

Relación Árboles

Alumno : Alberto Llamas González

2º Grado en Ingeniería Informática

Índice ejercicios

- Ejercicio 1
- Ejercicio 2
- Ejercicio 3
- Ejercicio 4
- Ejercicio 5
- Ejercicio 6
- Ejercicio 7
- Ejercicio 8
- Ejercicio 9
- Ejercicio 10
- Ejercicio 11
- Ejercicio 12
- Ejercicio 13
- Ejercicio 14

Ejercicios resueltos

Ejercicio 1

Sea A un árbol binario con n nodos. Se define el ancestro común más cercano (AMC) entre 2 nodos v y w como el ancestro de mayor profundidad que tiene tanto a v como a w como descendientes (se entiende como caso extremo que un nodo es descendiente de sí mismo). Diseñar una función que tenga como entrada un árbol binario de enteros y dos nodos v y w como salida y el AMC de v y w.

```
ArbolBinario<int>::nodo AMC (const ArbolBinario<int>&ab, const
ArbolBinario<int>::nodo v,
                                const ArbolBinario<int>::nodo w) {
    vector<ArbolBinario<int>::nodo> nodos;
    while(v->padre != 0){
        nodos.insert(v->padre());
        v=v->padre();
    }

    vector<ArbolBinario<int>::nodo>::iterator it;
    bool find = false;
    while(find && w->padre() != 0){
```

```

    it = nodos.find(w->padre);
    if(it != nodos.end()){
        find = true;
    }
    else
        w = w->padre();
    }
    if (find)
        return *it;
    else
        return ArbolBinario<int>::nodo nodo();
}

```

Ejercicio 2

Dado un árbol binario de enteros, se dice que está sesgado a la izquierda si para cada nodo, se satisface que la etiqueta de su hijo izquierda es menor que la de su hijo derecha (en caso de tener un sólo hijo, éste ha de situarse necesariamente a la izquierda).

Se pide:

1. Implementar un método que compruebe si un árbol binario A está sesgado hacia la izquierda.
2. Implementar un método que transforme un árbol binario en un árbol sesgado hacia la izquierda

La transformación debe preservar que si un nodo v es descendiente de otro w en A, también lo debe ser en el árbol transformado, por lo que no es válido hacer un intercambio de las etiquetas de los nodos

```

//1
bool sesgado(const bintree<int>&ab){
    return sesgado(ab.root());
}

bool sesgado(const bintree<int>::node n){
    if (!n.null()){
        if(n.left() >= n.right()){
            return false;
        } else {
            return sesgado(n.left()) && sesgado(n.right());
        }
    } else {
        return true;
    }
}

//2

```

Ejercicio 3

Un árbol de sucesos es un árbol binario donde cada nodo tiene asociada una etiqueta con un valor real en el intervalo $[0,1]$. Cada nodo que no es hoja cumple la propiedad de que la suma de los valores de las etiquetas de sus hijos es 1. Un suceso es una hoja y la probabilidad de que éste ocurra viene determinada por el producto de los valores de las etiquetas de los nodos que se encuentran en el camino que parte de la raíz y acaba en dicha hoja. Se dice que un suceso es probable si la probabilidad de que ocurra es mayor que 0.5. Usando el TDA Árbol binario:

1. Diseñar una función que compruebe si un árbol binario A es un árbol de sucesos. Su prototipo será

```
bool check_rep (const bintree &A)
```

