postdata9
www.wuolah.com/student/postdata9

41634

# ejerciciosex.pdf

*Exámenes resueltos*

**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

**Ejercicio 12. (Hoja Ejercicios Examen STL) (Diciembre 2010)**

**Considérese el TDA vector_disperso<T>, que representa un vector matemático instanciado para un tipo de dato numérico T. En estos vectores, la mayoría de componentes del vector son nulas y sólo unas pocas tienen valor distinto de cero, por lo que es recomendable utilizar una representación que sólo almacene de forma explícita las componentes no nulas. Se pide:**

**a) Dar una representación para el TDA vector_disperso<T> que tenga en cuenta las observaciones realizadas anteriormente.**

Vector disperso: aquel que almacena lo importante de un vector. Ejemplo: si tiene todo ceros y 3 números, sólo guarda los números.

Dado el siguiente vector su

| 0 | 0 | 1 | 0 | 0 | 2 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

vector disperso sería:

| <2,1> | <5,2> |
|---|---|

es decir, en la posición 2 hay un 1, en a posición 5 hay un 2, todo lo demás es 0.

Una buena representación para el vector disperso, sería un vector de pares de entero y T, esto es:

```
template <class T>
class vector_disperso<T>{
      private:

            //el int es el índice del vector, y T lo que hay
            vector<pair<int, T>> datos;

            //número de elementos del vector
            int n;
}
```

En el ejemplo anterior, el vector sería el mostrado y n = 6.

**b) Implementar la operación vector_disperso<T> vector_disperso<T>::operator+ (const vector_disperso<T> &v, const_vector<T> &w), que calcula y devuelve la suma de dos vectores.**

```
vector_disperso<T>   vector_disperso<T>::operator+   (const   vector_disperso<T>   &v,
const_vector<T> &w){

      while(i < datos.size() && j < v.datos.size(){
            if(datos[i].first == v.datos[j].first)

            else if( datos[i].first < v.datos[j].first

            else

      }

      while(i < datos.size()){
            res.datos.push_back(datos[i++]);
      }

      while(j < v.datos.size()){
            res.datos.push_back[datos[j++]);
      }
}
```

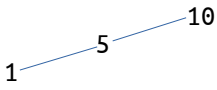
Otra versión:

```
//VD.h

template <class T>
class VD{
	private:
		T nulo;
		map<int,T> datos;
		int n;

	public:
		VD<T> operator+(const VD<T> &v)const{
			VD<T> res(v);
			res.n = (res.n < n) ? n : res.n;

			typename map<int,T>::const_iterator it;

			for(it = datos.begin(); it != datos.end(); ++it){
				res.datos[it → first] = res.datos[it → first]+it → second;
			}

			return res;
		}
		//si una clave no existe, la crea
}
```

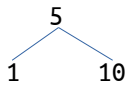
Tipo de datos dependiente del nombre cualificado(::)
Traducción → T y :: => typename

VD<T>:: No es typename

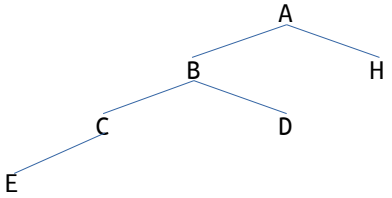
**Encontrar en qué nodo ocurre el desequilibrio**



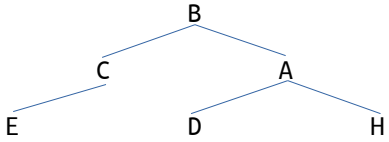

Hay desequilibrio =>




Esto se llama RSD 5, Rotación Simple a la Derecha de 5



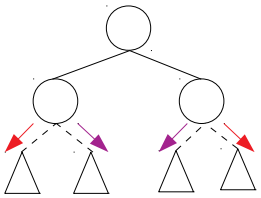

=>

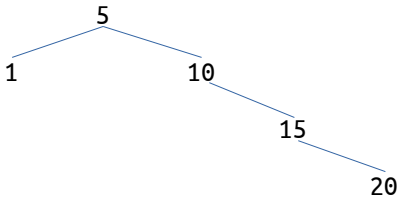



Como A rota a la derecha, pierde el hijo izquierdo, por ello, el hijo izquierdo de B pasa a ser el de A, ya que sabemos con certeza que NO tiene hijo izquierda.

