CAPÍTULO IV: ESTRATEGIAS DE DISEÑO DE ALGORITMOS

4.1 Recursión

Concepto: procedimiento recursivo

Un procedimiento que se llama a sí mismo, directa o indirectamente, se dice que es **recursivo.**

El uso de la recursión, permite a menudo, una descripción más clara y concisa del algoritmo, de lo que sería posible sin utilizar la recursión.

Los algoritmos recursivos son principalmente apropiados cuando el problema a ser resuelto o la función a ser calculada o la estructura de datos a ser procesada están ya definidos en términos recursivos.

Ejemplos de procedimientos recursivos:

- Recorrido de árboles.
- Exploración y evaluación de expresiones aritméticas.
- Juegos en los que un movimiento se puede dar en función de otro de menor nivel o complicación.

El programa recursivo puede ser escrito casi directamente de la definición del problema, en tanto que la forma no recursiva requiere cierto grado de habilidad en su programación. No solamente la tarea de escribir el programa es más sencilla, sino que también el programa resulta más comprensible para otras personas.

Si un procedimiento P contiene una referencia explícita a sí mismo, se dice **directamente recursivo.**

Procedure P (lista de parámetros)
BEGIN

P(lista de argumentos)

END;

Si un procedimiento P contiene una referencia a otro procedimiento Q, el cual contiene una referencia a P se dice que P es **inderectamente recursivo**.

Procedure P(lista de parámetros)
BEGIN

Q (lista de argumentos)

END;

Procedure Q(lista de parámetros)
BEGIN

P(lista de argumentos)

END;

El uso de la recursión puede, por lo tanto, no ser inmediatamente visible, al examinar el texto del programa.

Recursión e Iteración

Existen muchas funciones matemáticas que pueden ser definidas recursivamente.

Por ejemplo, para valores positivos de n, se tiene:

a) $$n! = \begin{cases} 1 & \text{si } n = 0 \\ n*(n-1)! & \text{si } n > 0 \end{cases}$$

b) $$X^n = \begin{cases} 1 & \text{si } n = 0 \\ X*X^{n-1} & \text{si } n > 0 \end{cases}$$

c) $$P_n(x) = \begin{cases} 1 & \text{si } n = 0 \\ X & \text{si } n = 1 \\ \dfrac{(2n-1)\,P_{n-1}(x) - (n-1)\,P_{n-2}(x)}{n} & \text{si } n > 1 \end{cases}$$

Polinomios de Legendre

Ejemplos:

a) Cálculo del N!

Solución recursiva:

```
function fact (n)
begin
        if (n=1) then
                fact ← 1
        else
           fact ← n * fact (n-1)
end
```

Ejemplo: valor ← fact (4)

1) 4! = 4 * 3!

2) 3! = 3 x 2!

3) 2!=2 * 1!

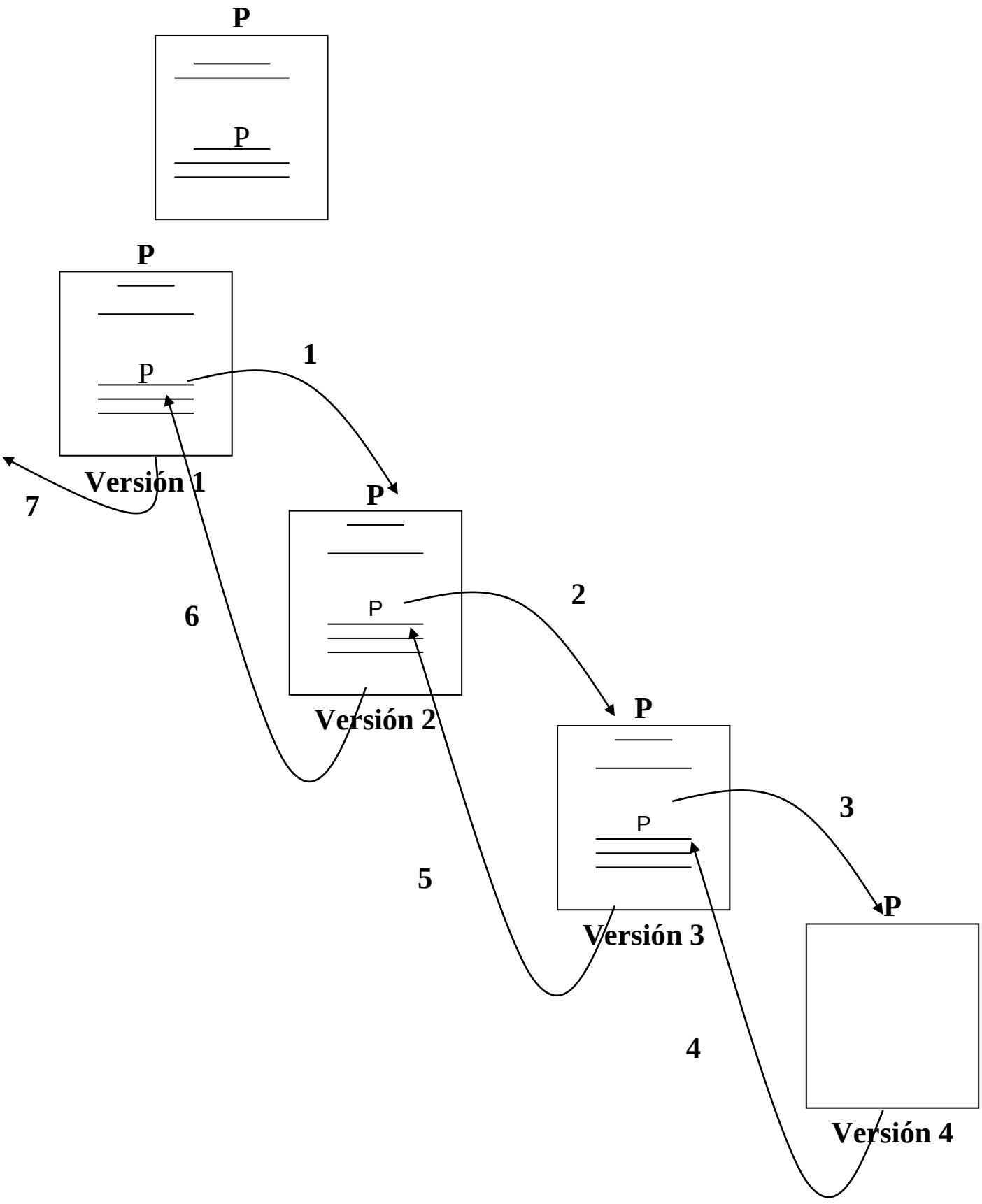
4) 1! =1

5) 2! = 2 * 1 = 2

6) 3! = 3 * 2 = 6

7) 4! = 4 * 6 = 24

Distintos llamados del procedimiento recursivo P



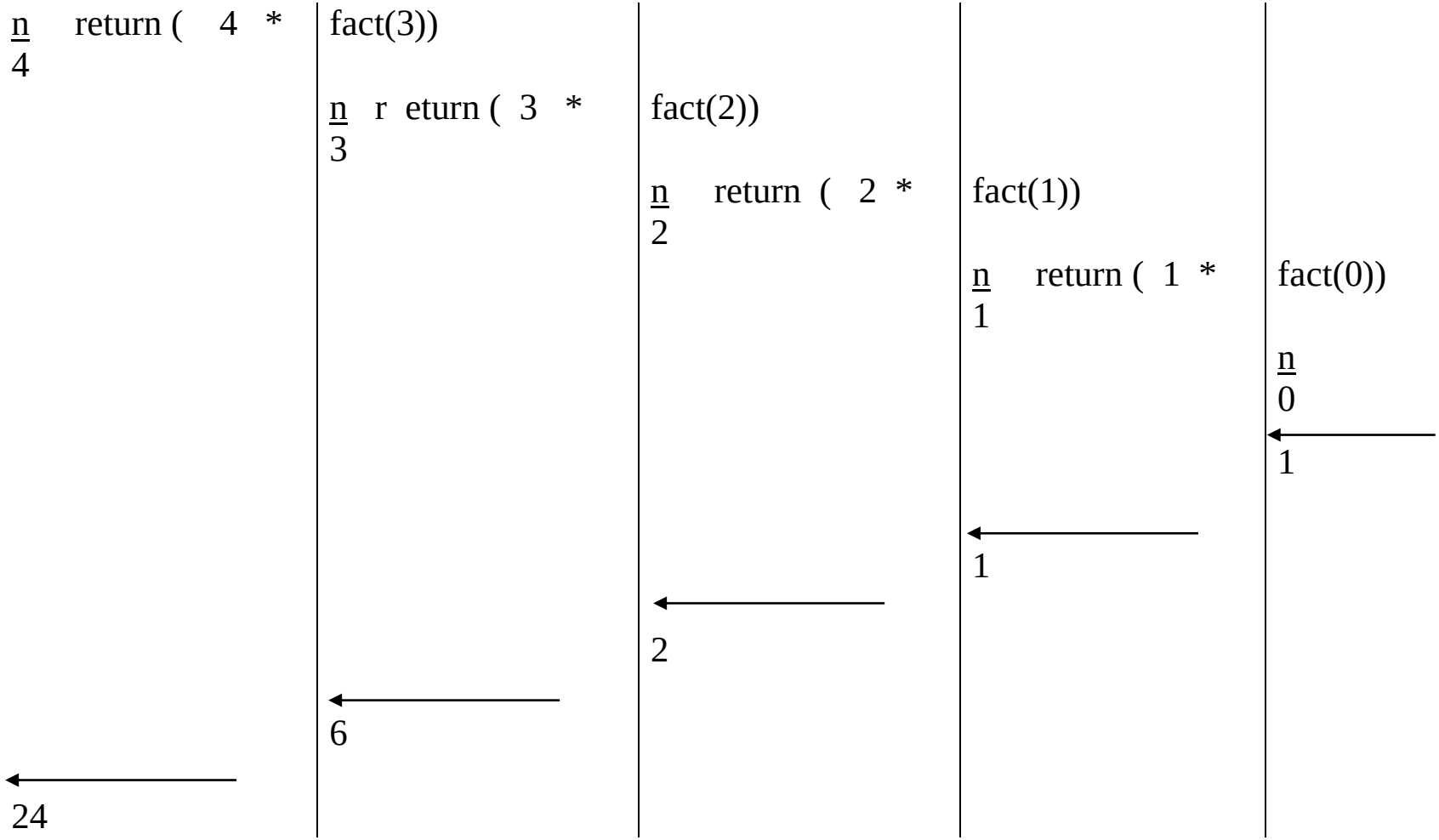
En Java:

```
public class FactApp {
    public static int factorial (int n) {
        if ( n == 0){
            return 1;
        }
        else {
            return (n * FactApp.factorial (n-1));
        }
    }
    public  static void main(String [] Args) {
        int fac = FactApp.factorial(4);
        StdOut.println("Factorial: " + fac);
    }
}
```

Solución Iterativa:

```
function fact (n)
begin
    K ← 1
    fact ← 1
    while (K ≤ n )do
    begin
        fact ← fact * K
        K ← K + 1
    end
end
```

Ruteo algoritmo recursivo



b) Serie de Fibonacci

$Fi = F_{i-1} + F_{i-2}$ $\quad i > 1$

$F_1 = 1$

$F_0 = 0$

Solución recursiva:

```
function fib (n)
begin
        if (n = 0) then
                fib ← 0
        else
                if (n = 1) then
                        fib ← 1
                else
                        fib ← fib (n-1) + fib (n-2)
end
```

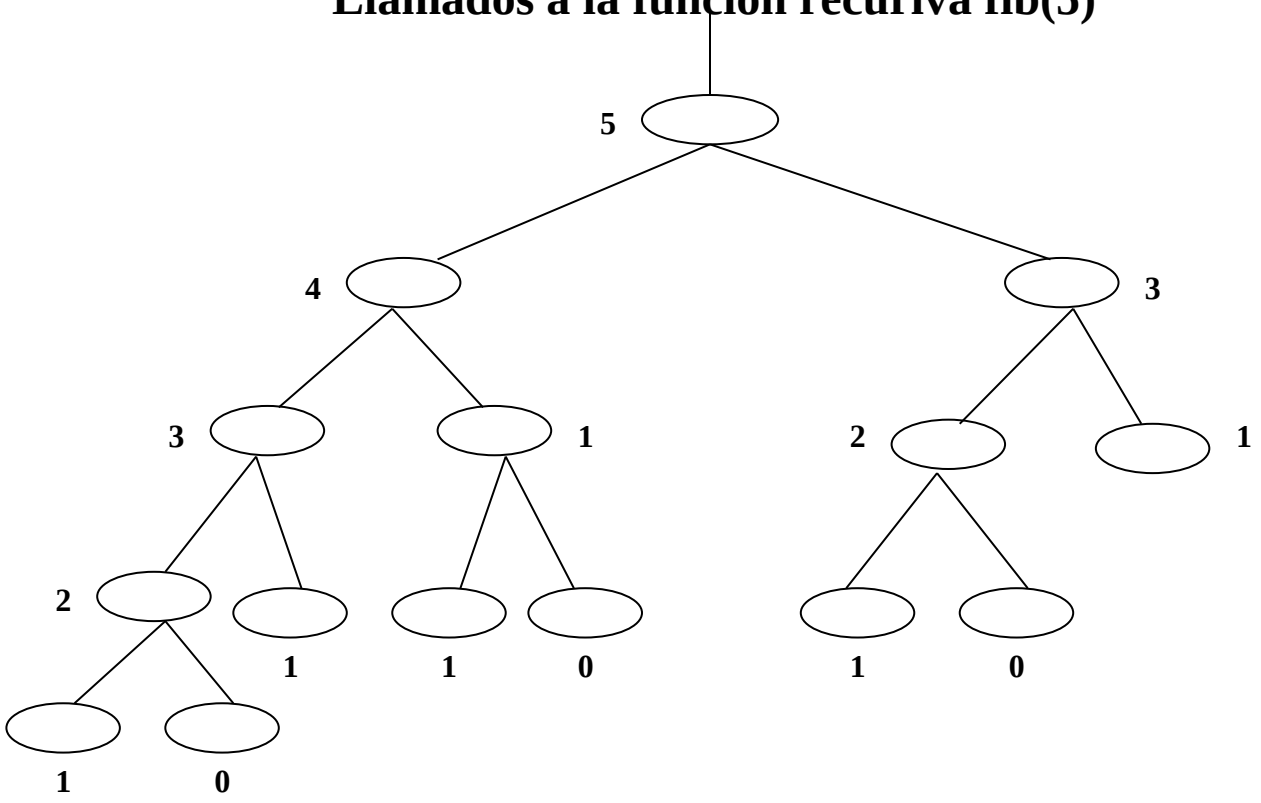
Se quiere el F_n

Ejercicio: Traducir el pseudocódigo anterior a Java.

Caso de: K ← fib (5)

- El número de llamadas al procedimiento, crece en forma exponencial.

Llamados a la función recuriva fib(5)



- En este caso, la solución recursiva es **claramente impráctica**, ya que el algoritmo recalcula un mismo valor muchas veces.

Solución Iterativa:

```
function fib (n) {
    if (n = 0) {
        fib ← 0
    }
    else
        if (n=1) {
            fib ← 1
        }
        else {
            penult ← 0
            ult ← 1
            i ← 2
            while (i ≤ n) {
                fib ← ult + penult
                penult ← ult
                ult ← fib
                i ← i +1
            }
        }
    }
}
```

c) Calculo de los Polinomios de Legendre.

Solución recursiva:

```
function P(n, x)
begin
    if (n = 0) then
        P ← 1
    else
        if (n = 1) then
            P ← x
        else
            P ← ((2*n - 1) *P( n -1,x) – (n - 1) *P( n - 2,x)) / n
end
```

Ejercicio: Traducir el pseudocódigo anterior a Java.

Solución Iterativa:

```
funtion pol (n, x)
 begin
    if (n = 0) then
        pol ← 1
    else
        if (n = 1) then
            pol ← x
        else
        begin
            anterior ← 1
            actual   ← x
            k        ← 2
            while (k ≤ n) do
                sgte ← ((2 * k – 1) * actual - (k-1) * anterior ) /k
                anterior ← actual
                actual   ← sgte
                k ← k+1
            endwhile
            pol ← sgte
        end
 end
```

Ejercicios

Búsqueda Binaria

- Se tiene el arreglo ordenado A(i), i = 0, N-1
- Buscar X en el arreglo A.
- Si X se encuentra en el arreglo, entregar su ubicación.
- Si X no se encuentra en el arreglo, retornar un -1.

0	1	2	3	4	5
2	6	13	27	40	47

A

40

Buscar por X en el arreglo ordenado A[0...N-1]:

- Se busca por X en los siguientes subarreglos:

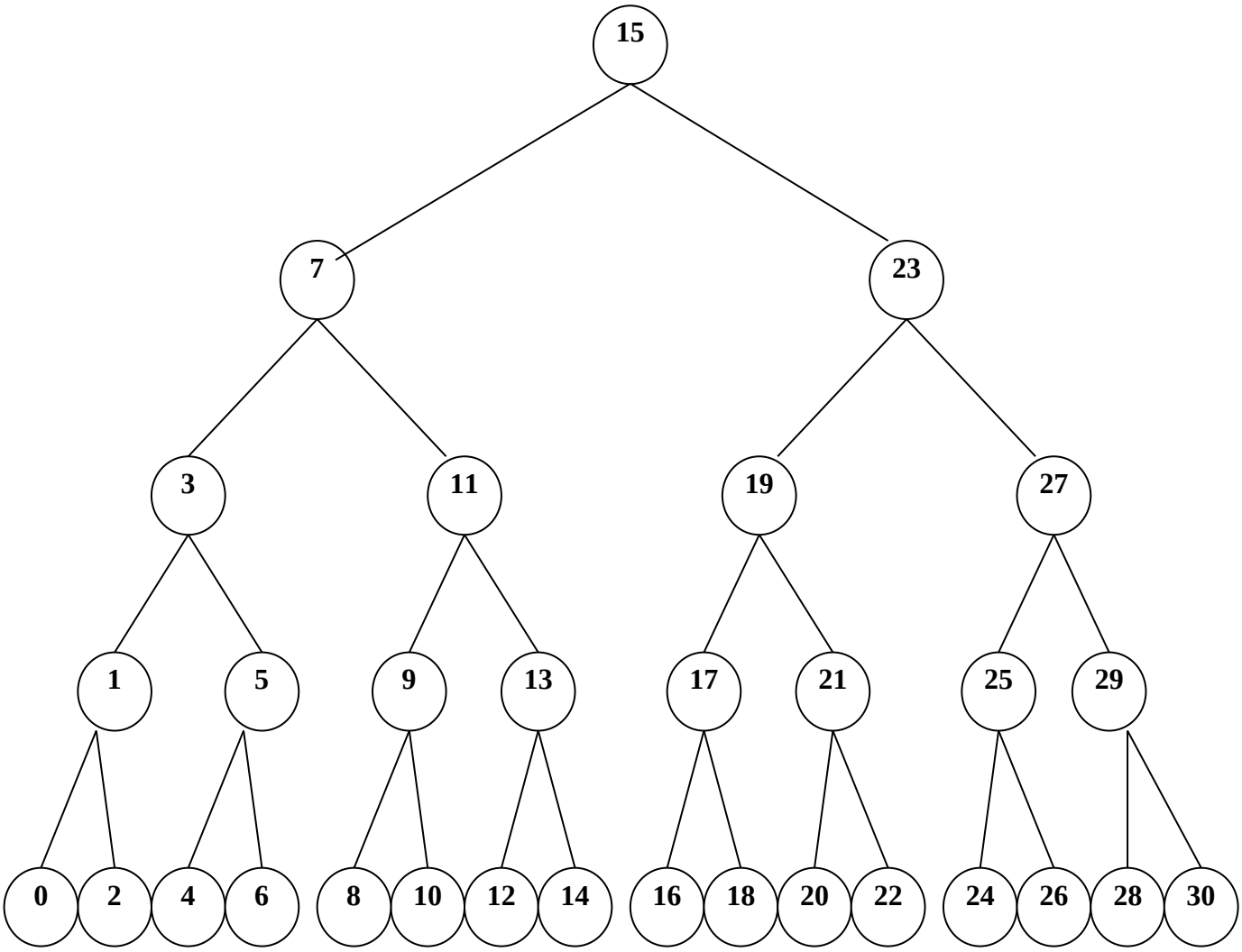
  A [0…MITAD –1]
  A [MITAD...MITAD]
  A [MITAD + 1...N-1]

- Comparando X con A [MITAD], dos de estas tres búsquedas se pueden eliminar.

- Si X está presente en el arreglo, se retorna un J tal que X=A[J]. En otro caso, J=-1.