2. Diseñar una función que indique si existe algún suceso probable en el árbol de probabilidades A. Su prototipo será:

```
bool probable (const bintree &A)
```

```
// 1
bool check_rep(const bintree<float> &A){
    return check_rep(A.root());
}

bool check_rep(bintree<float>::node n){
    bool es_sucessos = true;
    if (!n.null()){
        double et = 1.0;

        if (!n.left().null() || !n.right().null()){
            et = 0.0;
            if (!n.left().null())
                et += n.left();
            if (!n.right().null())
                et+= n.right();
        }
        es_sucessos = (et == 1.0) && (0.0 <= *n) && (*n <= 1.0) &&
check_rep(n.left()) && check_rep(n.right());
    }
    return es_sucessos;
}

//2
bool probable (const bintree<float> &A){
    return probable(A.root(), *n);
}

bool probable(bintree<float>::node n, double &prod){
    bool es_probable = true;
    if (!n.null()){
        if (n.left().null() && n.right().null()){
            prod = 1.0;
        } else {
            prod = n.value();
            if (n.left().null())
                prod *= probable(n.left(), prod);
            if (n.right().null())
                prod *= probable(n.right(), prod);
        }
    }
    return es_probable;
}
```

```

    prod *= *n;
    es_probable = (prod >= 0.5);
}
} else{
    es_probable = probable(n.left(), prod) || probable(n.right(), prod);
}

return es_probable;
}

```

Ejercicio 4

Dado un árbol binario de enteros (positivos y negativos) implementar una función que obtenga el número de caminos, en los que la suma de las etiquetas de los nodos que los componen sumen exactamente k

```
int NumeroCaminos (bintree & ab, int k);
```

```

int NumeroCaminos(bintree<int> &ab, int k){
    return NumeroCaminos(ab.root(), k);
}

int NumeroCaminos (bintree<int>::node n, int k){
    if (n.left().null() && n.right().null()){
        if (*n == k)
            return 1;
        else
            return 0;
    } else{
        int sumador = 0;
        if (!n.left().null())
            sumador+= NumeroCaminos(n.left(), k-*n);
        if (!n.right().null())
            sumador+= NumeroCaminos(n.right(), k-*n);
        return sumador;
    }
}

```

Ejercicio 5

Dado un bintree T, implementar una función

```
void prom_nivel(bintree &T, list &P);
```

que genere una lista de reales P, donde el primer elemento de la lista sea el promedio de los nodos del árbol de nivel 0, el segundo sea el promedio de los de nivel 1, el tercero el promedio de los de nivel 2, y así sucesivamente. Es decir, que si el árbol tiene profundidad N, la lista tendrá N+1 elementos de tipo float.

```

void prom_nivel(bintree<int> &T, list<float> &P){
    if (!n.null()){
        typedef pair <const bintree<int>::node, int> minfo;
        queue<minfo> miq;
        miq.emplace(T.root(), 0);
        list<float> salida;
        int nivel, suma, nhijos;
        float media;
        while(!miq.empty()){
            minfo aux = miq.front();
            miq.pop();

            if ((aux.second) == nivel){
                suma+= *aux.first;
                nhijos++;
            } else{
                media = suma / nhijos*1.0;
                salida.push_back(media);
                suma = *aux.first;
                media = 0; nhijos = 0;
                nivel++;
            }

            if (!aux.first.left().null())
                miq.emplace(aux.first.left(), aux.second+1);
            if (!aux.first.right().null())
                miq.emplace(aux.first.right(), aux.second+1);
        }
        media = suma/nhijos*1.0;
        salida.push_back(media);
        P = salida;
    } else{
        P=list<float> vacia;
    }
}

```

Ejercicio 7

Se dice que un árbol binario de enteros es inferior a otro si (teniendo la misma estructura de ramificación), los elementos del primero, en los nodos coincidentes en posición, son menores que los del segundo. Implementar una función booleana que dados dos árboles binarios, devuelva true si el primero es inferior al segundo:

```
bool es_inferior(const bintree &ab1, const bintree &ab2);
```

```

bool es_inferior(bintree<int> &ab1, bintree<int> &ab2){
    bool es_inferior(bintree<int> &ab1, bintree<int> &ab2){
        auto it2 = ab2.begin_preorder();
        for(auto it1 = ab1.begin_preorder(); it1 != ab1.end_preorder(); ++it1){
            if(*it2 < *it1)
                return false;
            ++it2;
        }
        return true;
    }
}

```

Ejercicio 8

Implementar una función:

```
bool es_menor(bintree&A,bintree&B);
```

que devuelve true si A < B, con A y B árboles binarios no vacíos. Se entiende que A < B si:

(a < b) ó (a = b) && (Ai < Bi) ó (a = b) && (Ai = Bi) && (Ad < Bd)

```

bool es_menor(bintree<int> &ab1, bintree<int> &ab2){
    return es_menor(ab1.root(), ab2.root());
}

bool es_menor(bintree<int>::node n1, bintree<int>::node n2){
    bool es_menor = true;
    if (n1.null() && n2.null())
        es_menor = false;
    else{
        if(*n1 < *n2 || (*n1 == *n2) && (*n1.left()) == (*n2.left()) &&
(*n1.right()) < (*n2.right()))
            || (*n1 == *n2) && (*n1.left()) == (*n2.left()))
            es_menor = true;
        es_menor = es_menor(n1.left(), n2.left()) && es_menor(n1.right(),
n2.right());
    }
    return es_menor;
}

```

Ejercicio 9

Implementar una función:

```
int nodos_k (bintree &A, int k);
```

que devuelve el número de nodos de un árbol binario cuya etiqueta es exactamente igual a k.

```

int nodos_k(bintree<int> &A, int k){
    int contador = 0;
    for(auto it = A.begin_preorder(); it != A.end_preorder(); ++it)
        if(*it == k)
            contador++;
    return contador;
}

```

Ejercicio 10

Implementar la función:

bintree::node siguiente_nodo_nivel(const bintree::node &n, const bintree &arb)
que dado un nodo n de un árbol binario arb, devuelve el siguiente nodo que está en ese mismo nivel del árbol. Si es el último nodo del nivel devuelve el primero del siguiente nivel

```

bintree<T>::node siguiente_nodo_nivel(const bintree<T>::node &n, const
bintree<T> &arb){
    if (n.null())
        return bintree<T>::node();
    else{
        if (n.root()){
            if (!n.left().null())
                return n.left();
            if (!n.right().null())
                return n.right();
        } else {
            typename queue<bintree<T>::node> miq;
            typename bintree<T>:: node raiz = arb.root();
            miq.push(raiz);
            while(!miq.empty()){
                typename bintree<T>::node aux = miq.front();
                miq.pop();

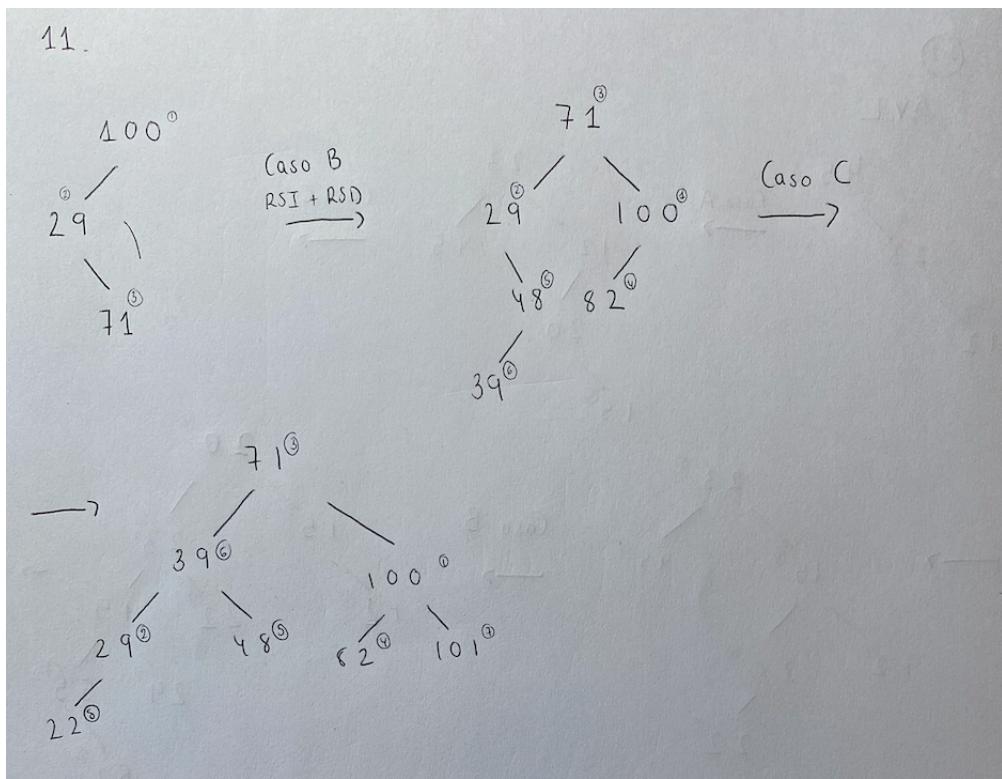
                if (!aux.left().null())
                    miq.push_back(aux.left());
                if (!aux.right().null())
                    miq.push_back(aux.right());

                if (aux == n)
                    return miq.front();
            }
        }
    }
}