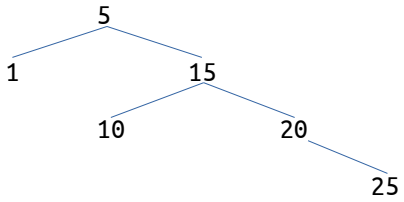
El desequilibrio puede ocurrir en:

- **Rama externa**
- **Rama interna**

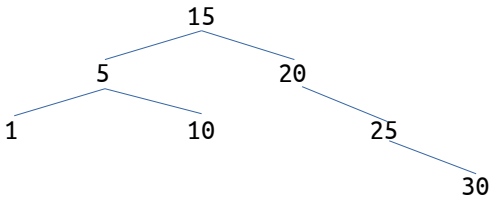




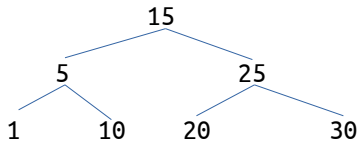

RSI 15
=>




RSI 15
=>

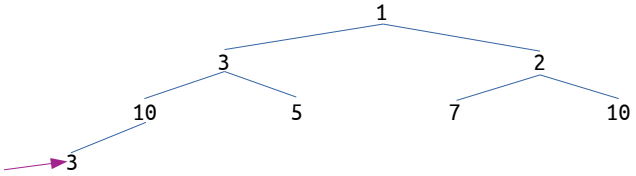


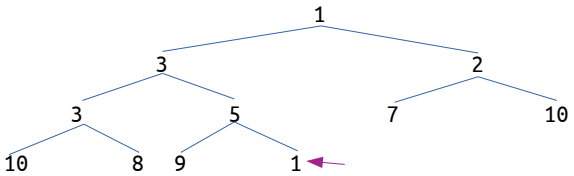

RSI 25
=>

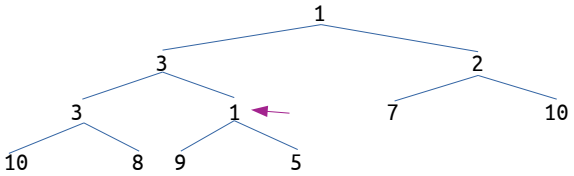


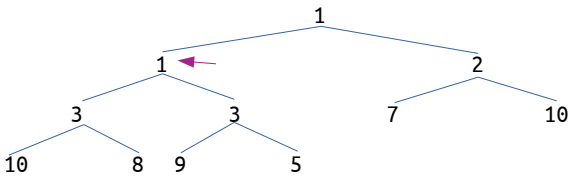
**Crear un apo con los siguientes números**