Búsqueda binaria en un arreglo de 31 posiciones

N = 31



```
// Muestra la búsqueda binaria recursiva

public class ordArray{
  private double[ ] a; // referencia al arreglo a
  private int cantidadElementos;// cantidad de ítems de datos
  //---------------------------------

  public ordArray(int max) { //constructor
       a = new double[max]; //crea el arreglo
       cantidadElementos = 0;
  }
  //---------------------------------
  public void insert(double value) {   // coloca un elemento en el arreglo
    int j;
    for(j=0; j<cantidadElementos; j++)
      //encuentra donde va (búsqueda lineal)
      if(a[j] > value){
            break;
      }
    }
    for(int k=cantidadElementos; k>j; k--){ // corrimiento
      a[k] = a[k-1];
    }
    a[j] = value;  // lo inserta
    cantidadElementos++;    // incrementa el tamaño
   }  // fin insert
  //---------------------------------

  public int find(double searchKey) {
      return recFind(searchKey, 0, cantidadElementos-1);
  } //fin find
  //---------------------------------
```

El contenedor debe estar ordenado para poder hacer la búsqueda binaria. Es por eso, que se busca donde insertar para mantener el orden

```
private int recFind(double searchKey,int lowerBound,int upperBound){
      int curIn =(lowerBound+upperBound)/2;

      if(a[curIn]==searchKey){
            return curIn;   //lo encontró
      }
      else {
            if(lowerBound  >  upperBound){
                  return cantidadElementos; // no lo encontró
            }
       else   {   // divide el rango
           if(a[curIn] < searchKey) { //está en la mitad superior
              return recFind(searchKey,curIn+1, upperBound);
           }
           else   {
                  // está en la mitad inferior
                  return recFind(searchKey,lowerBound,curIn-1);
                 }
      }  // fin else divide rango

  }  // fin recFind
  //---------------------------------

  public int size( ) { //idem getCantidadElementos()
      return cantidadElementos;
  } //fin size
  //---------------------------------

 public double getElemI(int i) {
      return a[i];
  } //fin getElemI

}  // fin clase ordArray
```

Podría también retorna un -1

Recuerde que debe chequear i

```
public class BinarySearchApp{

   public satatic void display(ordArray a){
  // despliega el contenido del arreglo

      for(int j=0; j<a.size(); j++) {// para cada elemento, lo despliega
            StdOut.print(a.getElemI(j) + " ");
      }
      StdOut.println("");
  } //fin display
  //---------------------------------

  public static void main(String[] args) {

     int maxSize = 100; // tamaño del arreglo
     ordArray arr; // referencia al arreglo

     arr = new ordArray(maxSize); // crea el arreglo

     // inserta items
     arr.insert(72);
     arr.insert(90);
     arr.insert(45);
     arr.insert(126);
     arr.insert(54);
     arr.insert(99);
     arr.insert(144);
     arr.insert(27);
     arr.insert(135);
     arr.insert(81);
     arr.insert(18);
     arr.insert(108);
```

```
    arr.insert(9);
    arr.insert(117);
    arr.insert(63);
    arr.insert(36);

    BinarySearchApp.display();  // despliega el arreglo

    int searchKey = 27; // búsqueda del item

    if( arr.find(searchKey) != arr.size() ){
          StdOut.println("Encontrado " + searchKey);
    }
    else{
          StdOut.println("No se puede encontrar" +  searchKey);
    }

  }  // fin main

}  // fin clase BinarySearchApp
```

Se podría imprimir además la posición donde lo encontró

Ejemplo de ruteo de la búsqueda binaria recursiva. (Este código es parte de la clase Lista)

```
public int busquedaBinaria(int sueldoClave){
    return busquedaBinaria(sueldoClave, 0, cantidadEmpleados-1);
}

private int busquedaBinaria(int sueldoClave, int i, int j){
    int mitad = (i + j) / 2;
    if (listaEmpleados [mitad].getSueldo() == sueldoClave){
        return mitad; //encontrado
    }
    else{
        if (i > j){
            return listaEmpleados.length; // No está
        }
        else{
            if (listaEmpleados [mitad].getSueldo() < sueldoClave){
                //Está en la mitad superior
                return busquedaBinaria(sueldoClave, mitad + 1, j);
            }
     else{
                //Está en la mitad inferior
                return busquedaBinaria(sueldoClave, i, mitad - 1);
            }
        }
}
```

return cantidadEmpleados;
o
return -1;

listaEmpleados

i	0	1	2	3	4	5	6	7
Empleado	Maria	Miguel	Javiera	Felipe	Pedro	Juan	Roberto	Nicolas
	300	400	500	600	650	700	750	800

listaEmpleados.busquedaBinaria(700)

sueldoClave
700

return

busquedaBinaria(700, 0, 7)

sueldoClave	i	j	mitad
700	0	7	3

return

busquedaBinaria(700, 4, 7)

sueldoClave	i	j	mitad
700	4	7	5

return 5

return 5

return 5

A continuación, se muestra una versión iterativa de la búsqueda binaria.

```
public class OrdArray {

  private double[] a; // referencia al arreglo a
  private int cantidadElementos; //cantidad de items

  //---------------------------------

  public OrdArray(int max)  { //constructor
    a = new double[max];  //crea el arreglo
    cantidadElementos = 0;
  }
  //---------------------------------

  public int size(){
    return cantidadElementos;
  } //fin size

 //---------------------------------
  public void insert(double value)
  //coloca el elemento en el arreglo
  //---------------------------------

  public boolean delete(double value)
  //elimina un elemento del arreglo
  //---------------------------------
```

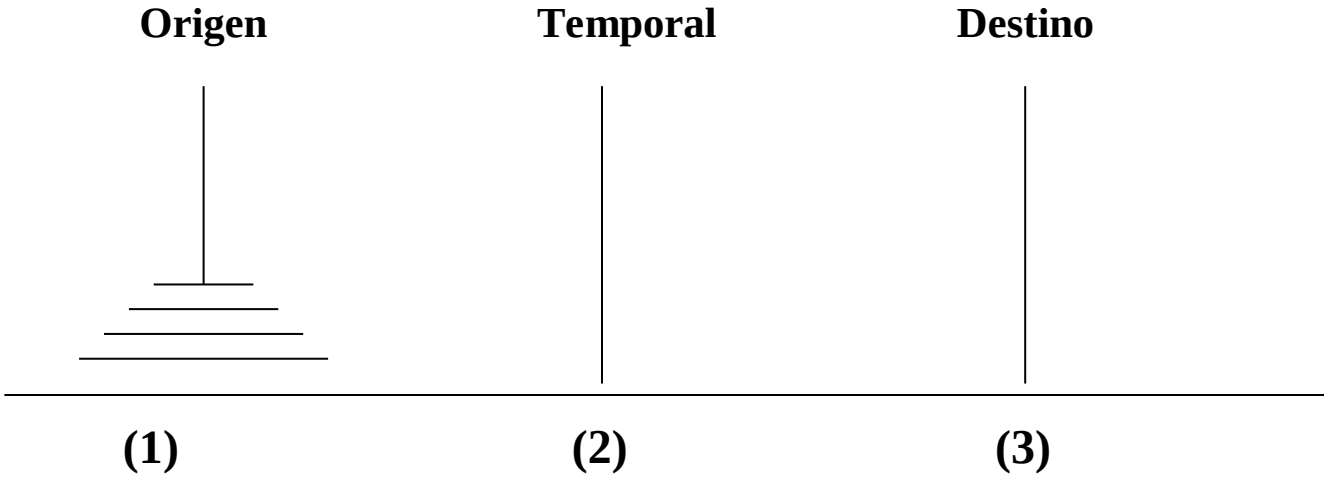
```
public int find(double searchKey) {
   int lowerBound = 0;
   int upperBound = cantidadElementos-1;
   int curIn;

   while(true) {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey){
            return curIn;  // lo encontró
      }
      else {
            if(lowerBound >upperBound) {
                  //no lo encontró
                  return cantidadElementos;
            }
            else {  // divide el rango
                  if(a[curIn] < searchKey){
                        lowerBound = curIn + 1;
                  }
                  //está en la mitad superior
                  else{
                        upperBound = curIn - 1;
                        //está en la mitad inferior
                  }
            }  // end else divide el rango
      } // fin else
   }//fin while

} // fin find
} // fin clase OrdArray
```

```
do {
    curIn = (lowerBound +
                upperBound ) / 2;

   if(a[curIn] < searchKey){
      lowerBound = curIn + 1;
   }
   else{
       upperBound = curIn - 1;
   }
}while (a[curIn] != searchKey
                and
      lowerBound <= upperBound)

if (a[curIn] == searcKey){
     return curIn;
 }
else {
     return nElems;
}
```

```
public class OrderedApp{

   public static void display(….) // despliega el contenido del arreglo

   //---------------------------------

   public static void main(String[] args){
      int maxSize = 100; // tamaño del arreglo
      OrdArray arr;  // referencia al arreglo

      arr = new OrdArray(maxSize); // crea el arreglo

      //inserta 10 items
      arr.insert(77);
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      int searchKey = 55; //búsqueda del item

      if( arr.find(searchKey) != arr.size() ) {
         StdOut.println("Encontrado " + searchKey);
      }
      else {
         StdOut.println("No lo puede encontrar " + searchKey);
       }
}  // fin main

}  // fin clase OrderedApp
```

Juego de las torres de Hanoi.

El objetivo del juego es mover la torre desde la posición de la izquierda a la de la derecha, de forma tal que en ningún momento quede un disco colocado sobre otro de tamaño inferior, para lo que se puede emplear el almacenamiento temporal. (Sólo se mueve un disco a la vez).



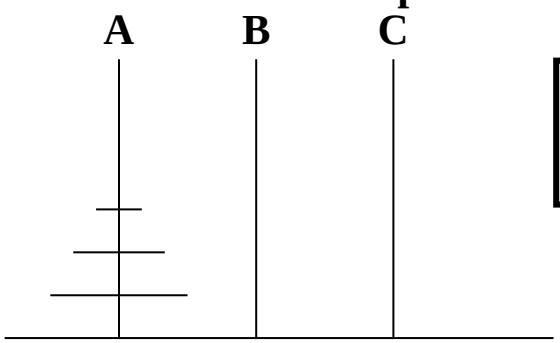
Se requiere diseñar un algoritmo que imprima ordenadamente, los movimientos que se deben efectuar para resolver el problema.

Nota: Sea N, la cantidad de discos que componen la torre. Entonces, el número de movimientos a efectuar para llevar a cabo el proceso es: $2^N - 1$

- **Para N = 3:**
  X $\rightarrow$ Y denota la instrucción:
  " Mover el disco superior desde el poste X al poste Y".
  X, Y: serán uno de los tres postes.

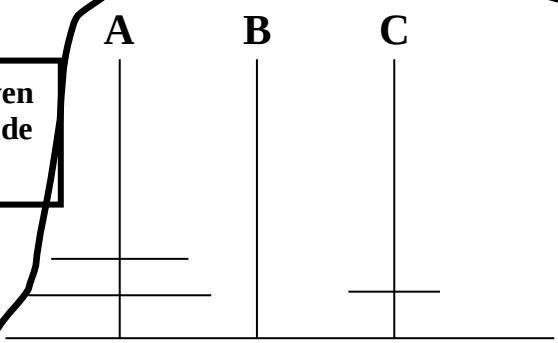
  Movimientos necesarios:
  A $\rightarrow$ C
  A $\rightarrow$ B
  C $\rightarrow$ B
  A $\rightarrow$ C
  B $\rightarrow$ A
  B $\rightarrow$ C
  A $\rightarrow$ C

Movimientos necesarios para mover 3 discos

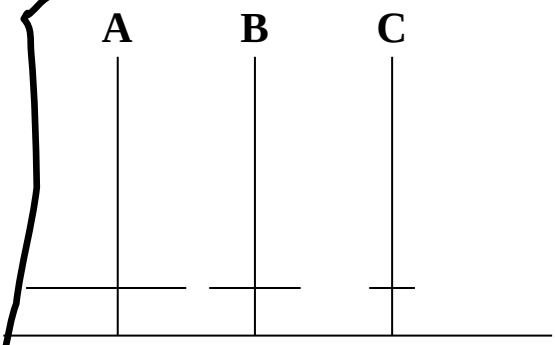


Inicial

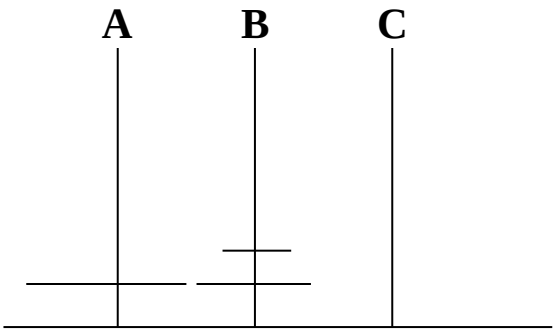
Se mueven 2 discos de A a B



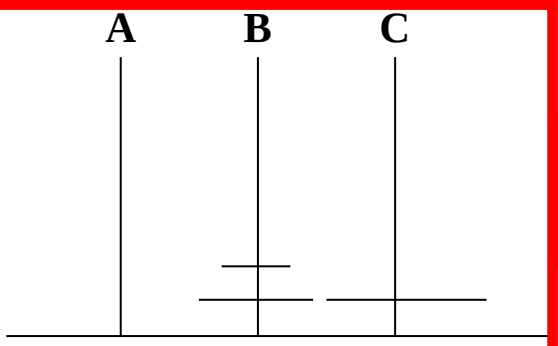
1) A → C



2) A → B

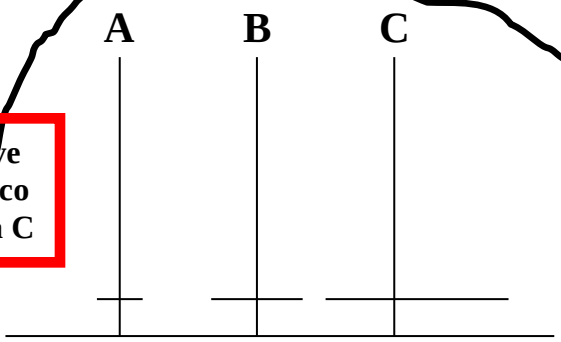


3) C → B

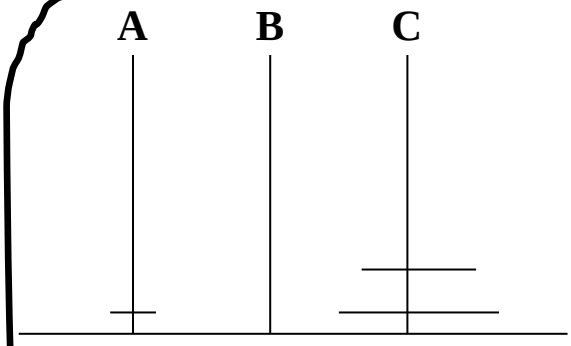


4) A → C

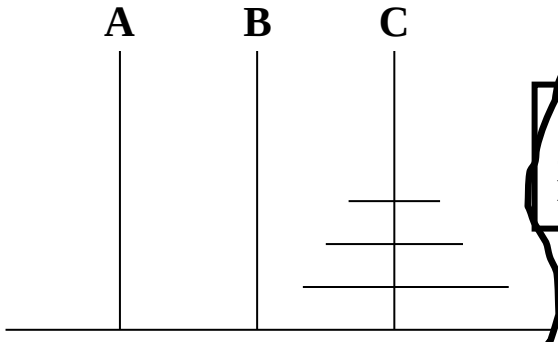
Se mueve un disco de A a C



5) B → A



6) B → C

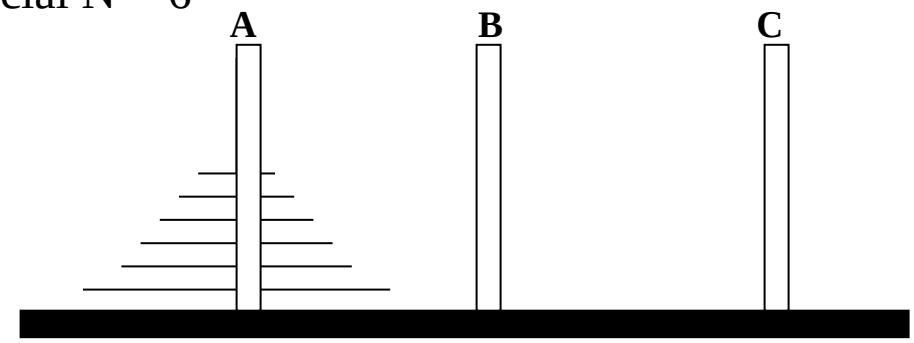


Se mueven 2 discos de B a C

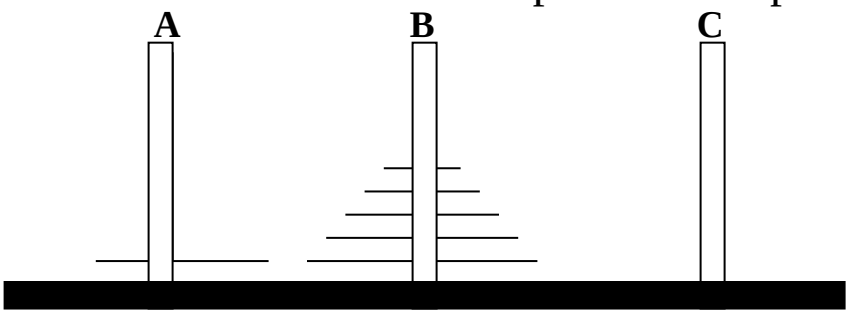
7) A → C

- **Para N = 6: Movimientos**

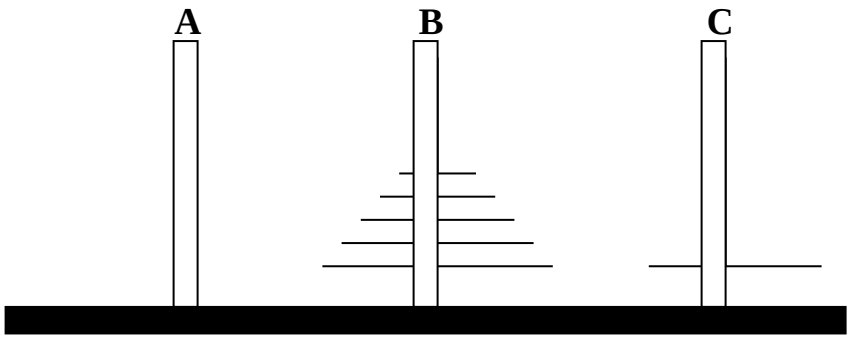
a) Inicial N = 6



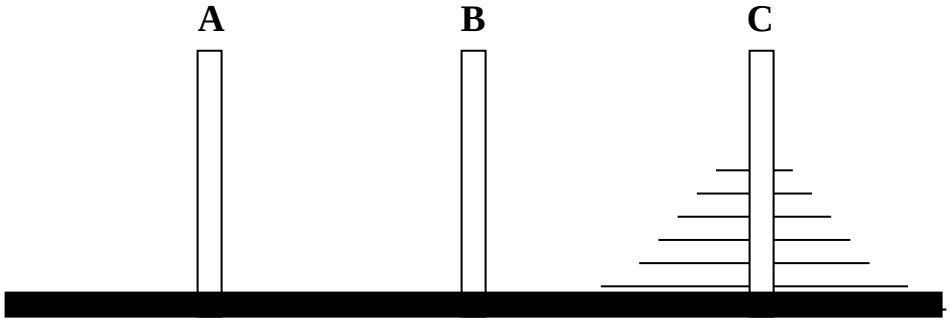
b) Se mueven los cincos discos superiores del poste A al poste B



c) Se mueve el disco superior desde el poste A al poste C



d) Se mueven los cinco discos superiores del poste B al poste C



- **Solución para el caso de 64 discos:**

MOVER (64,1,3)

La tarea MOVER (64,1,3) es equivalente a la siguiente secuencia de subtareas:

- MOVER(63,1,2)
- "Mover un disco desde la posición 1 a la 3"
- MOVER (63,2,3)

Esta descomposición se puede aplicar tantas veces como sea necesario, hasta que el movimiento de los discos sea una tarea trivial.

Por ejemplo:

MOVER (63, 1, 2) se puede expresar como:

- MOVER (62, 1, 3)
- "Mover un disco desde la posición 1 a la 2"
- MOVER (62, 3, 2)

Procedimiento general:

MOVER (N,A,B,C)

Objetivo:

Mover N discos desde la posición A la B, usando la posición C, para almacenamiento temporal de la torre.

MOVER (N,A,B,C) se puede llevar a cabo en 3 etapas:

- MOVER (N-1,A,C,B)
- "Mover un disco desde la posición A la B"
- MOVER (N-1,C,B,A)

Este proceso es válido si N > 1.

