gabri11
www.wuolah.com/student/gabri11
4163

soluciones3.pdf

EXAMENES RESUELTOS (Ejercicios)

2° Estructuras de Datos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

WUOLAH

El más PRO del lugar puedes ser Tú.

¿Quieres eliminar toda la publi de tus apuntes?

¡Fuera Publi!
Concéntrate al máximo

Apuntes a full.
Sin publi y sin gastar coins

Para los amantes de la inmediatez, para los que no desperdician ni un solo segundo de su tiempo o para los que dejan todo para el último día.

Quiero ser PRO

4,95 / mes

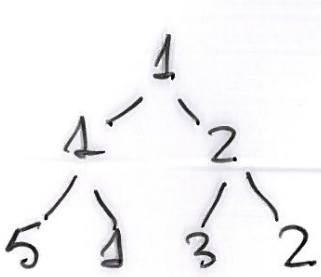
Se define un arbol BWM como un ABB en el que la altura de los subarboles izquierdo y derecho en cada nodo puede diferir como maximo en 2.

Construir el arbol BWM asociado al conjunto de claves: {11, 10, 4, 5, 2, 6, 7, 9, 8}

5

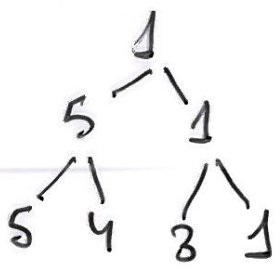
Un arbol de seleccion es un arbol binario en el que cada nodo tiene la etiqueta del menor de sus hijos. Diseñar una funcion que determine si un arbol binario T es un arbol de seleccion.

bool seleccion (bintree<int> & T);



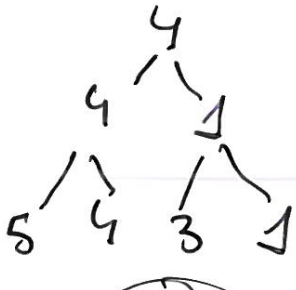
SI

TRUE



NO

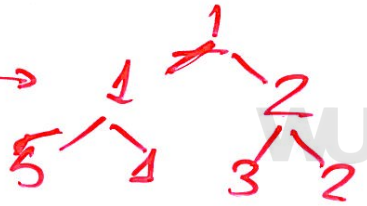
FALSE



NO

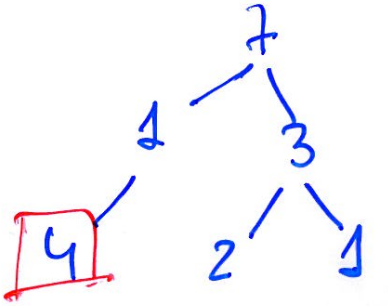
FALSE

Dado un vector de enteros, construir a partir de ellos un arbol de seleccion

{1, 5, 3, 2} ⟶ 

Se define la trayectoria de una hoja en un árbol binario como la suma del contenido de todos los nodos desde la raíz hasta la hoja multiplicado por el nivel en que se encuentra.

Implementar un procedimiento que, dado un árbol binario ~~cualquiera la trayectoria de sus hojas~~ devuelva la hoja con mayor trayectoria