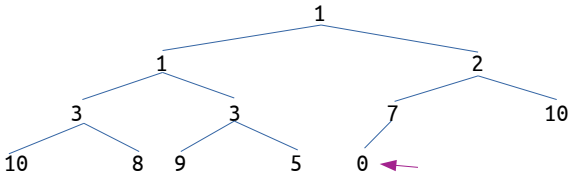
1, 3, 2, 10, 5, 7, 10, 3, 8, 9, 1, 0

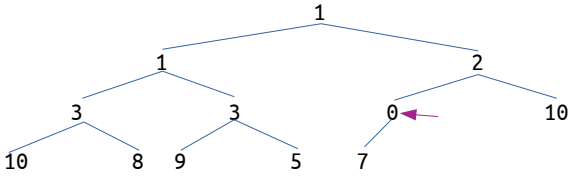


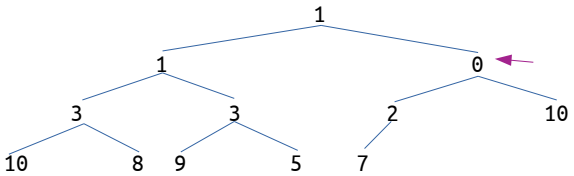



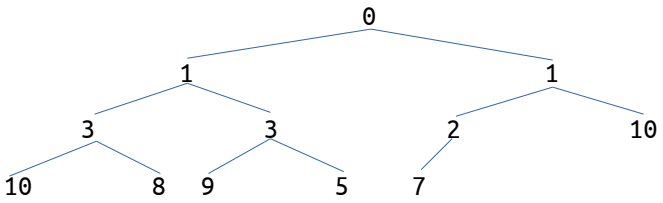



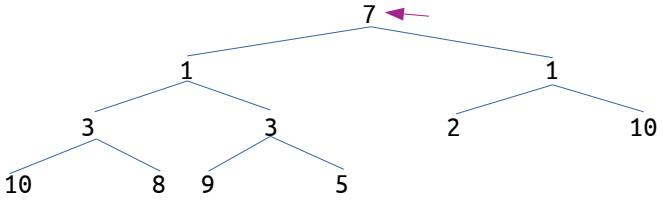



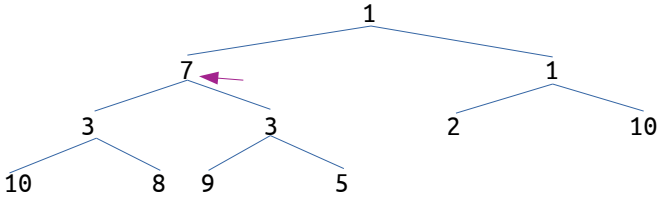



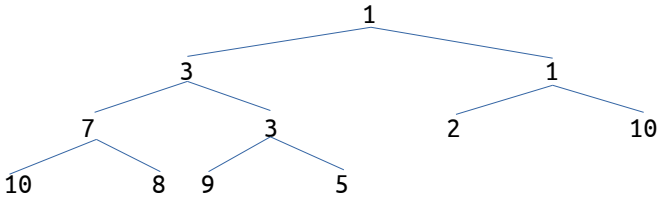





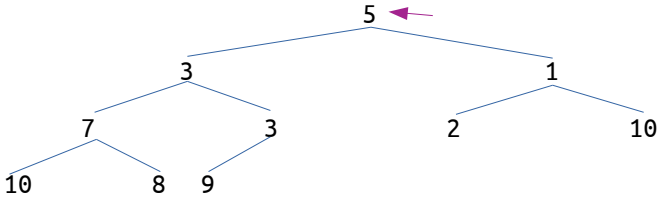

Vamos a borrar 2 veces la raíz:
Primera vez:

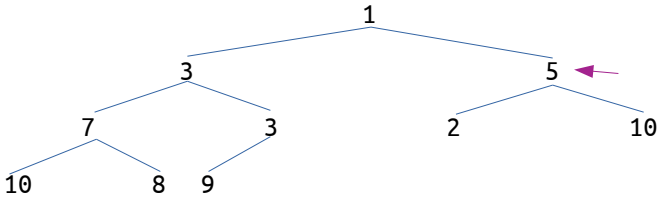


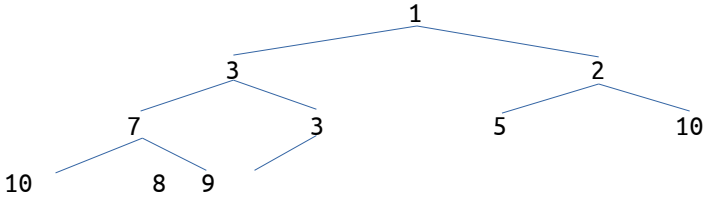




Segunda vez:

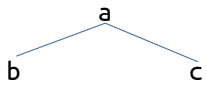


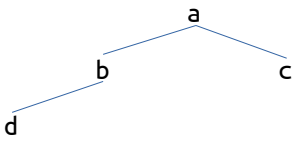



**Recorrido en árboles**

- Preorden
- Inorden
- Postorden



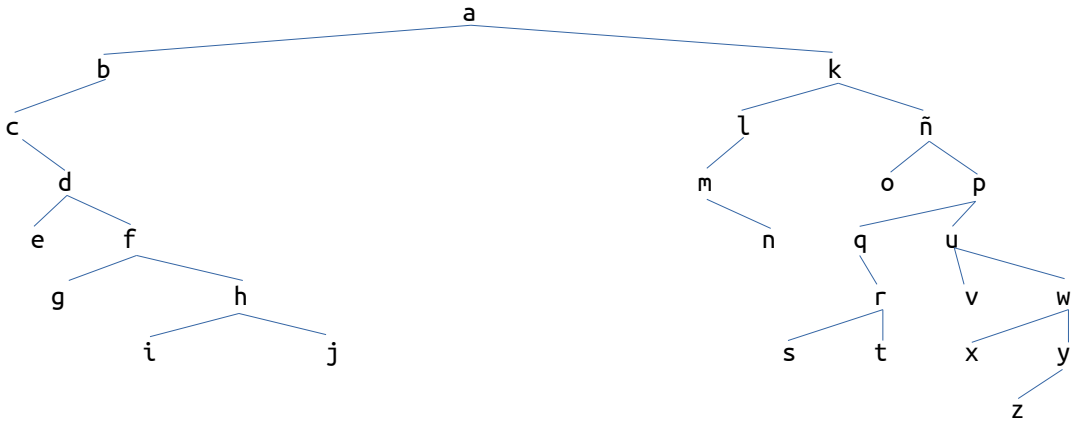

Pre: a b c
In: b a c
Post: b c a




Pre: a b d c

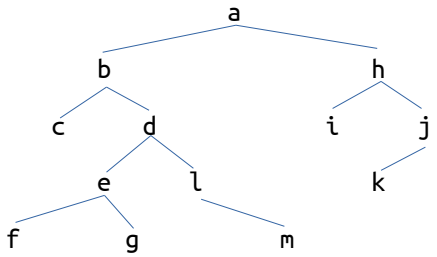
In: d b a c

Post: d b c a




Pre: a b c d e f g h i j k l m n ñ o p q r s t u v w x y z
In: c e d g f i h j b a m n l k o ñ q s r t p v u x w z y
Post: e g i j h f d c b n m l o s t r q v x z y w u p ñ k a

**Función para recorrer en preorden, inorden y postorden:**




Preorden: a b c d e f g l m h i j k

```
void Preorden(bintree<int>::node n){
      if(!n.null())
            cout << *n;
            Preorden(n.left());
            Preorden(n.right());
}
```

Inorden: c b f e g d l m a i h k j

```
void Inorden(bintree<int>::node n){
      if(!n.null())
            Inorden(n.left());
            cout << *n;
            Inorden(n.right());
}
```

Postorden: c f g e m l d b i k j h a

```
void Postorden(bintree<int>::node n){
      if(!n.null())
            Postorden(n.left());
            Postorden(n.right());
            cout << *n;
}
```

**Comprobar si un nodo es hoja o no**

```
void SoyNodoHoja(bintree<int>::node n){
	return n.left().null() && n.right().null();
}
```

**Función que devuelve la profundidad de un nodo**

```
int Profundidad(bintree<int>::node n){
	int p = -1;
	if(!n.null())
		p = Profundidad(n.parent()) + 1;
	return p;
}
```

**Función que devuelve la altura de un nodo**

```
int Altura(bintree<int>::node n){
	int h = -1;
	if(!n.null())
		h = max(Altura(n.left()), Altura(n.right())) + 1;

	return h;
}
```

**Función que calcule cuántas hojas tiene un árbol**

```
int numHojas(bintree<int> a){
	return numeroHojas(a.root());
}

int numeroHojas(bintree<int>::node n){
	int hojas = 0;
	if(!n.null())
		if(n.left().null() && n.right().null())
			hojas = 1;
		else
			hojas = numeroHojas(n.left()) + numeroHojas(n.right());
	return hojas;
}
```