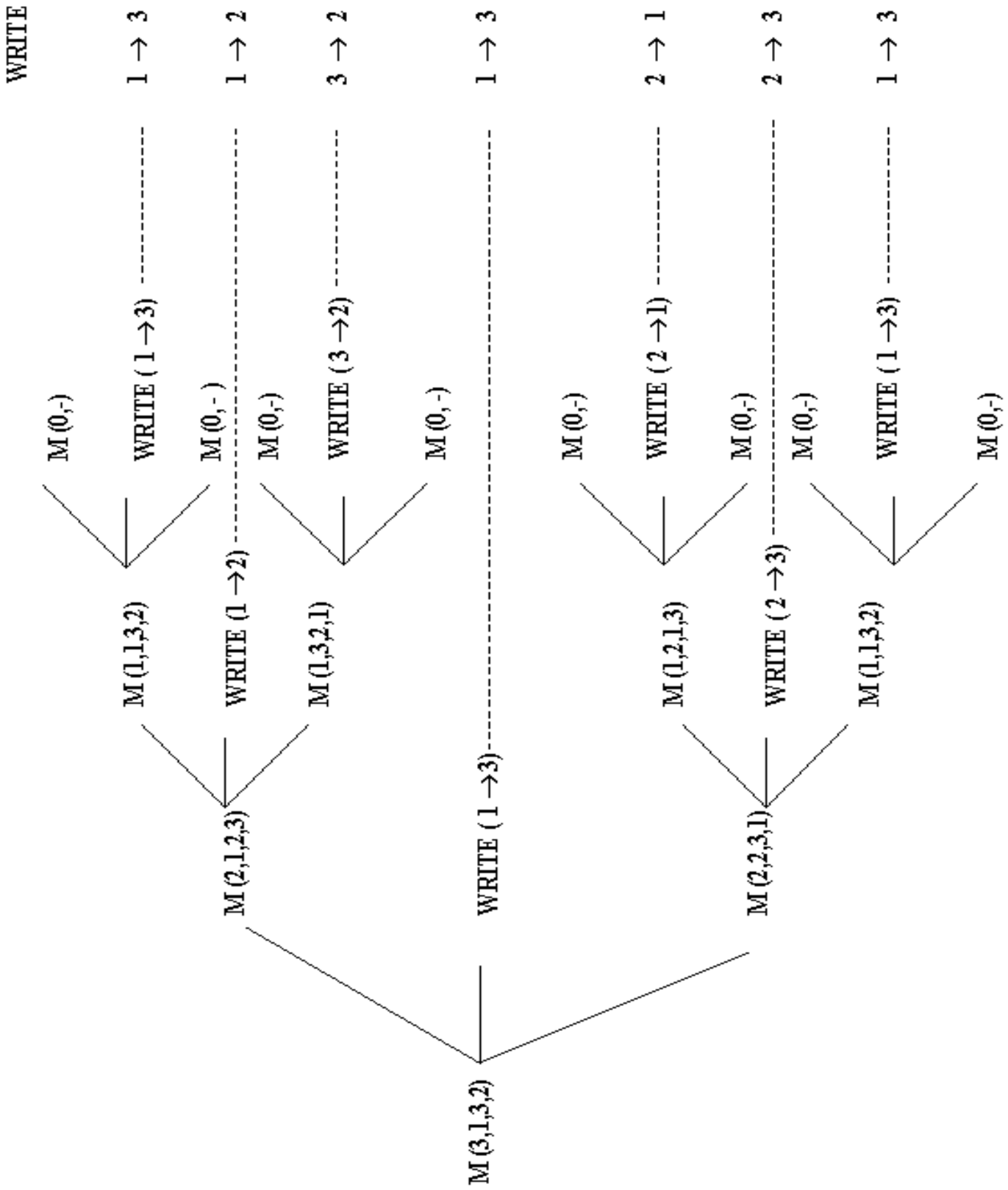
Solución:

```
procedure MOVER (numero, desde, hacia, temp)
begin
   if (numero >0) then
   begin
      MOVER (numero –1, desde, temp, hacia)
      write (desde,' → ' , hacia)
      MOVER(numero –1, temp, hacia, desde)
   end
end
```

Programa principal:

Para torre de 3 discos: MOVER (3,1,3,2)

Árbol generado por los llamados recursivos al tener N = 3



La solución en Java se muestra a continuación.

```
import ucn.StdOut;

public class TowersApp {
    static int nDisks = 3;

    public static void main(String[] args) {
        TowersApp.doTowers (nDisks,'A','B','C');
    }

    public static void doTowers (int topN, char src,
                                 char inter, char dest){
        if(topN==1){
            StdOut.println("Disco desde " + src +
                                          " a " + dest);
        }
        else{
            // mueve desde src hacia inter
            TowersApp.doTowers(topN-1, src, dest,
                                              inter);
            // mueve disco del fondo
            StdOut.println("Disco desde " + src +
                                          " a " + dest);
            // mueve desde inter hacia dest
            TowersApp.doTowers(topN-1, inter, src,
                                               dest);
        }
    }
}
```

Resultado:

```
Disco desde A a C
Disco desde A a B
Disco desde C a B
Disco desde A a C
Disco desde B a A
Disco desde B a C
Disco desde A a C
```

En resumen, la recursión permite resolver un problema a través de la solución de versiones más pequeñas del mismo problema.

Estructuras de datos recursivas: Árboles

Conceptos

Listas lineales: En las listas lineales, cada elemento tiene dos nodos como vecinos: un antecesor y un sucesor.

Ejemplo:
- Elementos de un arreglo unidimensional
- Campos en un registro
- Datos en un stack

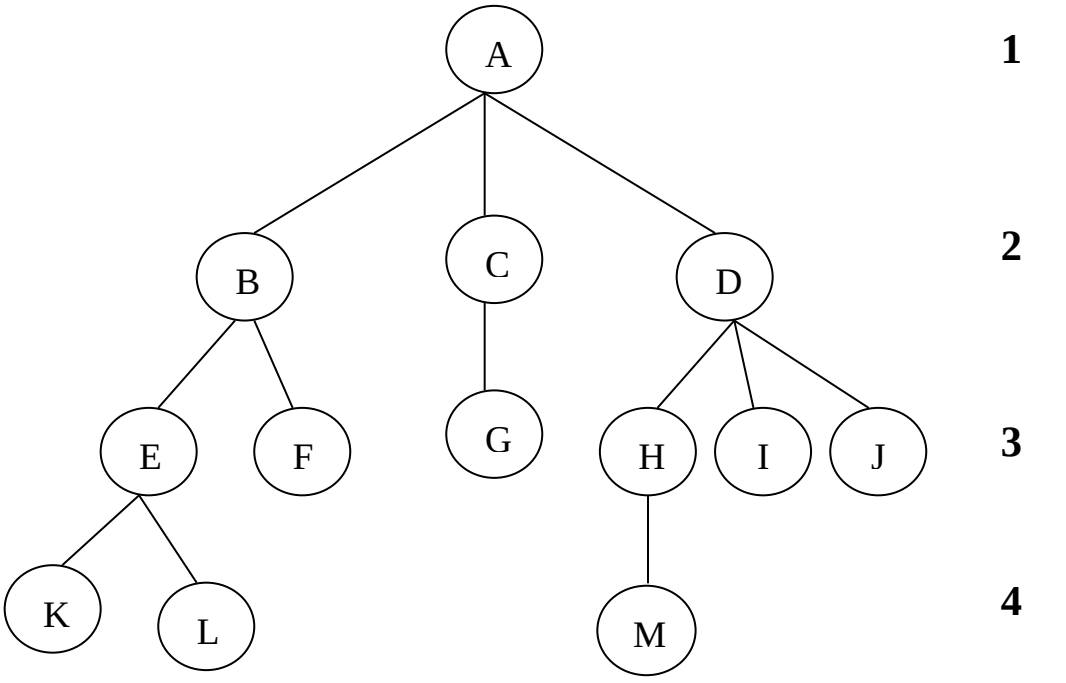
Listas no lineales: En las estructuras no lineales, cada elemento puede tener varios siguientes elementos, lo que introduce el concepto de estructura con ramificaciones.

Definición: Un árbol es un conjunto finito de uno o más nodos tal que:

1) Hay un nodo especial llamado raíz.

2) Los nodos restantes son particionados en n ≥, 0 conjuntos disjuntos T_1,T_2, ...T_n, donde cada uno de estos conjuntos es un árbol.
T_1, T_2,..., T_n son llamados los subárboles de la raíz.

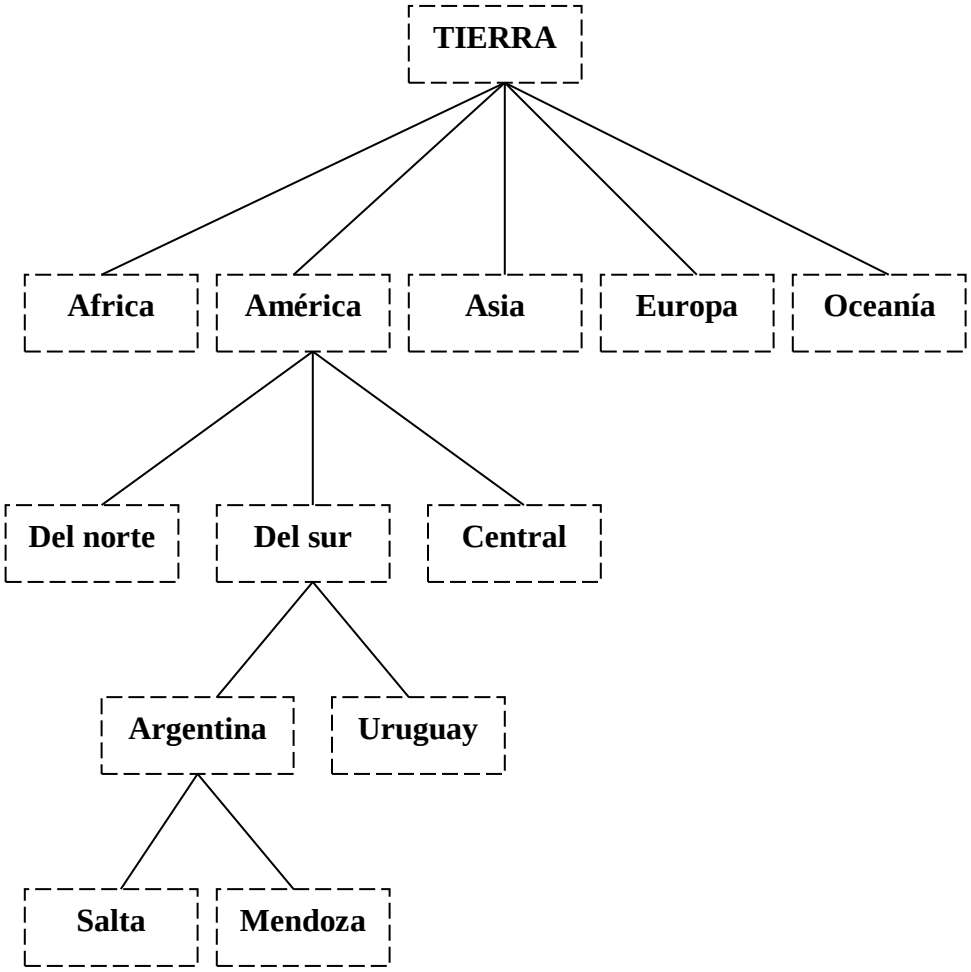
Ejemplo de árbol



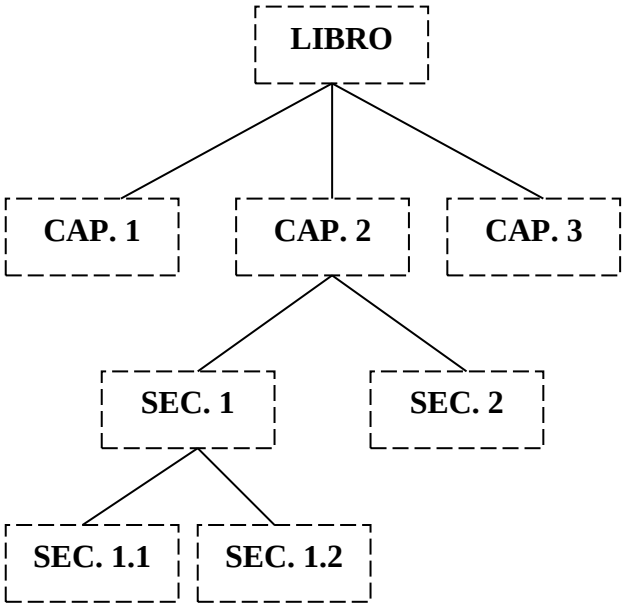
Utilidad de los árboles

i) Estructura de datos que se prestan al concepto de **jerarquía**.

Ejemplo 1: Jerarquía. Da lo mismo el orden.



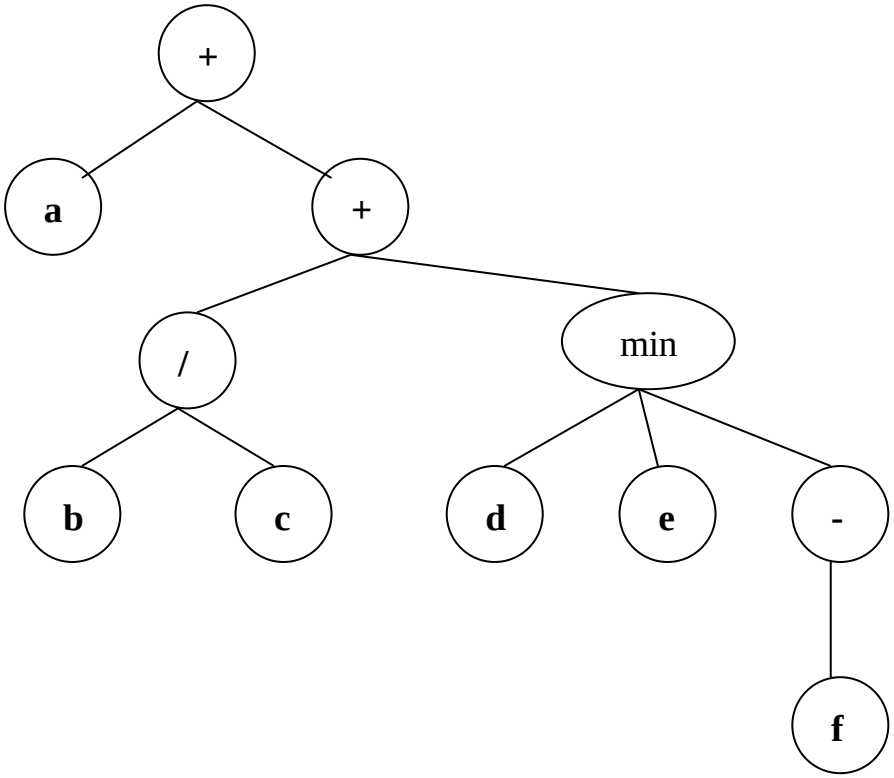
Ejemplo 2: Orden es relevante.



ii) Estructura Recursivas.

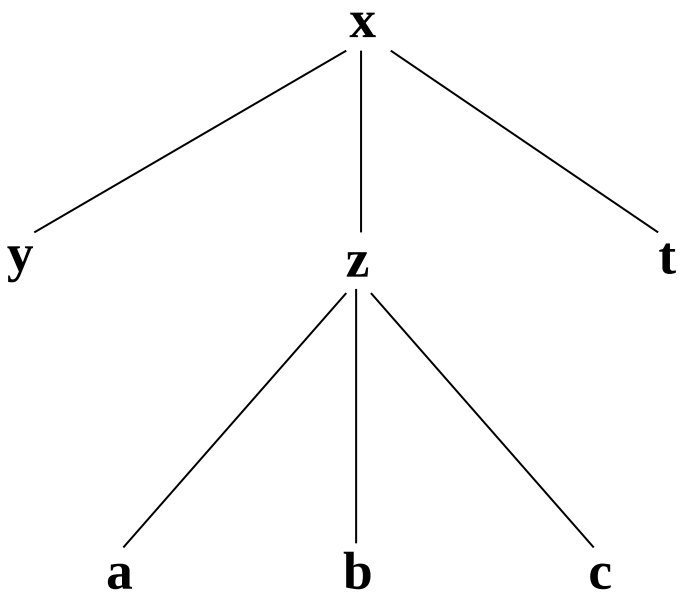
Ejemplo 1: Ejemplo de expresión matemática. Importa el orden.

a + b/ c + min (d, e, -f)



Ejemplo 2: Jerarquía de un registro. No importa el orden.

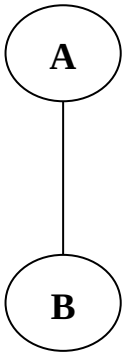
```
Var
  x: record
       y: integer;
       z: record
            a: char;
            b: integer;
            c: real;
          end;
       t: boolean;
  end;
```



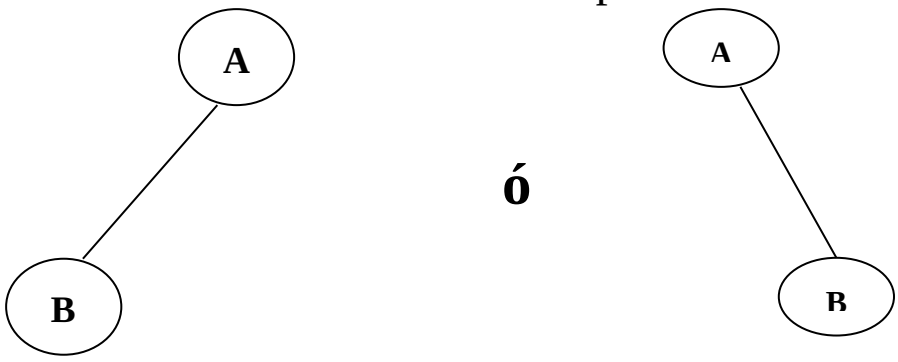
Árboles binarios

Definición: Un árbol binario es un conjunto finito de nodos, el que puede estar vacío o consistir de una raíz y dos árboles binarios disjuntos, llamados el subárbol izquierdo y el subárbol derecho.

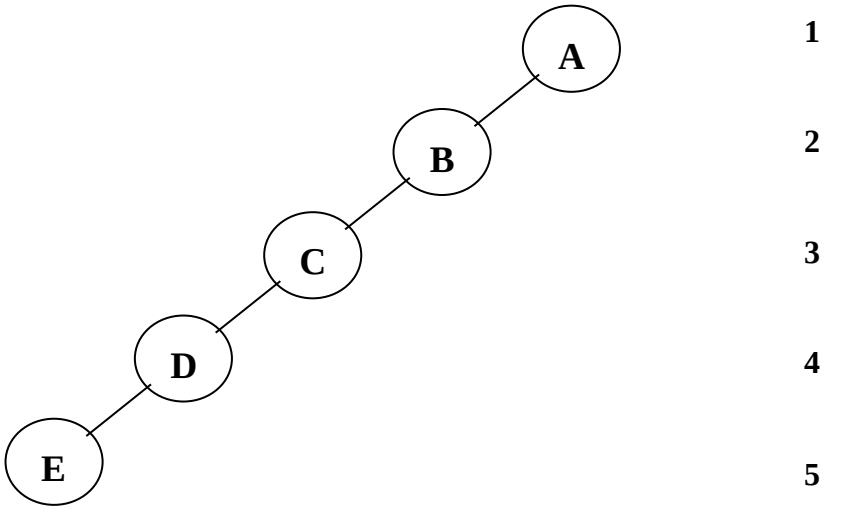
Notar que en árboles se puede tener

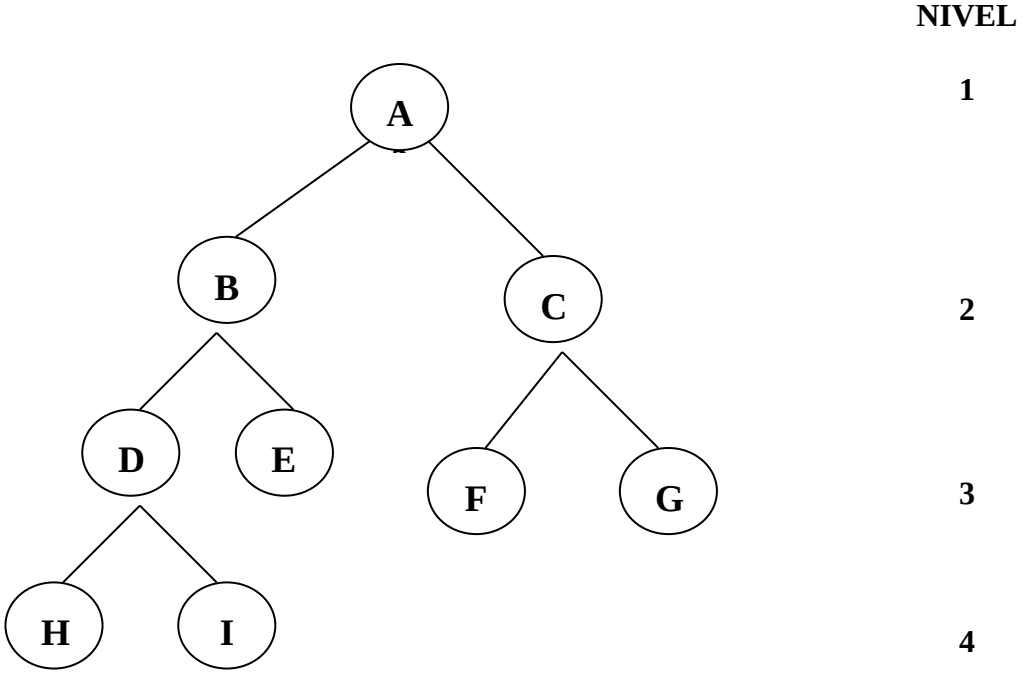


Los árboles binarios mostrados corresponden a árboles binarios diferentes.

