

int n, j; int r=1; int x=0;

do {

 j=1;

 while (j <= n)

 j = j * 2;

 r++;

 i++;

$O(n \log_2 n)$

} while (i <= n)

—————

int n, j; int r=2; int x=0;

do {

 j=1;

 while (j <= r)

 j = j * 2;

 r++;

 i++;

} while (i <= n);

—————

$O(\sum_{i=1}^n \log_2 i) = O(\log_2 2 + \log_2 3 + \dots + \log_2 n) =$

$= O(\log_2 (1.2.3 \dots n)) = O(\log_2 n!) \leq O(n \log_2 n)$

void eliminar (vector L, int x)

{

int aux, p; ~~x = find(L);~~

for (p = primero(L); p != ~~find(L);~~ ;)

{

aux = elemento (p, L);

if (aux == x)

borrar (p, L)

else p++

,

}

for (k = 1; k <= n ; k *= 2)

for (j = 1; j <= k ; j++)

sum2++

$$\sum_{k=1}^{\log_2 n} \sum_{j=1}^k 1 = \sum_{k=1}^{\log_2 n} k = \sum_{2^i=1}^{\log_2 n} 2^i = \frac{2^{\log_2 n} \cdot 2 - 1}{2 - 1}$$

(i = 0)

$\approx 2n - 1 \longrightarrow O(n)$

for (k = 1; k <= n ; k *= 2)

for (j = 1; j <= n ; j++)

sum2++

$$\sum_{k=1}^{\log_2 n} \sum_{j=1}^k 1 = \sum_{k=1}^{\log_2 n} n = n \log_2 n$$

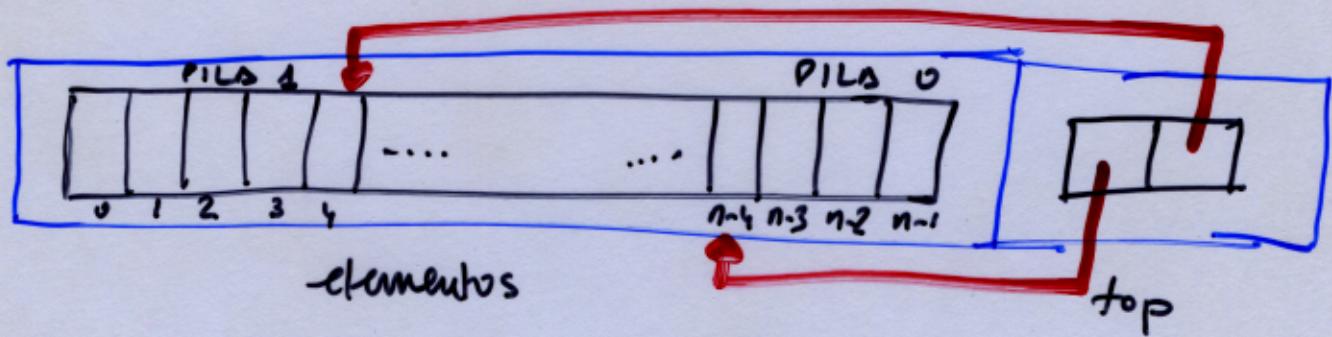
$O(n \log_2 n)$

```
stack<int> P(N);
for (i=1; i ≤ N; ++i)
    if (TEST(i))
        cout << i;
    else p.push(i);
while (!p.empty())
{
    i = p.top();
    p.pop();
    cout << i;
```

~~32~~

Detectar si una salida de valores enteros es consecuencia o no de aplicar el código anterior

Pila doble



private:

$T^* \text{ elementos};$

int top[2];

template <class T>

void Pila<T>::pop (int indicepila)

}

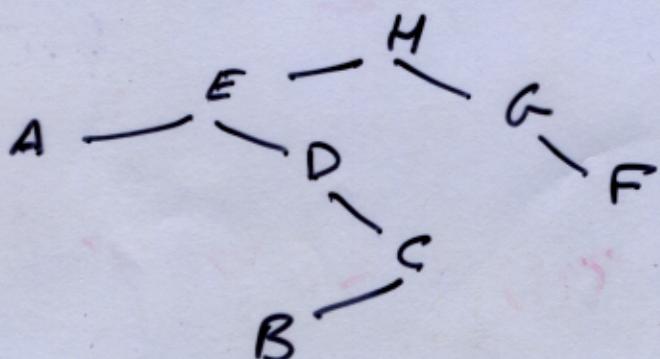
top[indicepila] = top[indicepila]

- (indicepila ? 1 : -1);

5

Post: A B C D E F G H

In: A E D B C H G F



Usando el TDS `list<int>`, construir una función

`void agrupar_elemento (list<int> &entrada, int k)`

que agrupe de forma consecutiva en la lista de entrada todos los apariciones del elemento k en la lista, a partir de la primera ocurrancia. P. ej.

- si entrada = $\{1, 3, 4, 1, 4\}$ y $k=1$ entonces
entrada = $\{1, 1, 3, 4, 4\}$
- si entrada = $\{3, 1, 4, 1, 4, 1, 1\}$ y $k=1$
entonces entrada = $\{3, 1, 1, 1, 4, 4\}$

`void agrupar_elemento (list<int> &entrada, int k)`

```

    list<int> iterator :: p, q;
    p = entrada.find(entrada.begin(), entrada.end(),
                      k),
    q = ++p;
    while (q != entrada.end())
        if (*q == k)
            entrada.insert (p, k),
            q = entrada.erase (q),
        else
            ++q;

```

Algoritmo.