```

Ejercicio 11

Construir un AVL a partir de las claves: {100;29;71;82;48;39;101;22}. Indicar cuando sea necesario el tipo de rotación que se usa para equilibrar.



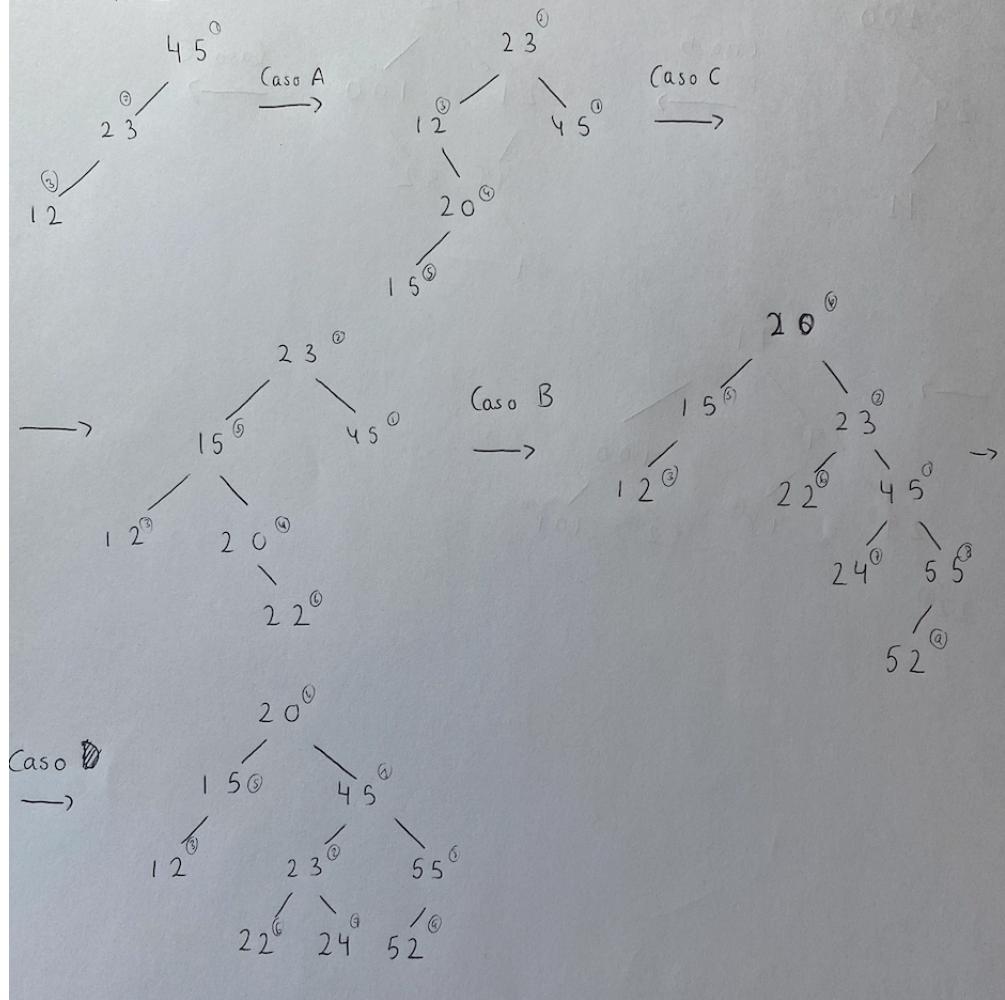