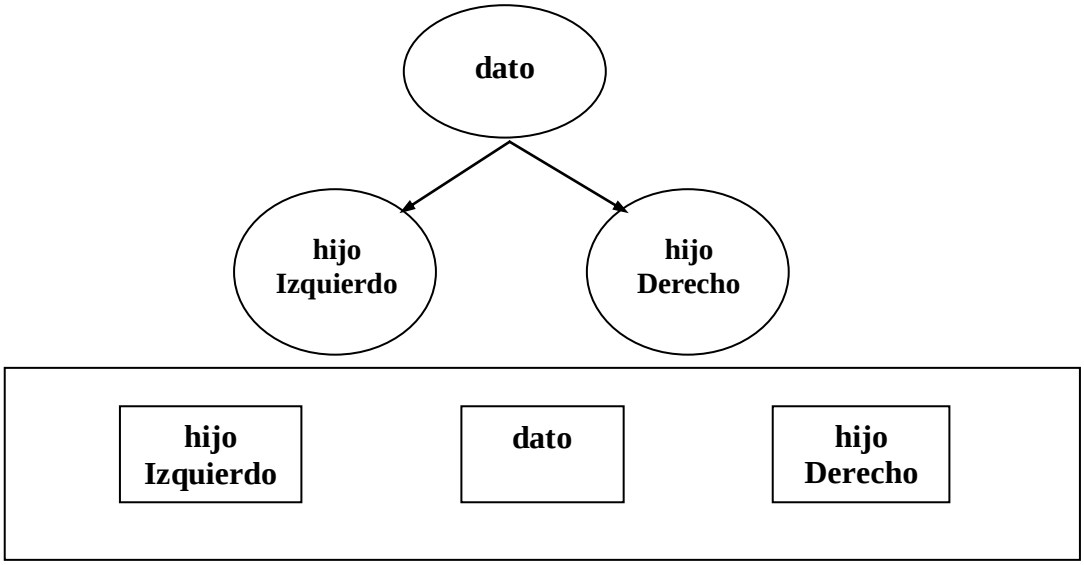


Ejemplos de árboles binarios



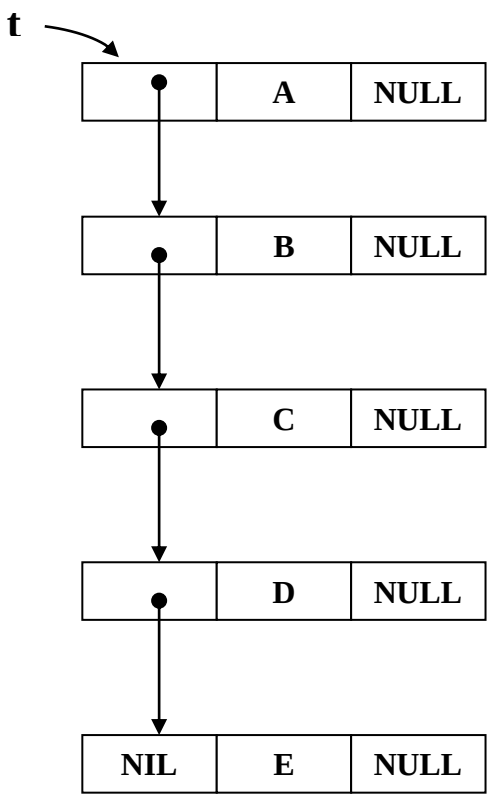


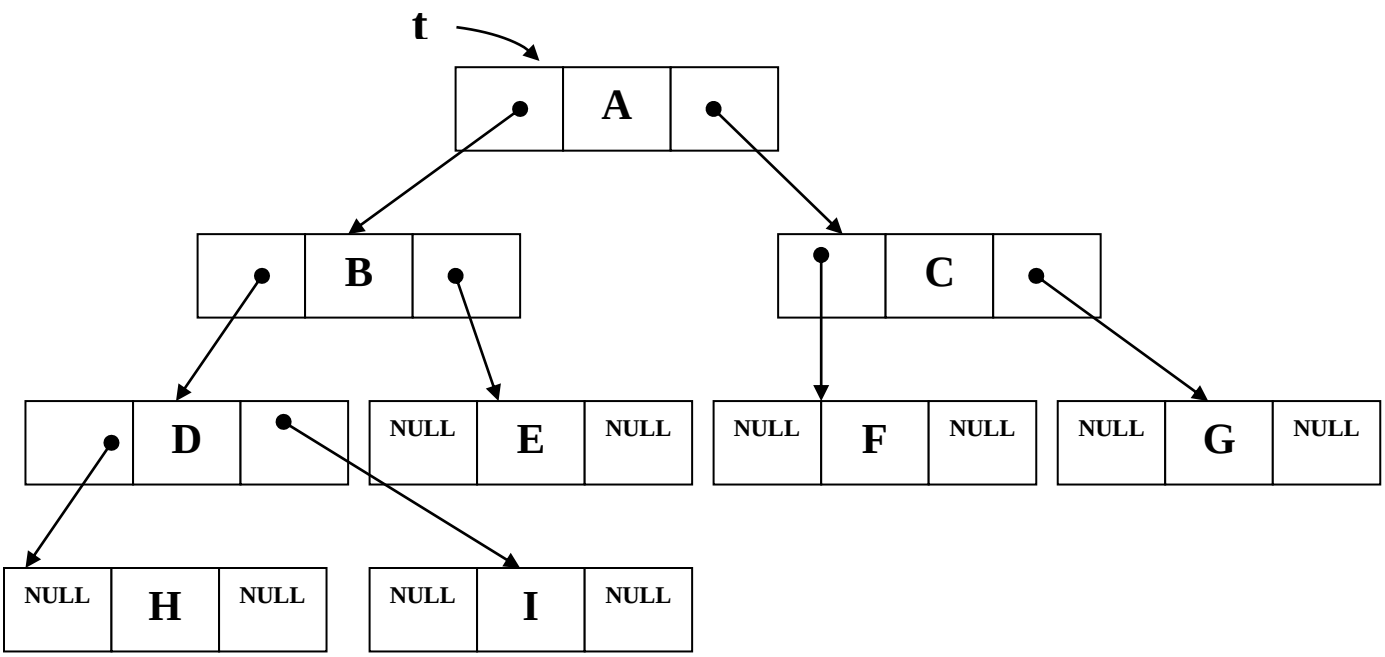
Representación de un nodo en un árbol binario



NODO

Ejemplos de árbol binario, utilizando nexos





Representación de árboles binarios (representación con nexos):

```
public class Nodo{
   private int dato;
   private Nodo hijoIzquierdo;
   private Nodo hijoDerecho;

   public Nodo(int dato){
      this.dato = dato;
      hijoIzquierdo = null;
      hijoDerecho = null;
   }
   public int getDato() {….
   …….
}

class ArbolBinario {
   private Nodo raiz;

   public ArbolBinario (){
       raiz = null;
   }
}
```

Recorrido de árboles binarios:

- Recorrer un árbol, es examinar una sola vez, cada uno de sus nodos.

- Existen tres tipos de actividades en el recorrido de un árbol binario:
  i) Visitar la raíz
  ii) Recorrer el subárbol izquierdo.
  iii) Recorrer el subárbol derecho.

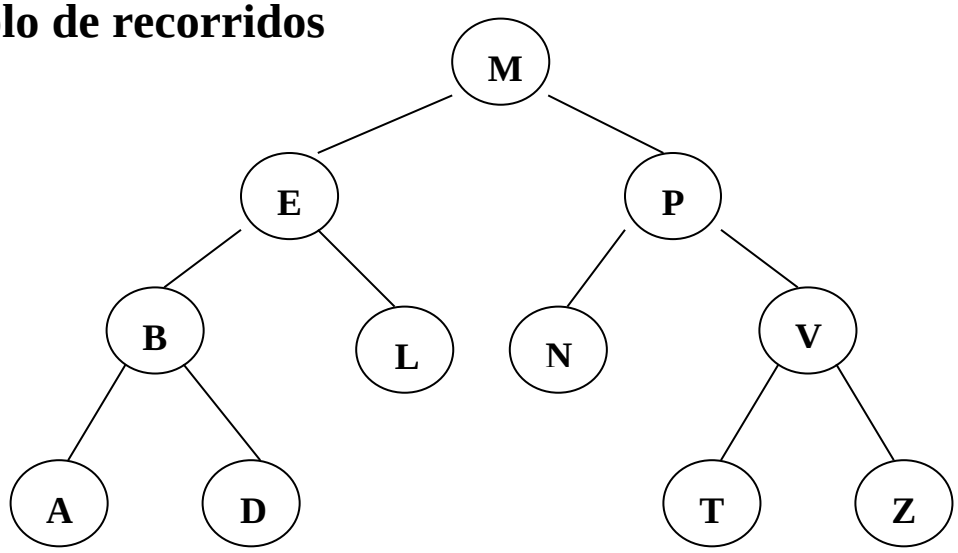
- El tipo de recorrido difiere, de acuerdo al orden en que se efectúan estas actividades.

a. Preorden:
1. Visitar la raíz.
2. Recorrer el subárbol izquierdo, en preorden.
3. Recorrer el subárbol derecho, en preorden.

b. Inorden:
1. Recorrer el subárbol izquierdo, en inorden.
2. Visitar la raíz.
3. Recorrer el subárbol derecho, en inorden.

c. Posorden:
1. Recorrer el subárbol izquierdo, en posorden.
2. Recorrer el subárbol derecho, en posorden.
3. Visitar la raíz.

Ejemplo de recorridos

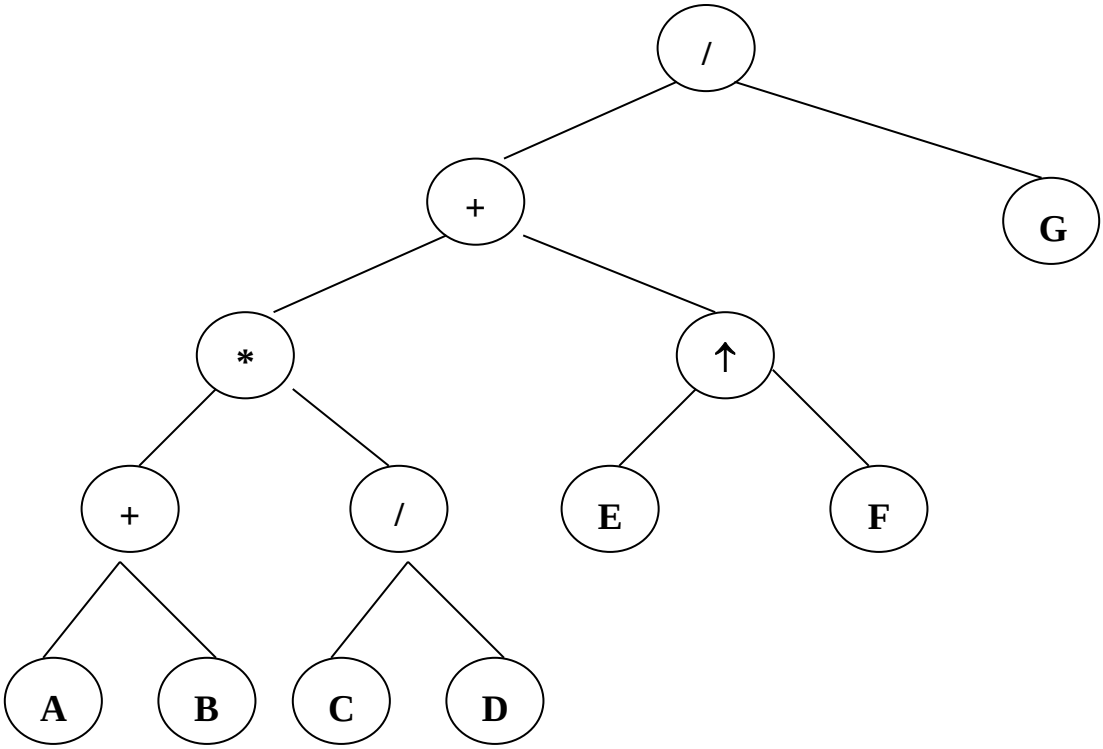


Preorden: M E B A D L P N V T Z

Inorden: A B D E L M N P T V Z

Posorden: A D B L E N T Z V P M

Ejemplo: Expresión aritmética: (((a + b) * c/d) + e ↑ f) /g



Preorden: / + * + A B / C D ↑ E F G (Forma prefijo)

Inorden: A + B * C / D + E ↑ F / G (Forma infijo)

Posorden: AB + CD / * EF ↑+ G/ (Forma posfijo)

En la representación infijo, se puede perder el orden de precedencia señalado en la expresión aritmética original, en caso de haber utilizado paréntesis para anular la jerarquía de evaluación de los distintos operadores.

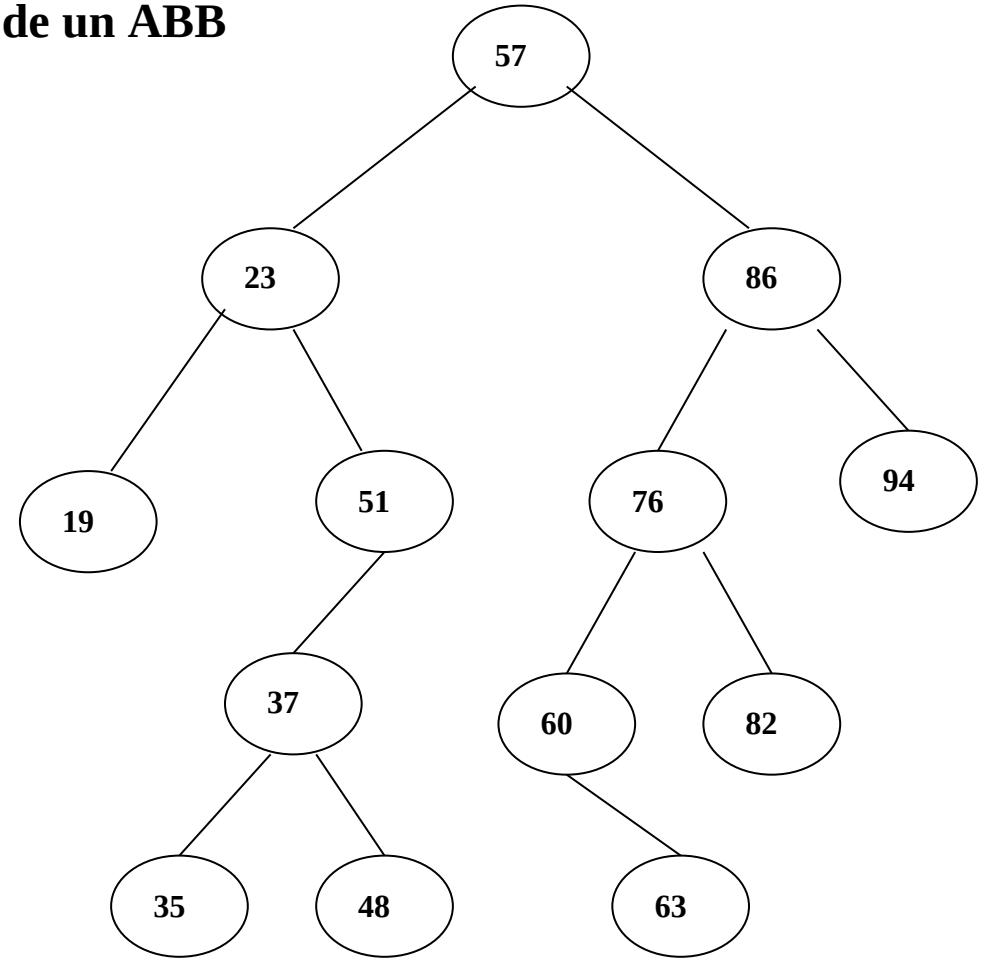
Como en la notación posfijo, se mantiene el orden de precedencia, el recorrido posorden es conveniente, para una correcta evaluación de la expresión aritmética.

Árboles binarios de busqueda, ABB (Binary Search Trees)

Definición: Un ABB, T, es un árbol binario, ya sea vacío ó en que cada nodo en el árbol contiene un identificador y se cumple:

i) Todos los identificadores en el subárbol izquierdo de T son menores (numérica o alfabéticamente) que el identificador en la raíz T.
ii) Todos los identificadores en el subárbol derecho de T son mayores que el identificador en la raíz de T.
iii) Los subárboles izquierdo y derecho de T son también ABB.

Ejemplo de un ABB



Formato Nodo:

El recorrido inorden: Permite recorrer el árbol, en orden ascendente del identificador.

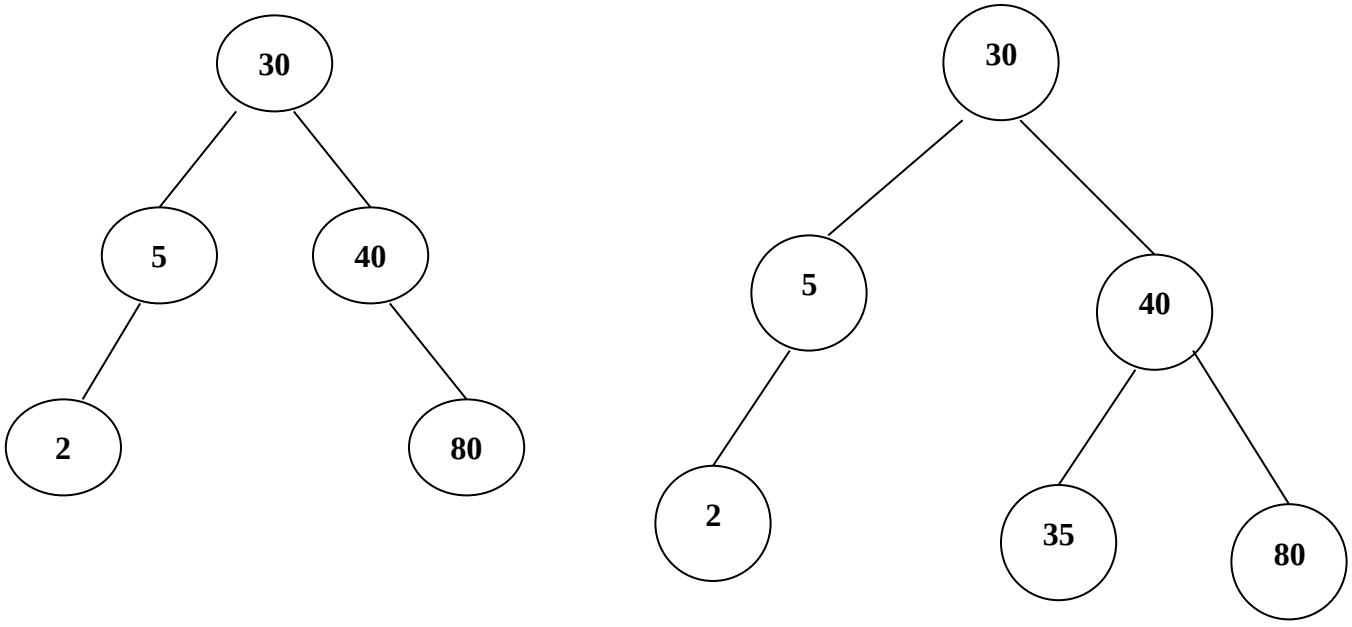
Cantidad de niveles v/s cantidad de nodos en un ABB

Cant. Nodos	Cant. Niveles
1	1
3	2
7	3
15	4
31	5
1.023	10
32.767	15
1.048.575	20
33.554.432	25
1.073.741.824	30
$2^N - 1$	N

Inserción en un ABB:

"Si el dato no se encuentra en el ABB, el dato se inserta en el punto en que termina la correspondiente búsqueda infructuosa."

Ejemplo de inserción: Insertar el 35.