**Función que me diga si un árbol es ABB**

```
bool arbol_ABB(bintree<int>::node a){
	bool es = true;
	if(!n.null())
		if((*n.left()) < n && (*n.right()))
			es = arbol_ABB(n.left()) && arbol_ABB(n.right());
	return es;
}

bool arbol_ABB(bintree<int>::node a){
	bool es = false;
	if(!n.null())
		if(!n.left().null() && (*n.left()) < (*n))
			if(!n.right().null() && (*n.right()) < (*n))
				es = arbol_ABB(n.left()) && arbol_ABB(n.right());
	return es;
}
```

```
bool arbol_ABB(bintree<int> a){
	return arbol_ABB(a.root());
}
```

Estas funciones de arriba están mal, ya que para saber si un árbol es ABB o no, necesitamos de una función auxiliar que me asegure que los hijos de un nodo son menores que él mismo.

Función para ver si los nodos hijos tienen un valor menor que valor, es una función auxiliar que vamos a usar para ver si un árbol es un ABB o no.

```
bool Menores(bintree<int>::node n, int valor){
	bool es = true;

	if(!n.null())
		es = *n < valor && Menores(n.left(),valor) && Menores(n.right(),valor);

	return es;
}

template<class F>
bool Menores(bintree<int>::node n, int valor, F f){
	bool res = true;
	if(!n.null())
		res = f(*n, valor) && f(*n, valor) …
	return res;
}

bool esABB(bintree<int>::node n)
	bool res1, res2, res3, res4;

	if(!n.null())
		res1 = Menores(n→hizq, n→et);
		res2 = Menores(n→hdcha, n→et);
		res3 = esABB(n→hizq);
		res4 = esABB(n→hdcha);

	return res1 && res2 && res3 && res4;
}
```

**(Junio 2010) Considérese un árbol binario de búsqueda BST<T>. Añade un método ReverseBST a la clase para cambiar el criterio de ordenación del BST. El receptor quedará modificado de forma que para cada nodo del árbol se cumple: a la izquierda están los elementos que sean mayores que el nodo y a su derecha los que sean menores o iguales.**

```
template<typename T>
class BST{
	private:
		bintree<T> arb;
		void reverse(typename bintree<int>::node n);
	public:
		bintree<T> ReverseBST();
		…
}

bintree<T> BST::ReverseBST(){
	return reverse(arb.root());
}

template<class T>
void BST<T>::reverse(typename bintree<T>::node n){

	//si el nodo no es nulo
	if(!n.null())
		bintree<T> ri, rd;

		//podamos el hijo izquierda y lo insertamos en ri
		arb.prune_left(n,ri);

		//podamos el hijo derecha y lo insertamos en rd
		arb.prunee_right(n,rd);

		//insertamos el hijo a la derecha como hijo izquierda
		arb.insert_left(n,rd);

		//insertamos el hijo a la izquierda como hijo derecha
		arb.insert_right(n,ri);

		//y así con todos los hijos
		ReverseBST(n.left());
		ReverseBST(n.right());
}
```

**(Febrero 2013) 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 este 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:**

**a) Diseñar una función que compruebe si un árbol binario A es un árbol de sucesos. Su prototipo será bool check_rep(const bintree<float> &A)**

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

//en esta función supongo que un nodo tiene 0 ó 2 hijos
bool check_rep(bintree<float>::node n){
	bool res = true;

	if(!n.null())
		if(!n.left().null() && !n.right().null())
			res = check_rep(n.left()) && check_rep(n.right()) &&
				(*(n.left()) + *(n.right()) == 1 ) && *(n)>= 0 && *(n) <= 1);
		else return res;
	return res;
}

