILE *pf, const BinaryTree *tree, const Element *ele) {
    Queue *q;
    Element *ele;
    Status flag = OK;
    int i = 0;

    if (!pf==NULL || !tree || bt_isEmpty(tree)) return -1;

    q = queue_init();
    if (q==NULL) return -1;

    flag = tree_printPathRec(root(tree), q, ele);

    if (flag == ERROR) {
        queue_free(q);
        return -1;
    }

    // imprime el camino
    while (queue_isEmpty(q) == FALSE) {
        ele = queue_extract(q);
        i = i + element_print(pf, ele);
        element_free(ele);
    }
    queue_free(q);
    return i;
}
```

```
Status tree_printPathRec(const BTNode *pn, Queue *q, const Element *ele) {
    int cmp;
    Status flag:

    /* caso base 1: árbol vacío, elemento no encontrado */
    if (pn==NULL) return ERROR;

    flag = queue_insert(q, info(pn));
    if (flag == ERROR) return flag;

    cmp = element_cmp (ele, info(pn));

    /* caso base 2: elemento encontrado */
    if (cmp==0){
        return OK;
    }

    /* caso general: */
    if (cmp < 0)
        flag = tree_printPath_rec (left(pn), pc, ele);
    else
        flag = tree_printPath_rec (right(pn), pc, ele);

    return flag;
}
```

b) El coste es el mismo que el de realizar una búsqueda en un AbDB, es decir, proporcional a la profundidad del árbol, y por tanto entre O(log(n)) para árboles equilibrados y O(n) en el caso peor.

.