Un ejemplo de utilización de un árbol binario de búsqueda se muestra a continuación.

```
public class NodoABB {
    private int dato; // item de dato (clave)
    private NodoABB hijoIzquierdo; //hijo izquierdo
    private NodoABB hijoDerecho; //hijo derecho

    public NodoABB (int dato){
        this.dato = dato;
        hijoIzquierdo = null;
        hijoDerecho = null;
    }

    public int getDato() {
        return dato;
    }
    public NodoABB getHijoIzquierdo() {
        return hijoIzquierdo;
    }
    public void setHijoIzquierdo(NodoABB hijoIzquierdo) {
        this.hijoIzquierdo = hijoIzquierdo;
    }
    public NodoABB getHijoDerecho() {
        return hijoDerecho;
    }
    public void setHijoDerecho(NodoABB hijoDerecho) {
        this.hijoDerecho = hijoDerecho;
    }
    public void setDato(int dato) {
        this.dato = dato;
    }
}
```

```
import ucn.StdOut;

public class ABB {
    private NodoABB raiz;
    //referencia al nodo raíz del árbol

    public ABB()    {  // constructor
        raiz = null;
    }

    public NodoABB encontrar (int clave){
    //encuentra el nodo con clave dada
        return buscar (raiz, clave);
    }// fin encontrar

    private NodoABB buscar (NodoABB nodo, int clave) {
        //Busca en el árbol, un nodo con la clave clave.
        //Retorna una referencia al nodo, si se encuentra.
        //Retorna NULL en caso contrario
        if(nodo == null) {
            return null;
        }
        else {
            if(clave == nodo.getDato()) {
                return nodo;
            }
            else {
                if(clave < nodo. getDato()) {
                    return buscar(nodo.getHijoIzquierdo(),
                                                    clave);
                }
                else{
                    return buscar(nodo.getHijoDerecho(),
                                                    clave);
                }
            }
        } //fin else
    }// fin de buscar
```

```
public void insertar (int id) {
    //Algoritmo iterativo para para el método insertar

    NodoABB nodo = new NodoABB(id);

    if(raiz==null) {
        raiz = nodo;
        //el nodo que se inserta queda como la raíz
    }
    else  {   // árbol inicial no está vacio
        // comienza en la raíz
        NodoABB current = raiz;
        NodoABB padre;
        while(true) {
        //la salida del ciclo se efectua internamente
            padre = current;
            if(id < current.getDato()) {
                //¿va a la izquierda?
                current = current.getHijoIzquierdo();
                if(current == null) {
                    //si es fin de la rama del
                    //árbol, inserta a la izquierda
                    padre.setHijoIzquierdo(nodo);
                    return;
                }
            }  // fin si va a la izquierda
            else {  // ¿o va a la derecha?
                current = current.getHijoDerecho();
                if(current == null) {
                    //si es fin de la rama del árbol
                    //inserta a la derecha
                    padre.setHijoDerecho(nodo);
                    return;
                }
            } //fin si va a la dereha
        }  // fin while
    }  // fin del else
}  // fin insertar
```

Construir algoritmo recursivo para insertar en un ABB

```
//-------------------------------------
//public boolean deletear(int clave)
//-------------------------------------

public void recorrer (int tipoRecorrido) {
    switch(tipoRecorrido) {
    case 1:
        StdOut.print("\nRecorrido Preorden: ");
        this.preOrden(raiz);
    break;

    case 2:
        StdOut.print("\nRecorrido Inorden:  ");
        this.inOrden(raiz);
    break;

    case 3:
        StdOut.print("\nRecorrido Posorden: ");
        this.postOrden(raiz);
    break;
    } //Fin switch

}// Fin recorrer

private void preOrden (NodoABB nodo) {
    if(nodo != null) {
        StdOut.print( nodo.getDato()+ "  ");
        preOrden(nodo.getHijoIzquierdo());
        preOrden(nodo.getHijoDerecho());
    }
}

private void inOrden (NodoABB nodo) {
    if(nodo != null) {
        inOrden(nodo.getHijoIzquierdo());
        StdOut.print(nodo.getDato()+ "  ");
        inOrden(nodo.getHijoDerecho());
    }
}
```

```
    private void postOrden (NodoABB nodo) {
        if(nodo != null) {
            postOrden(nodo.getHijoIzquierdo());
            postOrden(nodo.getHijoDerecho());
            StdOut.print(nodo.getDato() + "  ");
        }
    }

    //-------------------------------------
    //public void desplegarArbol()
    //-------------------------------------

}  // fin clase ABB
```

```
import ucn.StdOut;
import ucn.StdIn;

public class AppABB {

    public static void main (String[] args) {
        ABB abb = new ABB();
        abb.insertar(50);
        abb.insertar(25);
        abb.insertar(75);
        abb.insertar(12);
        abb.insertar(37);
        abb.insertar(43);
        abb.insertar(30);
        abb.insertar(33);
        abb.insertar(87);
        abb.insertar(93);
        abb.insertar(97);

        opcionesABB(abb);
    }  // fin main()
```

```
public static void opcionesABB(ABB abb) {
    StdOut.print("Entre primera letra de");
    StdOut.print("mostrar, insertar, ubicar, eliminar,
                                    recorrer o salir: ");
    int valor;
    char opcion = StdIn.readChar();

    while(opcion!= 's') {

        switch(opcion) {
        case 'm':
            //abb.desplegarArbol();
        break;

        case 'i':
            StdOut.print("Ingresar valor a insertar: ");
            valor = StdIn.readInt();
            abb.insertar(valor);
        break;

        case 'u':
            StdOut.print("Ingresar valor a ubicar: ");
            valor = StdIn.readInt();
            NodoABB encontrado = abb.encontrar(valor);
            if(encontrado != null){
                StdOut.print("Encontrado: ");
                StdOut.print( encontrado.getDato()+" ");
                StdOut.print("\n");
            }
            else {
                StdOut.print("No se pudo encontrar" +
                                        valor  + '\n');
            }
        break;
```

```
            case 'e':
                StdOut.print("Ingresar valor a deletear: ");
                valor = StdIn.readInt();
                /*boolean seElimino = abb.deletear(valor);
                if(seElimino){
                    StdOut.print("Eliminado " + valor +
                                                    '\n');
                }
                else {
                    StdOut.print("No se pudo eliminar " +
                                        valor +  '\n');

                }*/
            break;

            case 'r':
                StdOut.print("Ingrese tipo 1, 2 o 3: ");
                valor = StdIn.readInt();
                abb.recorrer(valor);
            break;

            default:   StdOut.print("Entrada inválida\n");
            }  // fin switch

            StdOut.print("Entre primera letra de");
            StdOut.print("mostrar, insertar, ubicar,
                        eliminar, recorrer o salir: ");
            opcion = StdIn.readChar();
        }  // fin while

    }//Fin OpcionesABB

} //Fin class AppABB
```

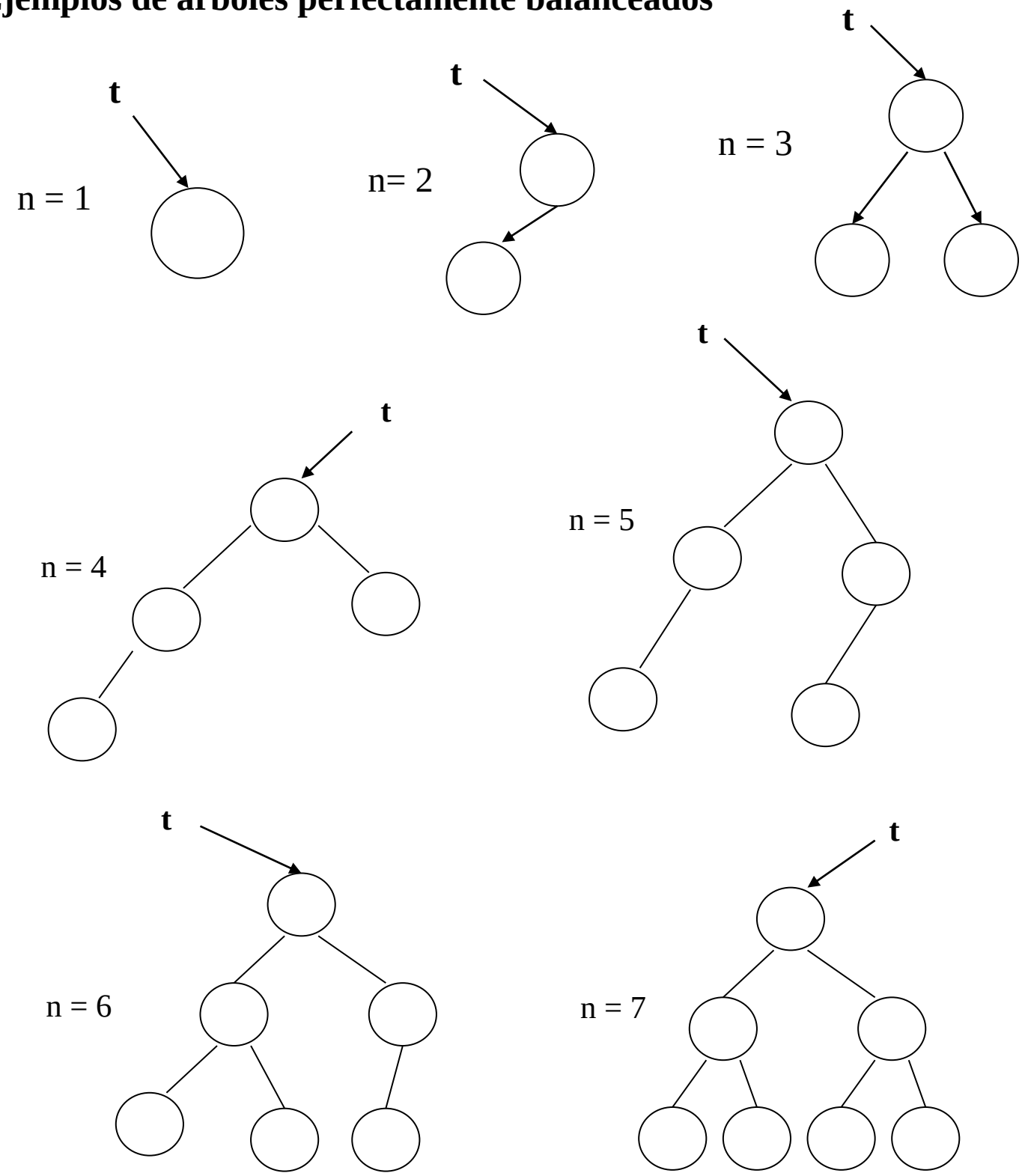
Algoritmo iterativo para el método encontrar en un ABB

```
public Nodo encontrar (int clave) {
//encontrar nodo con clave dada
// (asume un árbol no vacío)

    Nodo current = raiz;// comenzar en la raíz
    while(current.getDato() != clave) {//mientras no se encuentre el dato
        if(clave < current.getDato()) {  // ir a la izquierda
            current = current.getHijoIzquierdo();
        }
        else { // ir a la derecha
            current =   current.getHijoDerecho();
        }
        if(current == null)  {//si no tiene hijo
            return null;  //no lo encontró
        }
    } //fin while
    return current;   // lo encontró

}  // fin encontrar
```

Árbol binario completamente balanceado

Un árbol es perfectamente balanceado si para cada nodo, la cantidad de nodos en sus subárboles izquierdo y derecho difiere en uno como máximo.

Ejemplos de árboles perfectamente balanceados



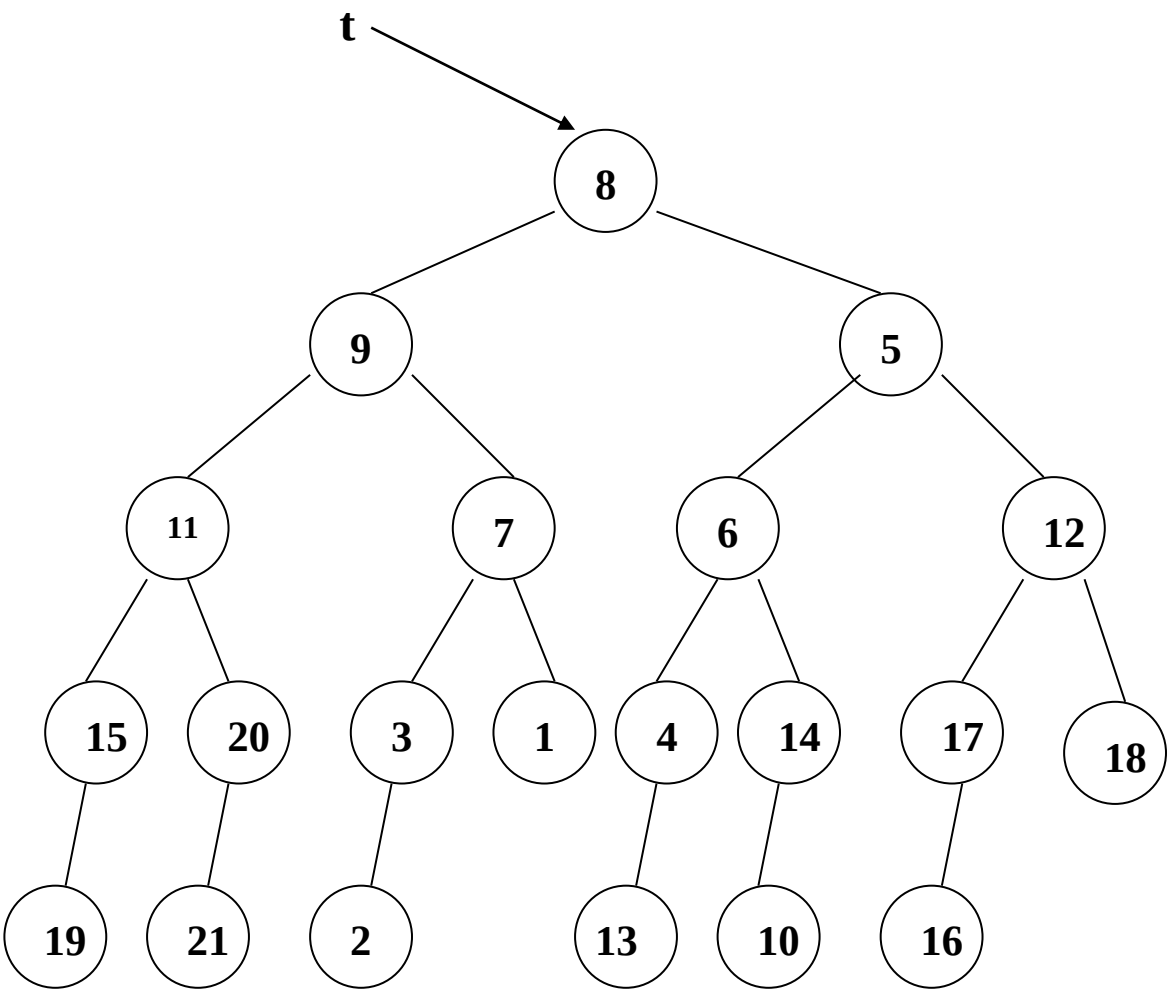
Ejemplo:
Se desea construir un árbol de n nodos, perfectamente balanceado.

```
private Nodo arbol(int n) { //Construye arbol de n nodos

  Nodo nodo;
  int x;
  int NL;
  int NR;

  if (n == 0)
       return null;

  else {
       NL = n/2;
       NR = n – NL - 1;
       x = StdIn.readInt();
       nodo = new Nodo();
       nodo.setDato(x);
       nodo.setHijoIzquierdo(arbol(NL));
       nodo.setHijoDerecho(arbol(NR));
       return nodo;
  }

}

public Arbol(int n) {
       raiz = arbol(n);
}
```

N = 21

Datos: 8 – 9 – 11 – 15 – 19 – 20 – 21 – 7 – 3 – 2 – 1 – 5 – 6 – 4 – 13 – 14 – 10 – 12 – 17 – 16 – 18

Árbol resultante de insertar cada uno de los 21 nodos



4.2 Dividir para Conquistar

Concepto

Método general:
Esta estrategia consiste en **DIVIDIR** un problema en subproblemas, **encontrar la solución a cada subproblema**, y luego **combinar** las soluciones parciales, de forma de resolver el problema original.

Es decir:

```
Resolver (P)
    Descomponer P en las subproblemas P1, P2, ..., Pk.
    Resolver (Pi), para i= 1, 2, ..., k.
    Encontrar la solución para P en base a la solución de los Pi.
End Resolver.
```

A menudo, los subproblemas resultante en la aplicación de la estrategia, son del mismo tipo que el problema original. En estos casos, la reaplicación del principio de dividir para conquistar se expresa en forma natural mediante un procedimiento recursivo.

Los subproblemas que se generan son cada vez más pequeños de forma que en un momento determinado, pueden ser resueltos directamente.

Ejemplos:

- Búsqueda del mayor y menor

- Algoritmos de ordenamiento:
  - MERGESORT.
  - QUICKSORT.

Ejercicios

a) Búsqueda del mayor y/o del menor

Mayor de un conjunto S
Encontrar el valor mayor de un conjunto S, el que contiene N elementos.

A continuación, se muestra el pseudocódigo que resuelve el problema planteado.

```
function max (S)
begin
    //Regresa el valor mayor del conjunto S
    if || S || = 2
        return ("valor mayor de los 2 elementos")
    else
        "Dividir S en 2 mitades: S1 y S2"
        max1 ← Max (S1);
        max2 ← Max (S2);
        return max(max1, max2)
    endif
end Max
```

En Java:

```
main() {
    …
    Arreglo arreglo = new Arreglo(50);
    StdOut.println(El mayor es: " + arreglo.max());
}
```

```
public class Arreglo {
    ...
    public int max(){
        return(max(0, cantidadElementos-1));
    }
    private int max(int posicionInicial,int posicionFinal){
        int mayor;
        if (posicionInicial == posicionFinal) {
        //un solo elemento
            mayor = arreglo [posicionInicial];
            return mayor;
        }
        else {
            if (posicionInicial == posicionFinal -1)   {
            //Dos elementos
                if (arreglo[posicionInicial] >
                                    arreglo[posicionFinal]){
                    mayor = arreglo[posicionInicial];
                }
                else{
                    mayor = arreglo[posicionFinal];
                }
                return mayor;
            }
            else {
                int mitad = (posicionInicial +
                                        posicionFinal)/ 2;
                int mayor1 = max (posicionInicial, mitad);
                int mayor2 = max (mitad +1, posicionFinal);
                if(mayor1 > mayor2){
                    return mayor1;
                }
                else{
                    return mayor2;
                }
            }
        }
    }//Fin max
} //Fin class Arreglo
```

Mayor y menor de un conjunto S

Encontrar el valor mayor y el valor menor de un conjunto S, el que contiene N elementos.

A continuación, se muestra el pseudocódigo que resuelve el problema planteado.

```
Procedure MaxMin (S)
begin

//Regresa un par de valores (a, b), donde a es el valor mayor y b
//es el valor menor en el conjunto S.

  if  || S ||  = 2 then //Cantidad de elementos del arreglo igual a 2
      Sea S = {a, b}
      return (Max (a, b), Min (a, b))

  else
     Dividir S en dos mitades S1, S2
     (max1, min1) ← MaxMin (S1)
     (max2, min2) ← MaxMin(S2)
     return( Max (max1, max2), Min ( min1, min2))
  end if
end MaxMin
```

```
Procedure MaxMin ( i, j; var: mayor, menor)
begin
# Datos en arreglo global  A(1..N)
# i, j: Posición inicial y final del arreglo analizado
# mayor, menor: Valores mayor y menor en el arreglo.
# A(k), k = i, j
  if (i = j) then
        mayor ← A(i)
        menor ← A(i)
  else
    if (i = j - 1) then
      if (A(i) < A (j)) then
          mayor ← A(j)
          menor ← A(i)
       else
          mayor ← A (i)
          menor ← A(j)
      end if
    else
      mitad ← ⌊(i + j)/2⌋
      MaxMin (i, mitad, MAY1, MEN1)
      MaxMin (mitad +1, j, MAY2, MEN2)
      menor ← Min ( MEN1 , MEN2)
      mayor ← Max (MAY1, MAY2)
    end if
  end if
end MaxMin
```

En el programa principal:

MAXMIN (1, N, VALMAYOR, VALMENOR).

Ejercicio: Codificar el pseudocódigo anterior en Java.

Ordenamiento por mezcla: MERGESORT.

Ordenar la secuencia de valores $X_1,X_2,...,X_n$.

Solución:

i. Dividir la secuencia en dos conjuntos:

   $A_1,A_2,...,A_{\lfloor n/2 \rfloor}$

   $A_{\lfloor n/2 \rfloor + 1}, ..., A_n$

ii. Cada conjunto es ordenado en forma independiente y las secuencias resultantes se mezclan para producir una secuencia única de N elementos.

Para ordenar un arreglo:

- Si el arreglo tiene un solo elemento, PARAR.
- En caso contrario:
  i. Ordenar (la primera mitad).
  ii. Ordenar (la segunda mitad)
  iii. Combinar las dos mitades.