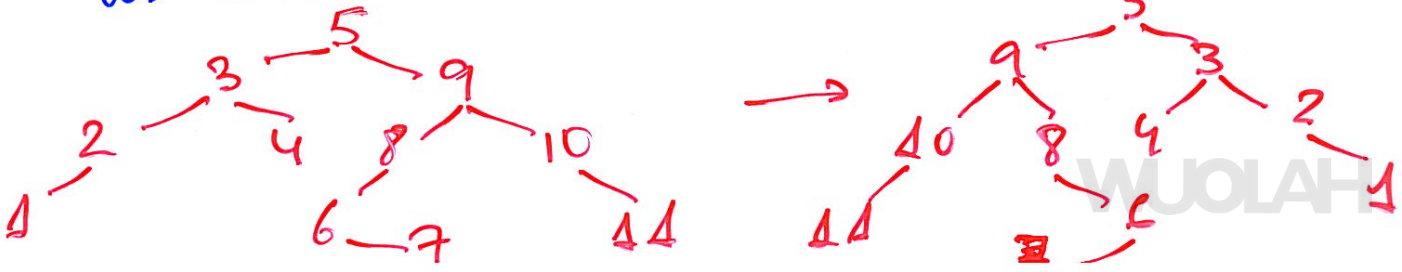
Tray (4) = 4x3 + 1x2 + 7x1 = 21

Tray (2) = 2x3 + 3x2 + 7x1 = 19

Tray (1) = 1x3 + 3x2 + 7x1 = 16

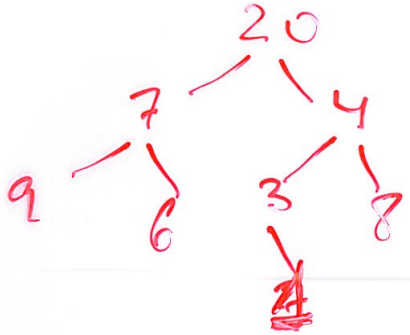
Se denomina "onaedro" de un ABB T, al árbol binario $\hat{T}$ construido de forma que todo hijo izquierda de T pasa a ser hijo a la derecha de $\hat{T}$ y todo hijo derecha de T pasa a ser hijo izquierda de $\hat{T}$

Implementar la función que dado un ABB construya su "onaedro" y diseñar un algoritmo para recorrer el "onaedro" de forma que visitemos los nodos en orden ascendente de valor