Ejercicio 12

Construir el AVL y el APO que resultan de insertar (en ese orden) los elementos del conjunto de enteros {45;23;12;20;15;22;24;55;52}

AVL

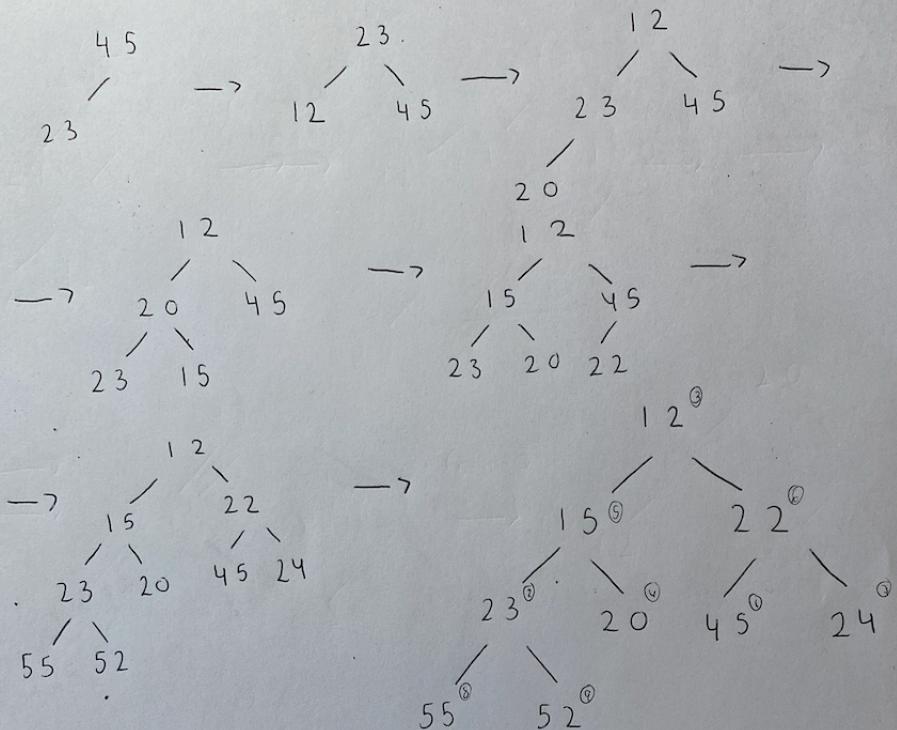
(4.2)