// Muestra el Mergesort en forma recursiva

```
public class ArregloDobles {

    private double[ ] arreglo;
    // arreglo referencia al arreglo
    private int cantidadElementos;
    // cantidad de items de datos

    public ArregloDobles(int max)    {// constructor
        arreglo = new double[max]; // crea el arreglo
        cantidadElementos = 0;
    }

    public int getCantidadElementos(){
        return cantidadElementos;
    }

    public double getElemI(int i){
        return arreglo[i];
    }

    public void insert(double value) {
    // coloca un elemento en el arreglo
        arreglo[cantidadElementos] = value; //lo inserta
        cantidadElementos++; // incrementa el tamaño
    }

    public void mergeSort()   {
        // llamado por main()
        // se requiere espacio de trabajo
        double[]workSpace=new double[cantidadElementos];

        recMergeSort(workSpace, 0, cantidadElementos-1);
    }
```

Se debe chequear i

Se debe chequear que quede espacio

```
private void recMergeSort(double[] workSpace,
                          int lowerBound, int upperBound){

    if(lowerBound == upperBound) {
    // si el rango es 1
        return;
        // no usa el ordenamiento.
        //Sale del procedimiento
    }
    else {
        // encuentra el punto medio
        int mid = (lowerBound+upperBound) / 2;

        // ordena la mitad de abajo
        recMergeSort(workSpace, lowerBound, mid);

        // ordena la mitad de arriba
        recMergeSort(workSpace, mid+1, upperBound);

        // mezcla las dos mitades ordenadas
        merge(workSpace,lowerBound,mid+1, upperBound);
    }  // fin else

}  // end recMergeSort()
```

```
    private void merge(double[] workSpace, int lowPtr,
                                int highPtr, int upperBound){

        int j = 0;  // indice del espacio de trabajo
        int lowerBound = lowPtr;
        int mid = highPtr-1;
        int n = upperBound - lowerBound + 1;
        //número de items

        while(lowPtr <= mid && highPtr <= upperBound) {
            if( arreglo[lowPtr] < arreglo[highPtr] ) {
                workSpace[j++] = arreglo[lowPtr++];
            }
            else{
                workSpace[j++] = arreglo[highPtr++];
            }
        } //Fin while

        while(lowPtr <= mid)    {
            workSpace[j++] =  arreglo[lowPtr++];
        }

        while(highPtr <= upperBound) {
            workSpace[j++] = arreglo[highPtr++];
        }

        for(j=0; j<n; j++) {
            arreglo[lowerBound+j] = workSpace[j];
        }

    }  // end merge

}  // end class ArregloDobles
```

```
import ucn.StdOut;

public class MergeSortApp {

    public static void desplegarArreglo(ArregloDobles arreglo){
        //despliega el contenido del arreglo
        for(int j=0;j<arreglo.getCantidadElementos();j++){
            //despliega el elemento
            StdOut.print(arreglo.getElemI(j) + " ");
        }
        StdOut.println("");
    }

    public static void main(String[] args) {
        int maxSize = 100; // tamaño del arreglo
        // crea el arreglo
        ArregloDobles arreglo = new ArregloDobles(maxSize);

        // inserta items
        arreglo.insert(64);
        arreglo.insert(21);
        arreglo.insert(33);
        arreglo.insert(70);
        arreglo.insert(12);
        arreglo.insert(85);
        arreglo.insert(44);
        arreglo.insert(3);
        arreglo.insert(99);
        arreglo.insert(0);
        arreglo.insert(108);
        arreglo.insert(36);

        MergeSortApp.desplegarArreglo(arreglo);//despliega

        arreglo.mergeSort(); // ordena el arreglo

        MergeSortApp. desplegarArreglo(arreglo);
        // despliega los ítems

    }// fin main
}
```

Notas:

- El algoritmo requiere de nuevos llamados hasta que el conjunto que se ordena tenga un solo elemento.

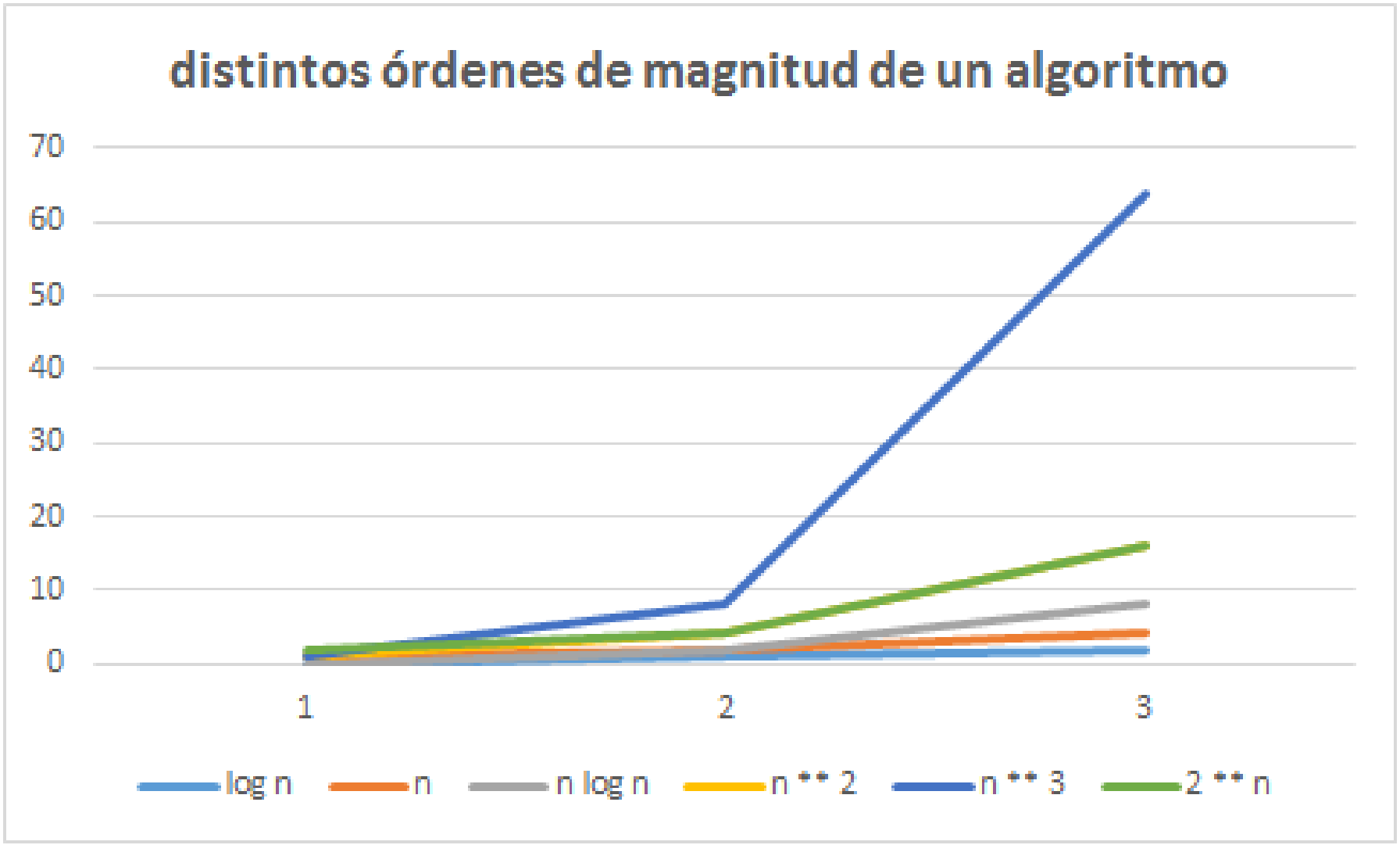
- En el caso de un conjunto de tamaño pequeño, la mayor parte del tiempo del algoritmo, se ocupa en llevar a cabo la recursión en vez de ordenar los datos.

- Esta situación se mejora terminando la recursión después de un cierto valor en el tamaño del conjunto que se quiere ordenar.

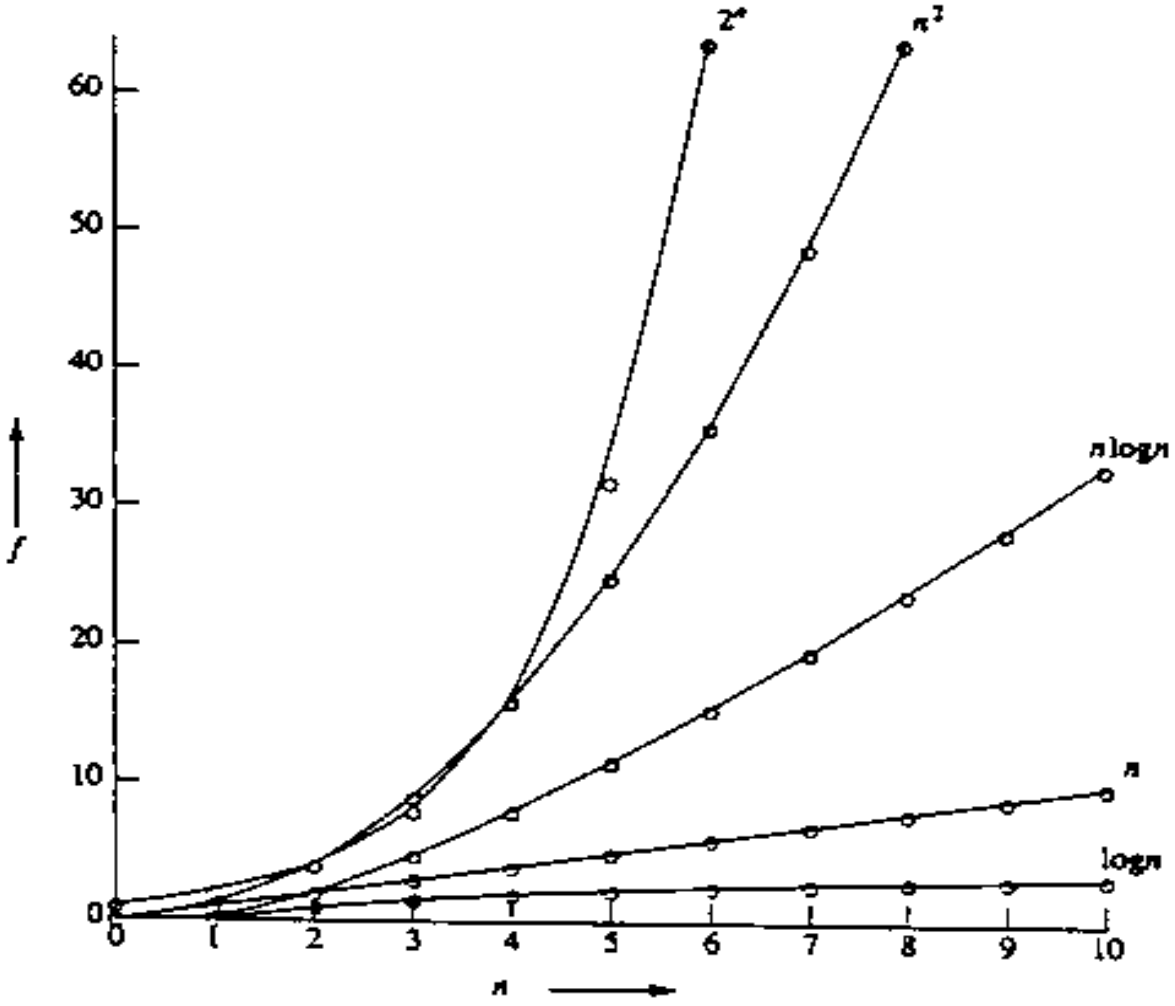
- El ordenamiento de un conjunto pequeño de datos, se puede efectuar utilizando un segundo algoritmo, que trabaje bien en el caso de conjuntos pequeños. (Usar por ejemplo, ordenamiento por inserción).

Una versión revisada del Mergesort, que considera las observaciones anteriores, se muestra a continuación.

```
private void recMergeSortRevisado(double[] workSpace,
                          int lowerBound, int upperBound) {

    if ((upperBound-lowerBound+1) < 16)//pocos datos
        ordenamientoSimple(lowerBound,upperBound);
    else {
        int mid = (lowerBound+upperBound)/2;
        //encuentra la mitad

        recMergeSortRevisado(workSpace,lowerBound,mid);
        //ordena la mitad inferior

        recMergeSortRevisado(workSpace,mid+1,upperBound);
        //ordena la mitad superior

        merge(workSpace,lowerBound,mid+1,upperBound);
    } //fin else

}//fin recMergeSortRevisado



private void ordenamientoSimple(int lowerBound,
                                          int upperBound){

    for(int i = lowerBound; i<= upperBound-1; i++){
        for(int j = i+1; j <= upperBound; j++){
            if (arreglo[i]> arreglo[j]){
                double aux= arreglo[i];
                arreglo[i]= arreglo[j];
                arreglo[j]= aux;
            }
        }
    }

}//Fin ordenamientoSimple
```


distintos órdenes de magnitud de un algoritmo
70
60
50
40
30
20
10
0
1
2
3
log n
n
n log n
n ** 2
n ** 3
2 ** n

Gráfica de los distintos órdenes de magnitud de un algoritmo



Valores de f(N), dependiendo del valor de N

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Tiempo de ejecución, dependiendo del orden de magnitud

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^7$yr	$3.17*10^{43}$yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

Nota: El orden de magnitud del algoritmo de las torres de Hanoi es $O(2^n)$

4.3 Backtracking

Concepto

- En la búsqueda por principios fundamentales para el diseño de algoritmos, el **backtracking** representa una de las técnicas más generales.

- La tarea es determinar algoritmos para encontrar soluciones a problemas específicos, no siguiendo una regla fija de cálculo, sino mediante prueba y error.

- El patrón común es descomponer el proceso de prueba y error en tareas parciales.

- A menudo, estas tareas son expresadas más naturalmente en términos recursivos y consisten de la exploración de un número finito de subtareas.

- Generalmente, se puede ver el proceso completo, como un proceso de búsqueda, que gradualmente construye y examina un árbol de subtareas.

- En muchos problemas, este árbol de búsqueda crece muy rápidamente, usualmente exponencialmente, dependiendo de un parámetro dado.

Patrón General

```
Procedure Try;
begin
    Inicializar selección de los candidatos;
    repeat
        Seleccionar el siguiente;
        if Aceptable then
        begin
            Registrar la selección;
            if (solución incompleta) then
            begin
                Try paso siguiente;
                if no – éxito then
                    Cancelar el registro
            end
        end
    until éxito or no hay más candidatos
end
```

Un patrón frecuentemente encontrado, utiliza un parámetro de nivel explícito, el que indica la profundidad de la recursión, y que permite una condición de término simple.

Más aún, si en cada etapa, la cantidad de candidatos a ser investigada es fija, por ejemplo m, entonces se puede aplicar el siguiente esquema.

Invocación del esquema se hace mediante: Try (1)

```
Procedure Try (i: integer);
   var K: integer;

begin
   K: = 0;
repeat
        K: = K + 1
        Selección del K – ésimo candidato;
        if Aceptable then
      begin
         Registrar la selección;
         if (i < n) then
         begin
            TRY (i + 1);
            if (no – éxito) then
                  Cancelar el registro
         end
      end
until éxito or (K = m)

end
```

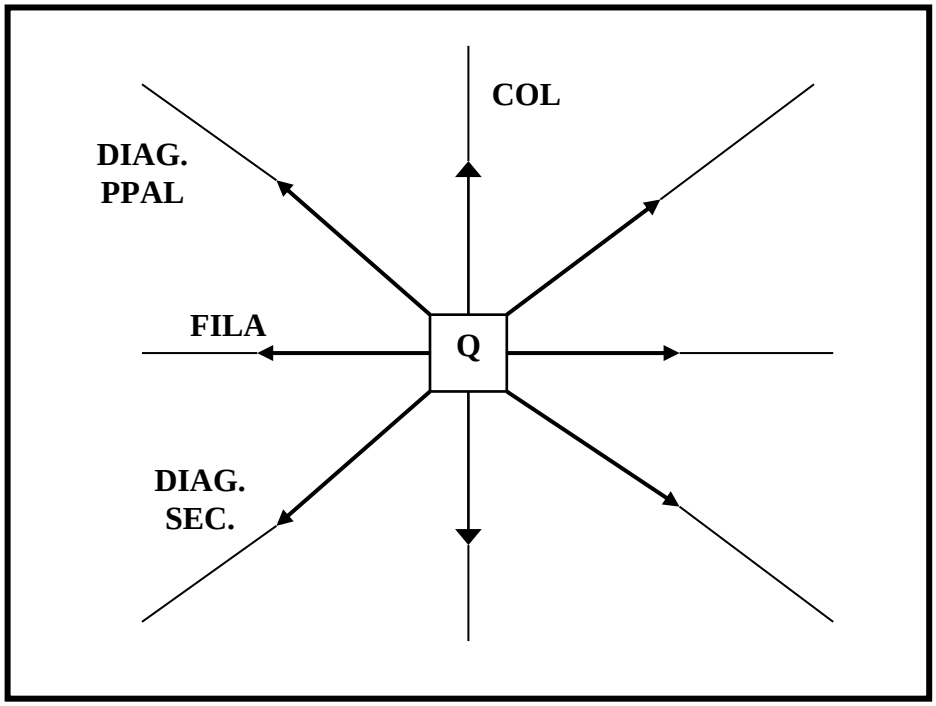
Ejercicios

a) El problema de las ocho reinas

El problema de las ocho reinas, consiste en colocar ocho reinas sobre un tablero de ajedrez, sin que se ataquen mutuamente.

Dos reinas se atacan mutuamente si están colocadas en la misma fila, en la misma columna o en la misma diagonal del tablero.

Trayectoria de una reina



Se debe diseñar un programa que encuentre una solución para el problema planteado, si es que existe una solución, o diga que no hay ninguna, en caso contrario.

Solución:

Es necesario considerar que cada fila del tablero de ajedrez debe contener una y sólo una reina.

Esto sugiere que uno de los caminos para encontrar una solución, consiste en desarrollar una forma sistemática de colocar las ocho reinas, teniendo la primera en algún casillero de la fila 1, la segunda en la fila 2, la tercera en la fila 3, etc.

Una forma de poner K reinas sobre las K primeras filas del tablero ($0 \leq K < 8$) sin que se ataquen mutuamente, se considera una aproximación a la solución.