//otra forma de hacerlo suponiendo que un nodo puede tener hasta 2 hijos
bool check_rep(bintree<float>::node n){
	bool es = true;

	if(!n.null())
		double etq = 1.0;
		if(!n.left().null() || !n.right().null())
			etq = 0.0;
			if(!n.left().null())
				etq+= *(n.left());
			if(!n.right().null())
				etq+= *(n.right());

		es = (etq == 1.0) && (0.0 <= *(n)) && (*(n) <= 1.0) && check_rep(n.left())
		     && check_rep(n.right();
	return es;
}
```

**b) Diseñar una función que indique si existe algún suceso probable en el árbol de probabilidades A. Su prototipo será: bool probable (const bintree<float> &A).**

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

bool probable(bintree<float>::node n){
	bool es = true;

	if(!n.null())
		if(n.left().null() && n.right().null())
			valor *= *n;
			es = (valor >= 0.5);
	else
		es = probable(n.left(), valor*(*n)) || probable(n.right(), valor*(*n));

	return es;
}
```

**Función que determine si un árbol es k_balanceado**

```
template <class T>
bool esk_balanceado(typename bintree<T> t, int k){
	return esk_balanceado(t.root(), k);
}

template <class T>
bool esk_balanceado(typename bintree<T>::node n, int k){
	bool balanceado = true;

	if(!n.null())
		balanceado = ((abs(NumeroNodos(n.left())-(abs(NumeroNodos(n.right()) < k)
				&& esk_balanceado(n.left(),k) && esk_balanceado(n.right(),k);

	return balanceado;
}

template <class T>
int NumeroNodos(typename bintree<T>::node n){
	int cont = 0;

	if(!n.null())
		cont = NumeroNodos(n.left()) + NumeroNodos(n.right()) + 1;

	return cont;
}
```

Las 2 funciones anteriores es una manera de hacerlo pero no es eficiente. Escribiremos a continuación

```
bool esk_balanceado(typename bintree<T>::node n, int k, int &num){
	bool balanceado = true;

	if(!n.null())
		int numIzq, numDcha;

		balanceado = esk_balanceado(n.left(), k, numIzq)
					&&
				esk_balanceado(n.right(), k, numDcha)
					&&
				abs(numIzq - numDcha) < k;

		num = numIzq + numDcha + 1;

	return balanceado;
}
```

**(Febrero 2012) (Hojita Examen Tablas Hash) Insertar las claves {10, 39, 6, 32, 49, 18, 41, 37} en una Tabla Hash cerrada de tamaño 11, usando como función hash h(k) = (2k+5)%11 y para resolver las colisiones rehashing doble usando como función hash secundaria h0(k) = 7 – (k%7). ¿Cuál es el rendimiento de la tabla?**

```
      hi(k) = (h(k) + h0(k)*(i-1)) % M

h1(10) = (2*10 + 5) % 11 = 3

h1(39) = (2*39 + 5) % 11 = 6

h1(06) = (2*06 + 5) % 11 = 6
      h2(06) = (6 + (7 – (6%7))*1) % 11 = (6 + 1) % 11 = 7

h1(32) = (2*32 + 5) % 11 = 3
      h2(32) = (3 + (7 – (32%7))*1) % 11 = (3 + 3) % 11 = 6
      h3(32) = (3 + (7 – (32%7))*2) % 11 = (3 + 6) % 11 = 9

h1(49) = (2*49 + 5) % 11 = 4

h1(18) = (2*18 + 5) % 11 = 8

h1(41) = (2*41 + 5) % 11 = 10

h1(37) = (2*37 + 5) % 11 = 2
```