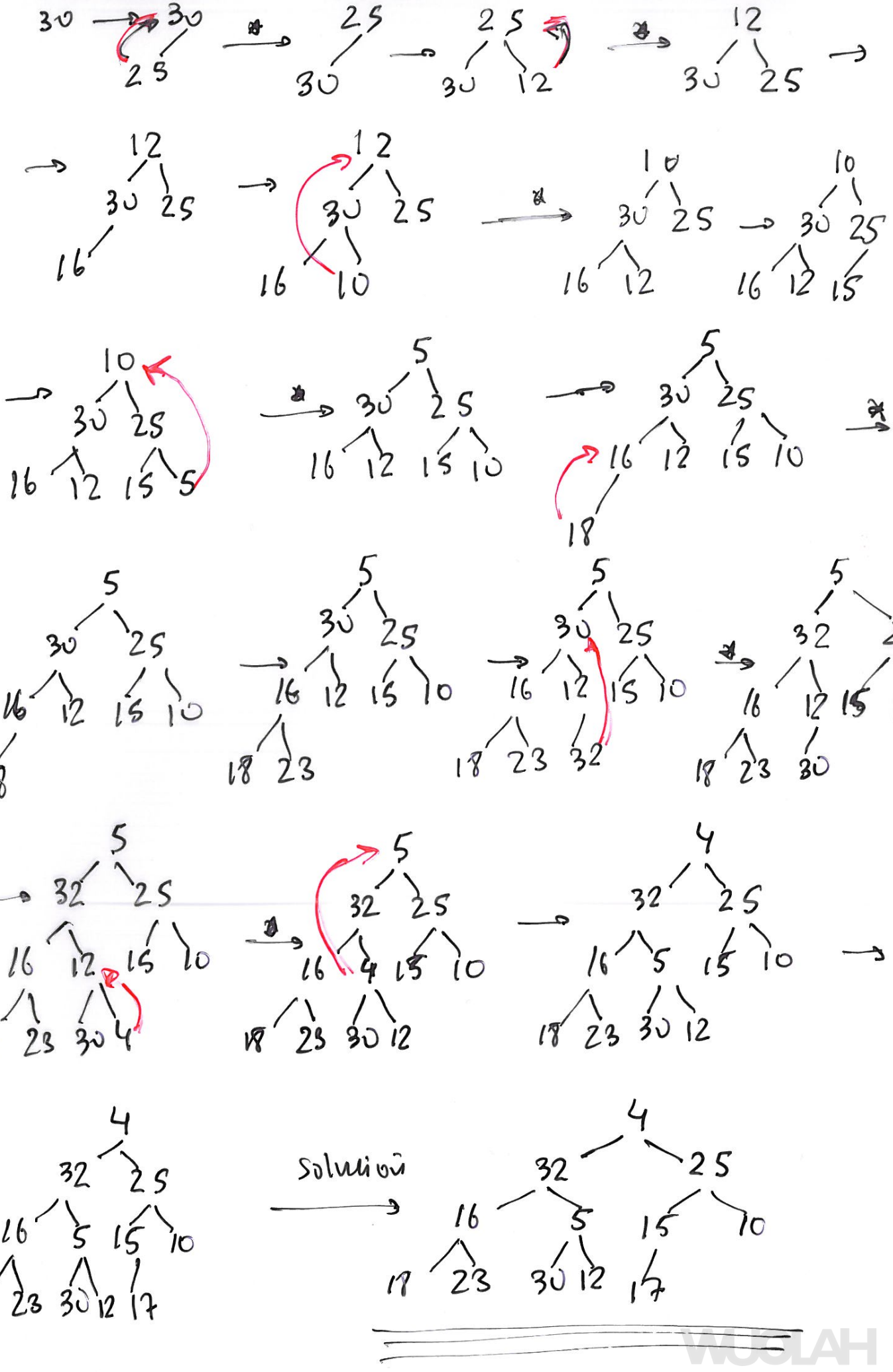


Un "recorrido guiado" sobre un arbol binario comienza listando la raiz para a continuación en cada iteración seleccionar el nodo más pequeño de entre los nodos disponibles en ese momento que no hayan sido listados independientemente de en qué rama se encuentren. Se entiende por nodo disponible aquel cuyo padre ya ha sido procesado (excluyendo la raiz)

Construir una función que permita hacer un recorrido guiado en un arbol. Pueden usarse estructuras auxiliares.



Recorrido guiado: 20, 4, 3, 1, 7, 6, 8, 9



Sea T un árbol binario con n nodos. Se define un K nodo como un nodo v que cumple la condición de que el número de descendientes en el subárbol izquierdo de v difiere del número de descendientes del subárbol derecho en al menos 5. Usando el TDA bintree construya una función que encuentre los 5 nodos de T

```
bool cinconodo (const bintree<int>::nodo n)
    {
    if abs(contar(n.left()) - contar(n.right())

    return true
    else return false;
    }

int contar (bintree<int>::nodo n)
    {
    if n = null()
    return 0;
    else return 1 + contar(n.left())
                  + contar(n.right())
    }

int recuento (bintree<int> a)
    { int numero = 0;
    bintree<int>::preorder_iterator p = a.begin();
    while (p != a.end())
        if cinconodo(p) numero++;
        ++p; }
```

Dado un árbol binario de enteros, implementar una función que cuente el número de caminos cuya suma de valores de las etiquetas de los nodos que los componen sea exactamente k

```
int numeroCaminos (bintree<int> &ab, int k,
                   bintree<int>::nodo n)
{
if (n.left()==bintree<int> null() &&
    n.right()==bintree<int> null())
      if (*n==k) return 1
      else return 0;
else {
      int contador=0;
      if (n.left()!=bintree<int>:: null()
          contador += numeroCaminos(ab, k-*n,
                                    n.left());
      if (n.right()!=bintree<int>:: null()
          contador += numeroCaminos(ab, k-*n,
                                    n.right());
      return contador;
}
}
```

```
multiset<int> multi_interseccion (const multiset<int>
                & m1, const multiset<int> m2)
{
  multiset<int>::iterator i1 = m1.begin();
  multiset<int>::iterator i2 = m2.begin();

while ((i1!=m1.end()) && (i2!=m2.end()))
    {
      if (*i1 == *i2)
        {
          result.insert(*i1);
          i1++;
          i2++;
        }
      else if (*i1 < *i2)
          {
            i1++;
          }
          else i2++;

    }

  return result;
}
```