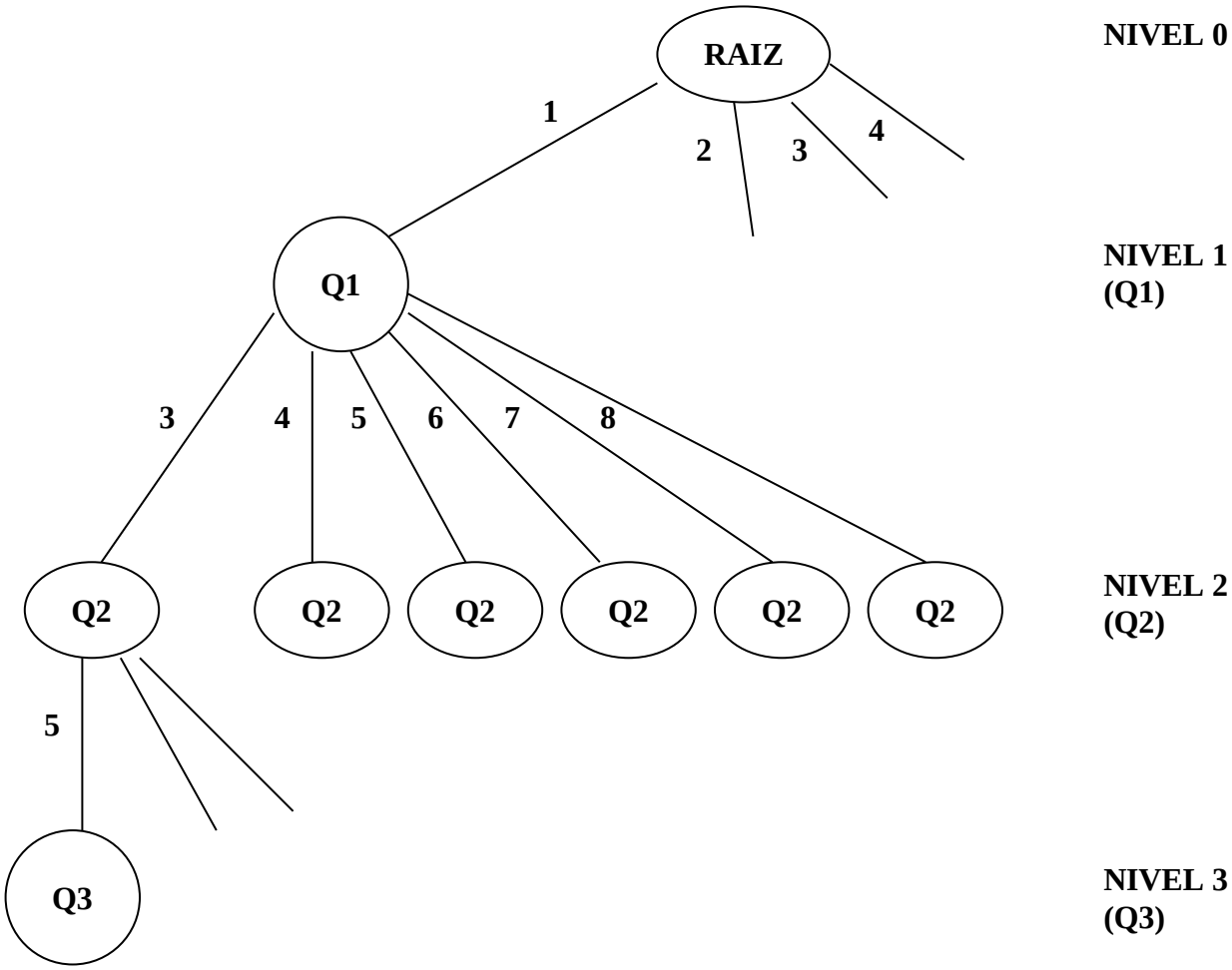
Por ejemplo: Q1 (6,2) Q2 (4,5)
Q3 (3,3) Q4 (7,6)

Cuatro reinas colocadas en el tablero

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1				**X**				
2								**X**
3			**Q3**					
4					**Q2**			
5							**X**	
6		**Q1**						
7						**Q4**		
8								

El estudio de este problema se debe acreditar a NICKLAUS WIRTH, en su artículo: “Program Development by Stepwise Refinement”
Communications of the ACM
Vol.14, Nº 4, abril 1971,
Pag. 221 – 226

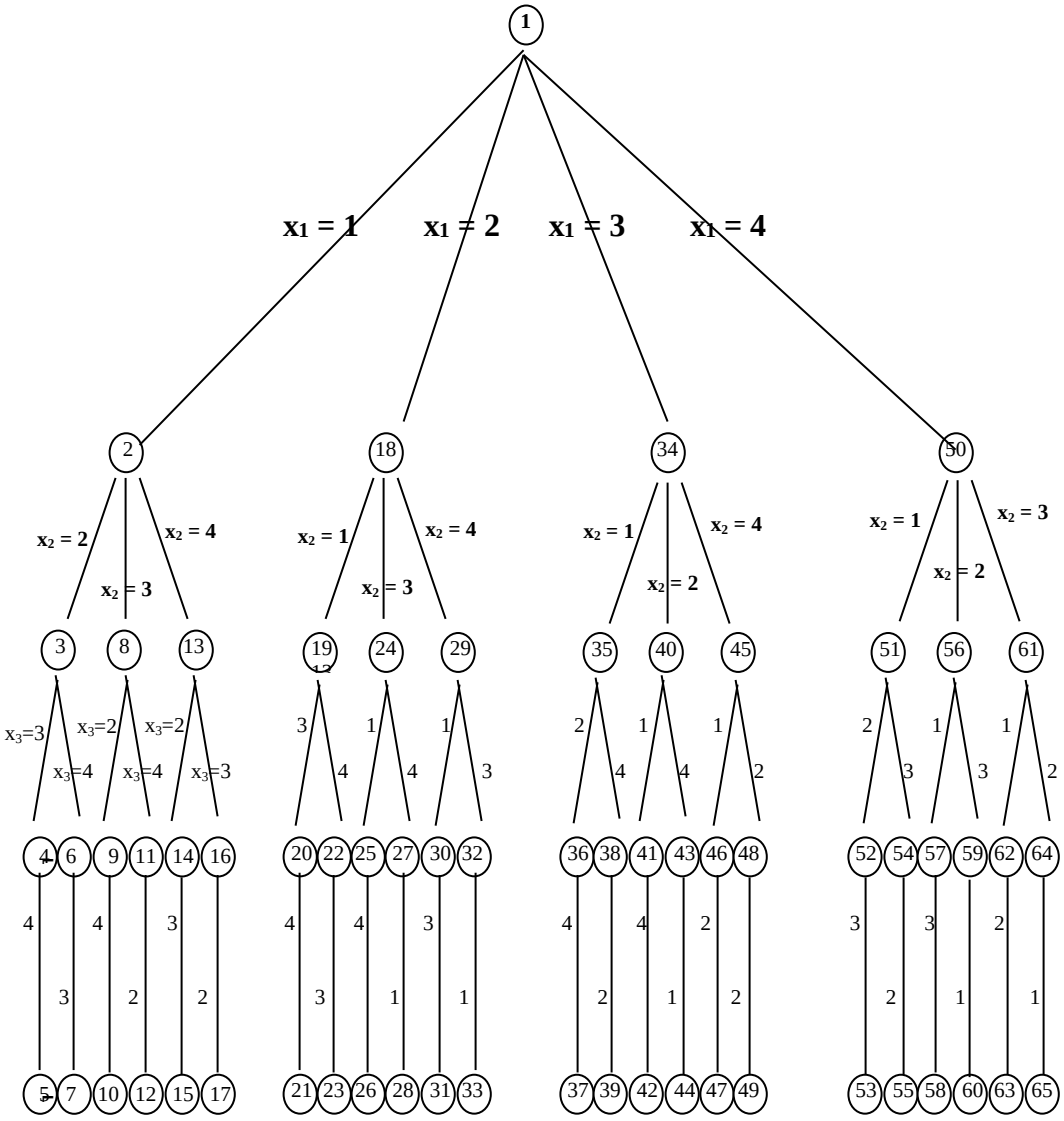
Árbol de decisiones para el problema de las 8 reinas



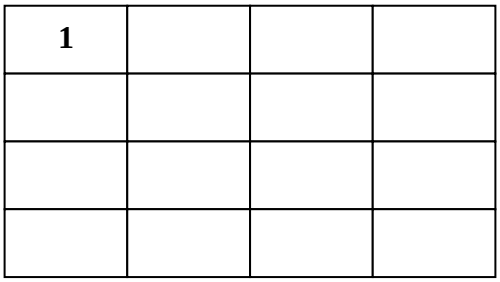
	1	2	3	4	5	6	7	8
1	Q1							
2			Q2					
3					Q3			
4		X					X	X
5								

Árbol completo para el problema de las 4 reinas

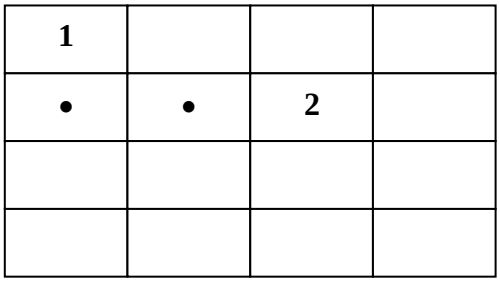
- 4 reinas en un tablero de 4 x 4.
- Los nodos están numerados de acuerdo a la búsqueda en profundidad.



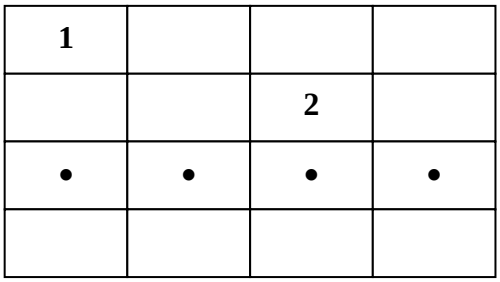
Ejemplo de backtracking para el problema de 4 reinas



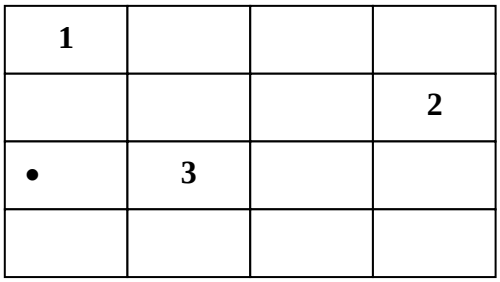
(a)



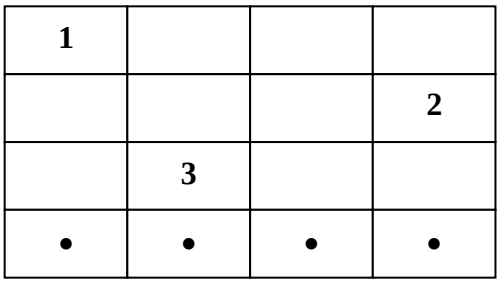
(b)



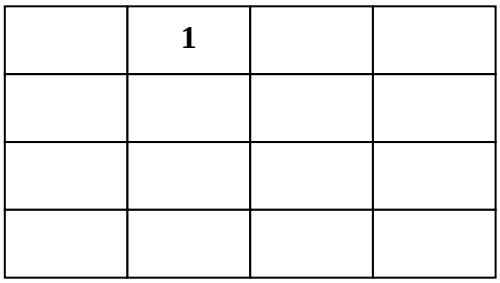
(c)



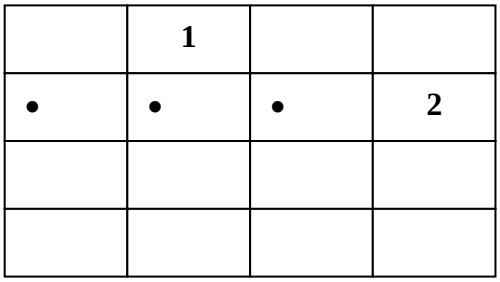
(d)



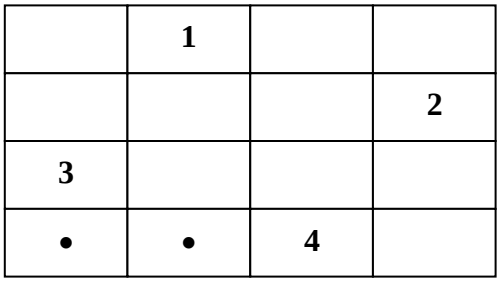
(e)



(f)

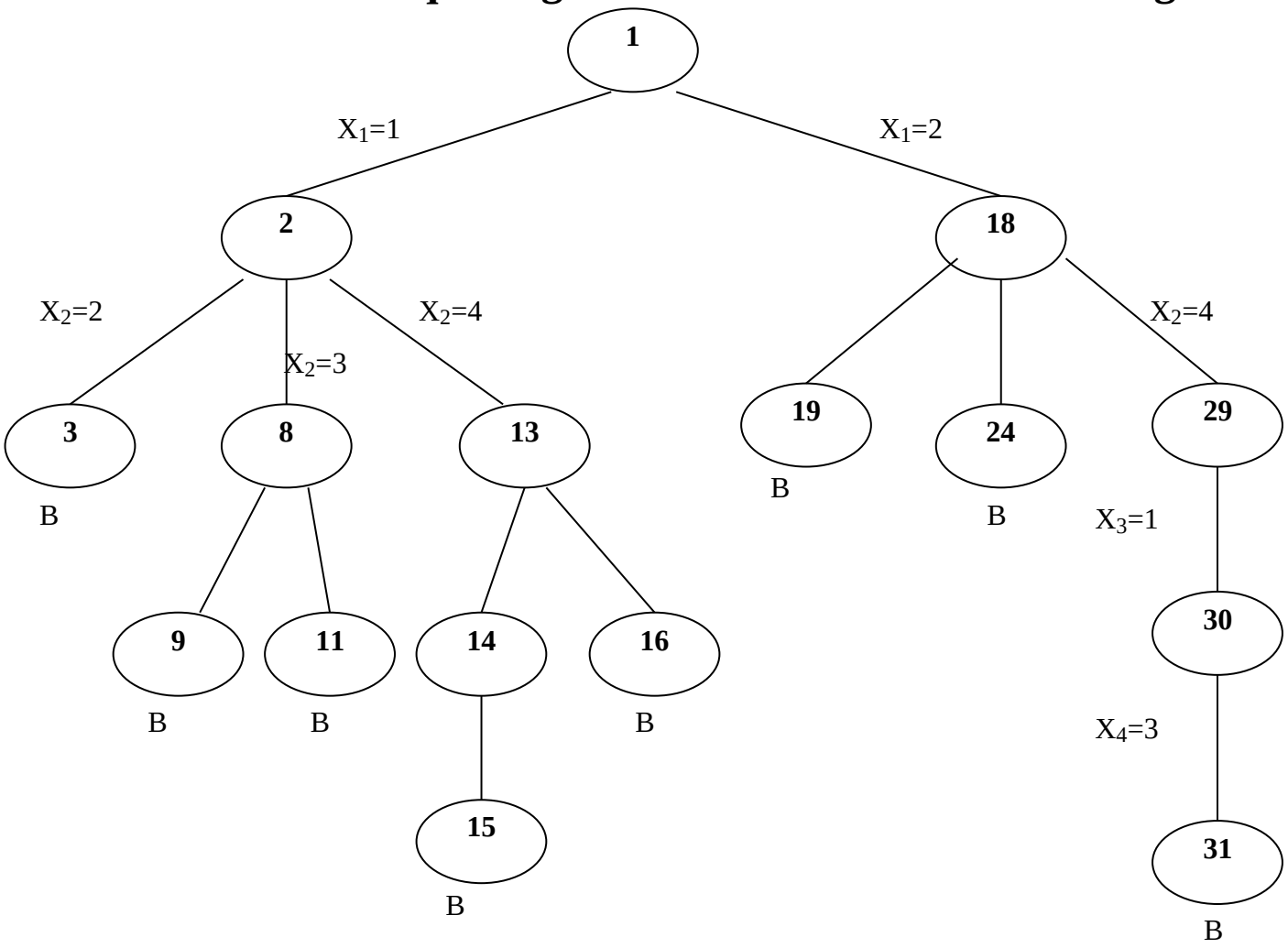


(g)



(h)

Parte del árbol que es generado durante el backtracking



Una solución al problema de las 8 reinas

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

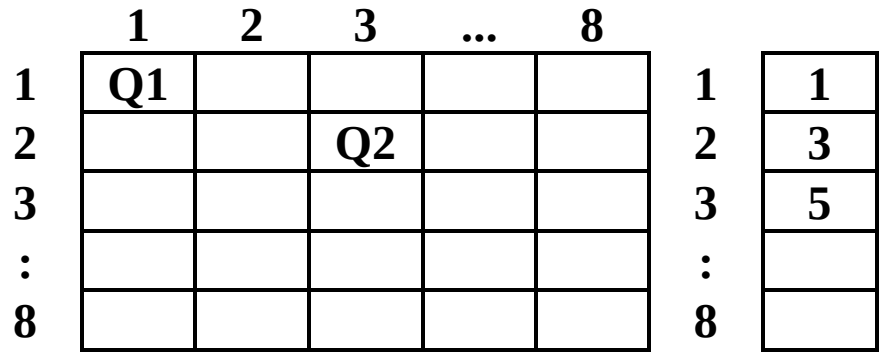
No es necesario representar el tablero de ajedrez usando un arreglo de 8 x 8 elementos. Es suficiente utilizar un solo vector de 8 elementos.

Por ejemplo:

COL (I), I = 1,8

donde:

COL (i), contiene el número de la columna en que está ubicada la i – ésima reina.



¿Cómo determinar si la reina en la posición (F1,C1) ataca a la reina en la posición (F2,C2)?

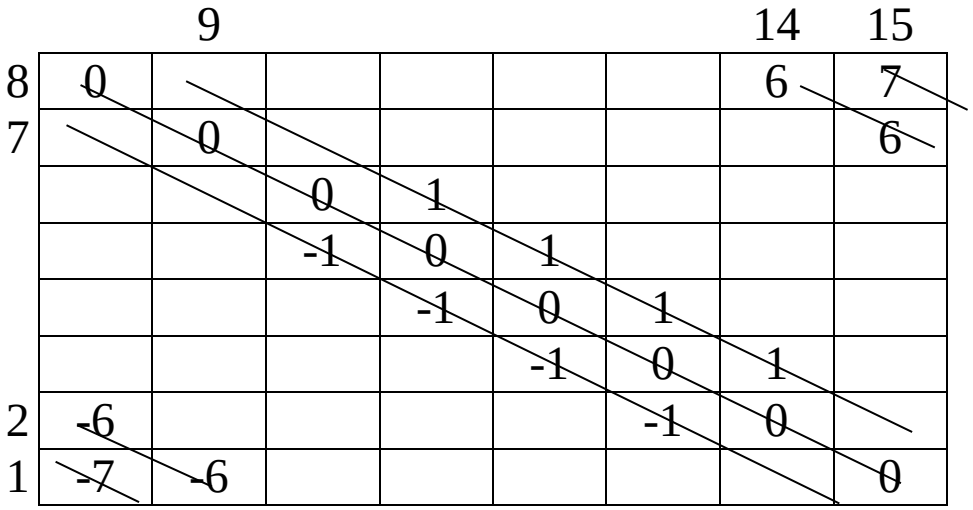
Considerar 3 pruebas:

a) Caso en que las dos reinas se encuentran en la misma columna.

¿C1 = C2?

b) Caso en que la reina en la posición (F1,C1), se encuentra en la diagonal principal que pasa por la posición (F2,C2).

Diagonales principales en el arreglo de 8x8



Se cumple que la diferencia de los subíndices para los elementos ubicados en la diagonal principal es una constante (columna - fila).

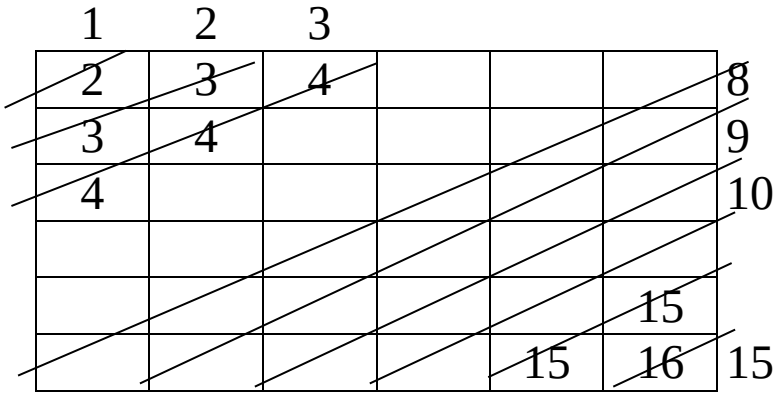
Por ejemplo, en la diagonal 3, la diferencia entre los subíndices de columna y fila para cada casillero es –5.

(6,1) 1 – 6 = -5
(7,2) 2 – 7 = -5
(8,3) 3 – 8 = -5

¿(C1 – F1) = (C2 – F2)?

c) Caso en que la reina en la posición (F1,C1), se encuentra en la diagonal secundaria, que pasa por la posición (F2,C2).

Diagonales secundarias en el arreglo de 8x8



Se cumple que la suma de las subíndices para los elementos de cualquiera de las diagonales secundarias es constante (columna + fila).

¿(C1 + F1) = (C2 + F2)?

Primer refinamiento

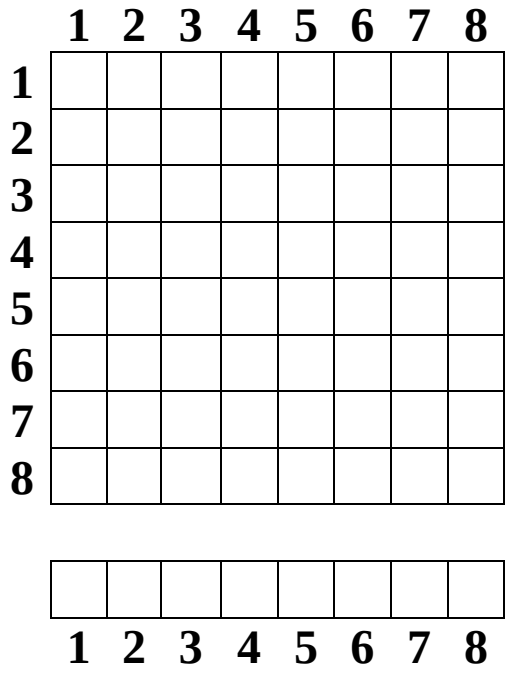
```
Procedure TRY  (i: integer)
begin
    "Inicializar la selección de la posición de la i-ésima reina"
    repeat
        "Seleccionar la posición";
        if "posición a salvo" then
        begin
            "Ubicar la reina"
            if (i < 8) then
            begin
                TRY (i + 1);
                if "intento fracasa" then
                    "Remover Reina"
            end
        end
    until  "Ubicación exitosa " or " No existen más posiciones "
end
```

Segundo Refinamiento (Transformado a función)

```
boolean Try (integer i) {
        boolean q;
        "Inicializa la selección de la posición de la i-ésima reina (K = 0)"
        repeat {
                "Selecciona la posición" (K = K + 1)"
                q = false;
                if ("posición a salvo") {
                        "Ubicar la reina"
                        if  (i < 8) {
                                q = Try (i+1)
                                if (¬ q) {
                                        "Remover reina"
                                }
                        }
                        else {
                                q = true;
                        }
                }
        }
        until (q or K = 8)
        return q
}
```

Matriz de 8x8 y su equivalente vector de columnas de 8 posiciones para el problema de las 8 reinas

$solucion_i$ = Contiene la posición de la columna de la i-ésima reina



Solución:

Estructuras

solucion, array [1 .. 8] of integer;
columnas, array [1 .. 8] of boolean
diagonalesSecundarias, array [2 .. 16] of boolean;
diagonalesPrincipales, array [-7 .. 7] of boolean;

- **solucion[i]** Denota la columna donde se encuentra la i-ésima reina.
- **columnas[j]** Indica que ninguna reina se encuentra en la columna j – ésima.
- **diagonalesSecundarias[K]** Indica que ninguna reina ataca la diagonal secundaria K- ésima.
- **diagonalesPrincipales[K]** Indica que ninguna reina ataca la diagonal principal K- ésima.