AVL



AVL

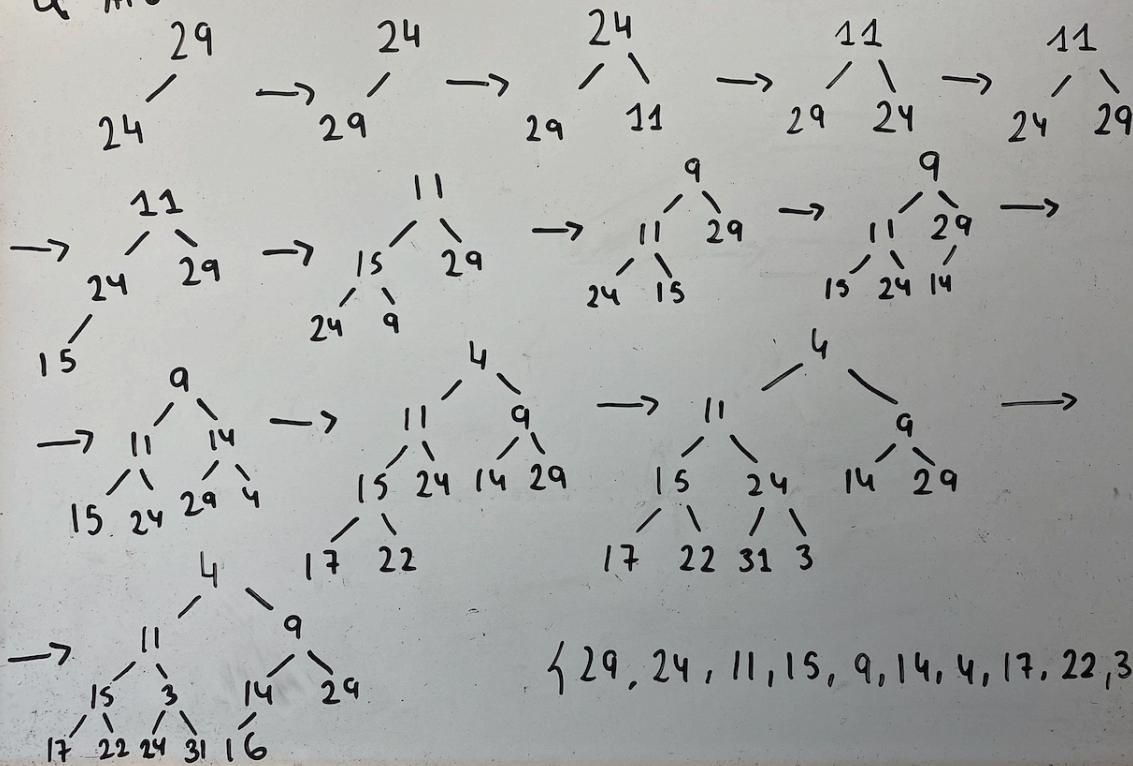
12 APO:



Ejercicio 13

Un "q-APO" es una estructura jerárquica que permite realizar las operaciones eliminar-mínimo e insertar en un tiempo $O(\log_2(n))$, y que tiene como propiedad fundamental que para cualquier nodo X la clave almacenada en X es menor que la del hijo izquierdo de X y esta a su vez menor que la del hijo derecho de X , siendo el árbol binario y estando las hojas empujadas a la izquierda. Implementar una función para insertar un nuevo nodo en la estructura y aplicarla a la construcción de un q-APO con las claves {29, 24, 11, 15, 9, 14, 4, 17, 22, 31, 3, 16}.

Q-APO



Ejercicio 14

Se denomina "odanadero" T^* de un ABB T , al arbol binario construido de forma que todo hijo izquierdo de un nodo en T pasa a ser hijo derecho de T^* y todo hijo derecho de un nodo en T pasa a ser hijo izquierdo de ese nodo en T^* . Implementar una función que tenga como entrada un ABB y devuelva su "odanadero". ¿Cómo habría que recorrer el "odanadero" para que las etiquetas de los nodos se listen en orden ascendente de valor? *Habría que recorrello en inorder*

```

bintree<int> onaedro(bintree<int> &ab) {
    ab.root() = onaedro(ab.root());
    return ab;
}

void swap(bintree<int>::node n, bintree<int>::node m){
    bintree<int>::node aux;
    aux = n.left();
    n.left() = m.right();
    m.left() = aux;
}

bintree<int>::node onaedro(bintree<int>::node &n){
    if (!n.null()){
        swap(n.left(), n.right());
        onaedro(n.left());
        onaedro(n.right());
        return n;
    } else{

```

```
    return null;
}
}

//He pensado otra forma sin usar swap pero no se si se puede hacer
/*
bintree<int>::node onaedro(bintree<int>::node &n){
    if (!n.null()){
        bintree<int>::node aux;
        aux = n.left();
        //suponemos implementados metodos de asignacion de nodos
        n.setLeftSon(onaedro(n.right()));
        n.setRightSon(onaedro(aux));
        return n;
    } else{
        return null;
    }
}

*/
```