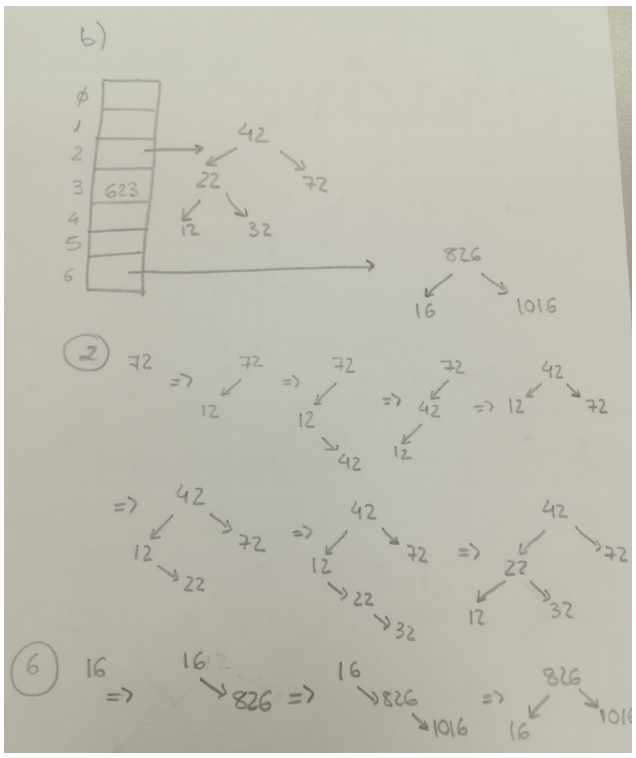
| | | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 37 | |
| 3 | 10 | |
| 4 | 49 | |
| 5 | | |
| 6 | 39 | |
| 7 | 6 | |
| 8 | 18 | |
| 9 | 32 | |
| 10 | 41 | |

**(Febrero 2013) Supongamos que en una Tabla Hash abierta hacemos uso en cada cubeta de árboles AVL en lugar de listas.**

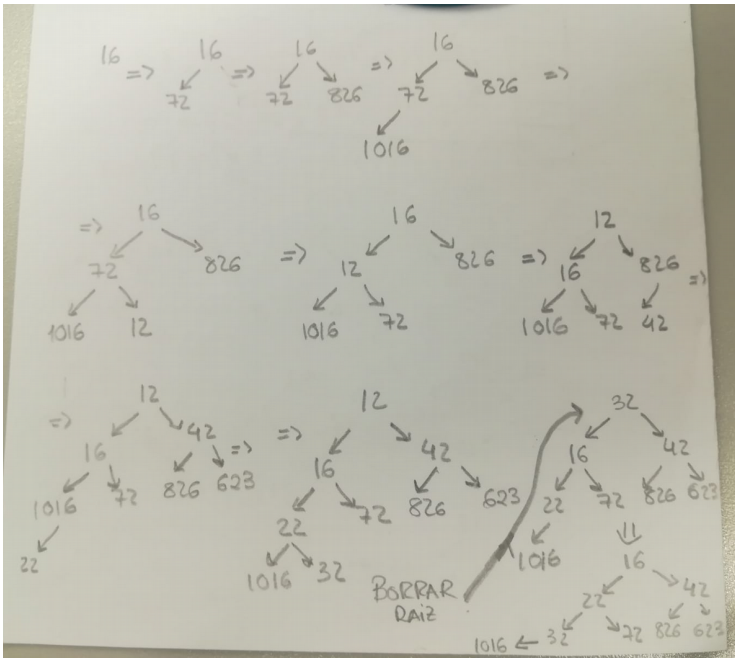
**a) ¿Cuál es la eficiencia (en el caso peor) de la función buscar?**

La eficiencia sería log (m), siendo m el número de nodos de un AVL de una cubeta.

**b) Mostrar la tabla resultante de insertar los elementos {16, 72, 826, 1016, 12, 42, 623, 22, 32}**




**c) Construir un APO a partir del anterior conjunto de claves y a continuación elimina el elemento más pequeño.**



```
template <class K, class V>
class TablaHashAbierta{
	private:
		vector<list<pair<K,V>>> th;
		int fh(K k)const;

	public:
		bool buscar(const K &clave) const;
		bool insertar(const K &clave, const &V valor);
}

bool TablaHashAbierta::buscar(const K &clave)const{
	bool esta = false;
	int pos = fh(clave);
	list<pair<k,v>>::const_iterator it;
	it = th[pos].begin();

	while((it != th[pos].end()) && !esta)
		if((*it).first == clave)
			esta = true;
		++it;

	return esta;
}

bool Insertar(const K &clave, const V &valor){
	bool esta = !Buscar(clave);

	if(!esta)
		int pos = fh(clave);
		th[pos].push_back(make_pair<clave,valor>);

	return esta;
}
```