Ubicar la reina:

solucion[i]: = j

columnas[j]: = false

diagonalesSecundarias[i+j]: = false

diagonalesPrincipales[i-j]:= false

Remover la Reina:

columnas [j]: = true

diagonalesSecundarias [i+j]: = true

diagonalesPrincipales [i-j]:= true

Posición a salvo:

columnas [j] ^ diagonalesSecundarias [i+j] ^ diagonalesPrincipales [i-j]

//Juego de las ocho reinas. Encuentra una solución

Las estructuras están definidas como estáticas (variables globales), de manera de no tener que trabajar con parámetros. Esto por el hecho que aquí lo importante es el backtracking

```
import ucn.StdOut;

public class Reinas {

    static boolean[] columnas = new boolean[9];
    static boolean[] diagonalesSecundarias =
                                        new boolean[17];
    static boolean[] diagonalesPrincipales =
                                        new boolean[15];
    static int[] solucion = new int[9];


    public static void main(String[] args) {

        boolean respuesta;
        int i;
        for(i=1; i<9 ; i++) columnas[i]=true;
        for(i=2; i<17; i++) diagonalesSecundarias[i]=true;
        for(i=0; i<15; i++) diagonalesPrincipales[i]=true;

        respuesta = Reinas.intenta(1);

        if(respuesta)
            for(i=1; i<9 ;i++){
                StdOut.println(solucion[i]);
        }
        else{
            StdOut.println("Problema sin solución");
        }

    }//fin del main
```

```
    public static boolean intenta(int i)    {
        int j;
        boolean q;
        j=0;
        do {
            j = j+1;
            q = false;

           if(columnas[j] & diagonalesSecundarias[i+j] &
                             diagonalesPrincipales[i-j+7]){
               solucion[i] = j;
               columnas[j] = false;
               diagonalesSecundarias[i+j] = false;
               diagonalesPrincipales[i-j+7] = false;

               if(i<8) {
                   q = intenta(i+1);
                   if(!q) {
                       columnas[j] = true;
                       diagonalesSecundarias[i+j] =true;
                       diagonalesPrincipales[i-j+7]=true;
                   }
               }
               else{
                   q = true;
               }
           };
        }while(!q && j !=8);

        return q;

    }//fin del método intenta

}//fin clase Reinas
```

Solución: 1 - 5 - 8 - 6 -3 - 7- 2 - 4

Fila 1 Fila 3 Fila 8

Otra solución para el problema de las 8 reinas

	1	2	3	4	5	6	7	8
1	X							
2							X	
3					X			
4								X
5		X						
6				X				
7						X		
8			X					

X = (1 , 7 , 5 , 8 , 2 , 4 , 6 , 3)
Fila 1 Fila 3 Fila 8

Todas las soluciones al problema de las ocho reinas

```
Procedure Try (i: integer);
var K: integer;
begin
    for K:  = 1 to 8 do
    begin
        Selección del K- ésimo candidato;
        if Aceptable then
        begin
            Registrar la selección
            if i < n then
                Try (i + 1)
            else
                Imprimir la solución
            endif
            Cancelar el registro
        end
    end for
end Try
```

// 8 reinas: Todas las soluciones

```
import ucn.StdOut;

public class Reinas {
    static boolean[] columnas = new boolean[9];
    static boolean[] diagonalesSecundarias = new boolean[17];
    static boolean[] diagonalesPrincipales = new boolean[15];
    static int[] solucion = new int[9];

    public static void main(String[] args) {
        int i;
        for(i=1; i<9 ; i++) columnas[i]=true;
        for(i=2; i<17; i++) diagonalesSecundarias[i]=true;
        for(i=0; i<15; i++) diagonalesPrincipales[i]=true;

        intentaTodasLasSoluciones(1);
    }

    public static void intentaTodasLasSoluciones (int i){
        for(int j=1; j<9; j++)  {
            if(columnas[j] & diagonalesSecundarias[i+j] &
                             diagonalesPrincipales[i-j+7]){
                solucion[i] = j;
                columnas[j] = false;
                diagonalesSecundarias[i+j] = false;
                diagonalesPrincipales[i-j+7] = false;
                if(i<8){
                    intentaTodasLasSoluciones(i+1);
                }
                else {
                    for(int k=1; k<9; k++){
                        StdOut.println(solucion[k] + "  ");
                    }
                }
                columnas[j] = true;
                diagonalesSecundarias[i+j] = true;
                diagonalesPrincipales[i-j+7] = true;
            }
        }
    }
}//fin clase Reinas
```

¿Cómo modelar un problema de backtracking?

a. ¿Qué representa la raíz del árbol?

b. ¿Cuáles son los potenciales candidatos?

c. Dibuje el árbol de búsqueda (2 niveles además de la raíz)

d. ¿Cuántos niveles tendrá el árbol de búsqueda?

e. ¿Qué representa cada nivel del árbol?

f. ¿Qué estructuras de datos necesita para el problema? ¿Para qué sería cada una de ellas?

g. ¿Qué significa ser aceptable?

h. ¿Qué significa registrar la selección?

i. ¿Qué significa cancelar la selección?

j. ¿Qué significa solución incompleta?

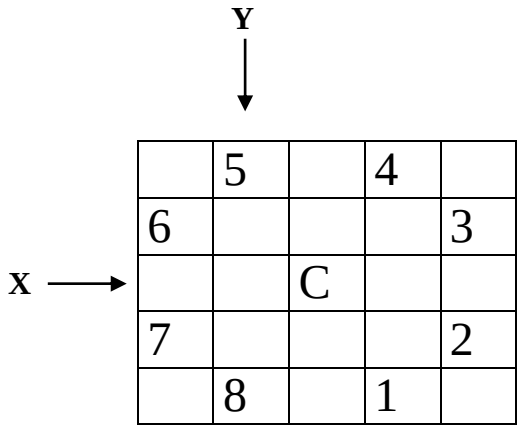
Nota: Para (g), (h) e (i), explique además como responderá la pregunta considerando las estructuras de (f)

b) Recorrido de un caballo en el tablero de ajedrez.

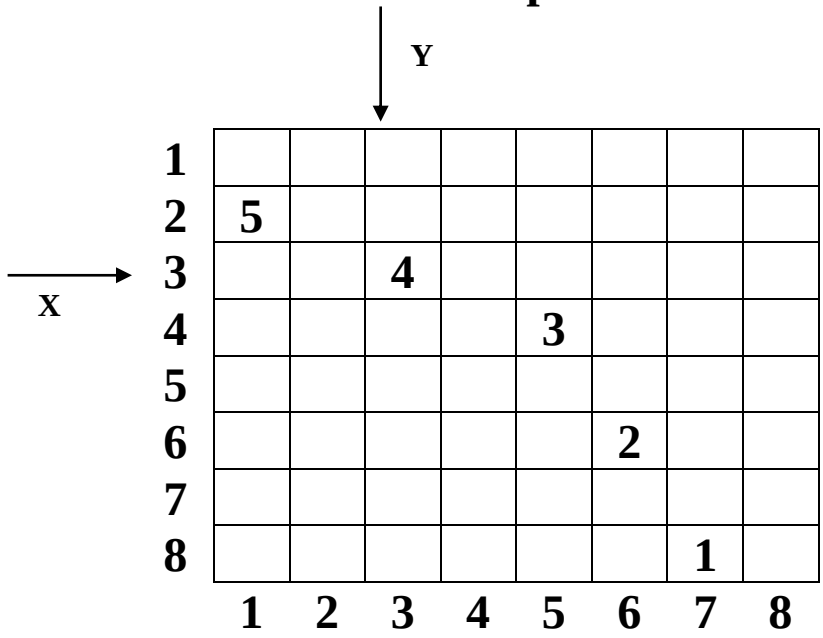
Se dispone de un tablero de tamaño n x n. Un caballo, al que se le permite moverse de acuerdo a las reglas del ajedrez, es ubicado en el tablero, en la posición (x_0,y_0).

Se requiere diseñar un algoritmo que permite al caballo efectuar un recorrido completo del tablero de modo que cada casillero sea visitado exactamente una vez.

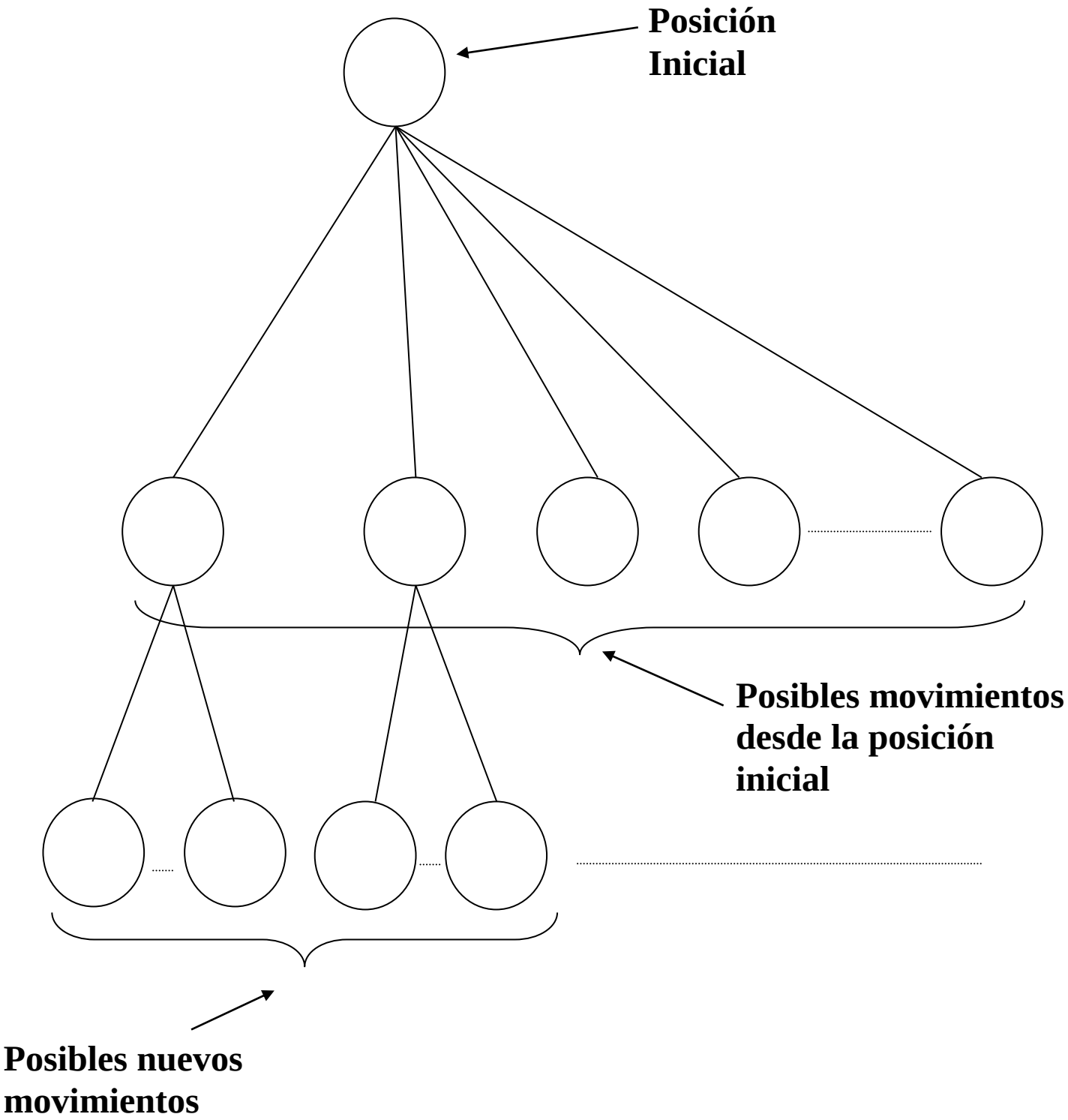
Movimientos posibles de un caballo



Posibles movimientos del caballo a partir de la ubicación inicial (8,7)



Árbol para el problema del recorrido del caballo



Primer refinamiento

Procedure "intentar próximo movimiento"

```
begin
   "inicializar selección del movimiento"
   repeat
         "Seleccionar el posible movimiento desde una lista de siguientes
          movimientos"
         if("movimiento aceptable") then
         begin
               "Registrar el movimiento"
               if("tablero no está lleno") then
               begin
                     "Intentar próximo movimiento"
                     if("intento no exitoso") then
                           "Borrar movimiento anterior"
               end
         end
   until("Movimiento fue exitoso" or "no existen posibles movimientos")
end
```

Una descripción más precisa del algoritmo, hace necesario decidir acerca de la representación de los datos.

Tablero(i, j) i = 1,N
               j = 1,N

donde:

Tablero(x,y) =0 Indica que la posición (x, y) no ha sido visitada.

Tablero (x,y) = i Indica que la posición (x, y) ha sido visitado en el i- ésimo movimiento ($1 \leq i \leq n^2$).

Tablero: array [9][9] of integer. Primera fila (0) y primera columna (0) no se usarán. Suponiendo un tablero de 8 x 8

Elección de parámetros para el procedimiento:

- Condiciones de partida para el movimiento:

  Se debe especificar el número del movimiento que se efectúa (i) y las coordenadas (x, y) desde las que se realiza el movimiento.

- Informar el resultado del movimiento:

  Utilizar la variable booleana q donde:
  - q = true, indica ÉXITO.
  - q = false, indica FRACASO.
- n x n: Tamaño del tablero de ajedrez con el que se trabajará

Refinamiento dos

boolean Try (int i, x, y);
q: boolean:

```
begin
  "Inicializar selección del movimiento"
  repeat
     q = false;
     "sean nuevaX, nuevaY , las coordenadas del siguiente movimiento
                                          definido por las reglas del ajedrez"
     if (1 ≤ nuevaX ≤ n) ∧ (1 ≤ nuevaY ≤ n) ∧
                                 Tablero[nuevaX, nuevaY] =0) {//Es aceptable
        Tablero[nuevaX, nuevaY] := i;
        if (i < n^2 ) {
              q = try (i +1, nuevaX, nuevaY );
              if ( ! q )
                    Tablero [nuevaX, nuevaY] : =0
        }
        else
           q = true
      }
  Until q or ("no existen posibles movimientos")
    return q;
end
```

Refinamiento 3

```
boolean Try (int i, int x, int y) {
        boolean q;
        "Inicializa la selección del movimiento (K = 0)"
    do {
            "Sean nuevaX y nuevaY las coordenadas del siguiente movimiento
            definido por las reglas del ajedrez"
            (K = K + 1)    Obtiene las nuevas coordenadas por
                           el posible movimiento k –ésimo

            q = false;
        if (1 ≤ nuevaX  ≤ n  &&  1 ≤  nuevaY  ≤ n   &&
                                Tablero[nuevaX,nuevaY] =0){ //Es aceptable?

              Tablero[nuevaX,nuevaY] = i;
            if  (i < n^2) {
                  q = Try (i+1, nuevaX, nuevaY)
                 if (¬ q) {
                       Tablero[nuevaX,nuevaY] = 0;
                 }
            }
            else{
                  q = true;
            }
        }

    }while (!q  && K != 8);

        return q;

}
```

Movimientos del caballo

	5		**4**	
6				**3**
		C		
7				**2**
	8		**1**	

Forma de obtener las coordenadas (nuevaX,nuevaY), a partir de las condiciones iniciales (x,y):

$$nuevaX \leftarrow x + a_k$$

$$nuevaY \leftarrow y + b_k$$

$a_1 := 2 \quad b_1 := 1$

$a_2 := 1 \quad b_2 := 2$

$a_3 := -1 \quad b_3 := 2$

$a_4 := -2 \quad b_4 := 1$

$a_5 := -2 \quad b_5 := -1$

$a_6 := -1 \quad b_6 := -2$

$a_7 := 1 \quad b_7 := -2$

$a_8 := 2 \quad b_8 := -1$

Solución en pseudo Java

```
main (…) {
        a[1] = 2;
        a[2] = 1;
        a[3] = -1;
        a[4] = -2;
        a[5] = -2;
        a[6] = -1;
        a[7] = 1;
        a[8] = 2;
        b[1] = 1;
        b[2] = 2;
        b[3] = 2;
        b[4] = 1;
        b[5] = -1;
        b[6] = -2;
        b[7] = -2;
        b[8] = -1;

        for (int i = 1 ; i <= n ; i++)
                for (int j = 1 ; j <= n; j++)
                        Tablero[i][j] = 0;

        Tablero[1][1] = 1;

        q = Try( 2,1,1);

        if q then……..

} //Fin main
```

```
boolean Try (int i, int x, int y ){
      int k = 0;
      boolean q;
      do
            q = false;
            k = k+1;
            int nuevaX  = x + a[k];
            int nuevaY  = y + b[k];
            if (1 <= nuevaX && nuevaX <= n && 1 <= nuevaY &&
                        nuevaY <= n && h[nuevaX][nuevaY] ==0)  {
                h[nuevaX][nuevaY] = i;
                if (i < (n*n)) {
                      q = Try (i + 1, nuevaX, nuevaY);
                      if (!q) {
                            h[nuevaX][nuevaY] = 0
                      }
                }
                else {
                      q = true;
                }
            }
      while (!q  && k !=8);
      return q;

} //endTry
```

La **solución obtenida** para Try (2, 3, 3), con n = 5 es:

23	**10**	**15**	**4**	**25**
16	**5**	**24**	**9**	**14**
11	**22**	**①**	**18**	**3**
6	**17**	**20**	**13**	**8**
21	**12**	**7**	**2**	**19**

A continuación se muestra el código Java para el problema del caballo.

```
import ucn.StdIn;
import ucn.StdOut;

public class RecorridoCaballo {
    static int [] a = new int[8+1];
    static int [] b = new int[8+1];

    public static boolean intenta(int N, int i, int x,
                                  int y, int [][] Tablero ) {
        boolean q1;
        int k=0;
        do {
            q1 = false;
            k = k +1;
            int nuevaX = x + a[k];
            int nuevaY = y + b[k];

            if(nuevaX >= 1 && nuevaX <= N && nuevaY >=1 &&
              nuevaY <=N && Tablero[nuevaX][nuevaY] == 0){

                Tablero[nuevaX][nuevaY]= i;
                if (i < (N*N)) {
                    q1=RecorridoCaballo.intenta(N, i+1,
                                    nuevaX, nuevaY, Tablero);
                    if (q1 == false){
                            Tablero[nuevaX][nuevaY] = 0;
                    }
                }
                else{
                    q1 = true;
                }
            }
        }
        while(q1 == false && k!= 8);
        return q1;
    } //Fin intenta
```

```
public static void main(String [] args) {
    StdOut.print("Ingrese tamagno tablero ajedrez
                                (Mayor o igual a 2): ");
    int N = StdIn.readInt();
    int [][] Tablero = new int[N+1][N+1];

    StdOut.println("Ingrese las coordenadas de partida
              del caballo(Valores entre 1 - "+ N+")");
    StdOut.print("Ingrese valor de X: ");
    int x = StdIn.readInt();
    StdOut.print("Ingrese valor de Y: ");
    int y = StdIn.readInt();

    for(int i=1;i<=N;i++){
        for(int j=1; j<=N; j++){
            Tablero[i][j] = 0;
        }
    }

    a[1] = 2;
    a[2] = 1;
    a[3] = -1;
    a[4] = -2;
    a[5] = -2;
    a[6] = -1;
    a[7] = 1;
    a[8] = 2;
    b[1] = 1;
    b[2] = 2;
    b[3] = 2;
    b[4] = 1;
    b[5] = -1;
    b[6] = -2;
    b[7] = -2;
    b[8] = -1;

    Tablero[x][y]= 1;

    boolean q=RecorridoCaballo.intenta(N,2,x,y,Tablero);
```

```
        if (q == true) {
            //Impresión de una matriz"
            StdOut.println("Resultado recorrido caballo");
            for(int i=1;i<=N;i++) {
                for(int j=1; j<=N; j++){
                    StdOut.print(Tablero[i][j] + " ");
                }
                StdOut.println("");
            }
        }

        else{
            StdOut.println("No hay solucion");
        }

    } //fin main

}//Fin RecorridoCaballo
```

Tarea: Hacer el algoritmo para encontrar todas las soluciones