1. buscar primera aparición de k en entrada, y almacena su posición.
2. recorrer entrada buscando apariciones de k
 - 2.1. cuando se encuentre una:
 - 2.1.1. insertar elemento en P
 - 2.1.2. borrar elemento.

void agrupar_elemento (list<int> &entrada,
int k)

{ list<int> iterator: :P, q;

$P =$ entrada. find (entrada. begin ()),

entrada. end (), k);

$q = ++P;$

while ($q \neq$ entrada. end ())

if ($*q = = k$)

{ entrada. insert (P, k);

$q =$ entrada. erase (q);

else $++q;$

Usando el TDR list<T>, construir una función template <class T> void dividir-lista (list<T> & ~~lista~~, T <)

que agrupe en la primera parte de la lista los elementos menores que le y en la segunda los mayores o iguales. Han de usarse iteradores, no se permiten estructuras auxiliares, no se puede modificar el tamaño de la lista y la función ha de tener $O(n)$

Ejemplo.

Entrada = {1, 3, 4, 14, 11, 9, 7, 16, 25, 19, 7, 8, 9}

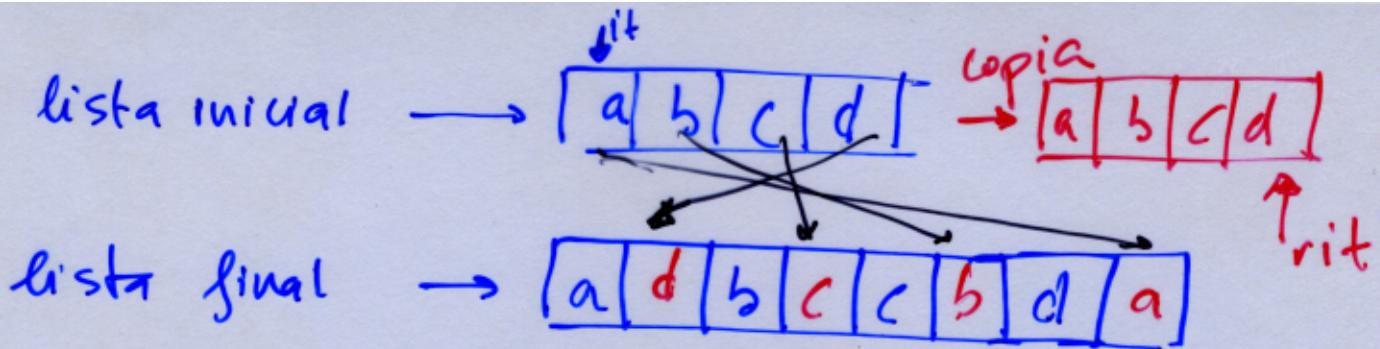
IC = 8

Salida = {1, 3, 4, 7, 3, 9, 11, 16, 25, 19, 14, 8, 9}

typename list<T> :: iterator it = lista.begin();

while (it != lista.end()) \rightarrow q = lista.end();

```
    if (*it > IC)
        lista.push_back(*it);
        it = lista.erase(it);
    else
        ++it;
```



template<class T>

void duplicar (const list<T> & lista_inicial,
list<T> & lista_final)

{

assert (lista_inicial.size() > 0);

list<T> copia_lista (lista_inicial);

typename list<T>:: const_iterator it;

typename list<T>:: reverse_iterator rit;

for (it = lista_inicial.begin(); rit != copia_lista.
rbegin();
it != lista_inicial.end(); rit != copia_lista.
rend(),

++it; ++rit)

{

lista_final.push_back(*it);

lista_final.push_back(*rit);

}

}

Distinar una función que dada una lista de enteros elimine todos aquellos que no sean más grandes que todos los anteriores.

$$\langle 1, 3, 4, 2, 4, 7, 7, 1 \rangle \longrightarrow \langle 1, 3, 4, 7 \rangle$$

```
void subsecuencia (list<int> & l)
{
    list<int>::iterator it = l.begin();
    while (it != l.end())
    {
        list<int>::iterator next = it;
        ++next;
        if (next != l.end() && *it > *next)
            l.erase(next);
        else
            it = next;
    }
}
```

Dada una lista de enteros con elementos repetidos, diseñar (usando el TDA lista) una función que construya a partir de ella una lista ordenada de listas, de forma que en la lista resultado, los elementos iguales, se agrupen en la misma sublista. Ejemplo:

Entrada = {1, 3, 4, 5, 6, 3, 2, 4, 4, 5, 5, 3, 2, 7}

Salida = {{4, 4, 1, 6}, {2}, {3, 3}, {4, 4}, {5, 5, 5, 5}, {6}, {7}}

list < list < int > > final;

list < int > l1, l2;

list < list < int > > iterador; q; list < int > iterator::p;

l1.sort(); q = final.begin();

while (!l1.empty())

{ ~~q = final.begin();~~

p = l1.begin(); elem = *p;

while (*p == elem)

{ l2.insert(l2.begin(), elem);

p = l1.erase(p);

final.insert(q, l2);

l2.clear();

++q;

Dadas 2 colas con elementos repetidos implementar la función:
(y ordenados)

que $\text{queue} < \text{int} >$ multiintersección ($\text{queue} < \text{int} > g_1$,
 $\text{queue} < \text{int} > g_2$)

que calcule la multiintersección de 2 colas g_1 y g_2 y devuelva el resultado en otra cola.

$g_1: [2, 2, 3, 3]$

$g_2: [4, 2, 3, 3, 3, 4]$

multiintersección $[2, 3, 3]$

Disenar una función que dada una lista L devuelva otra lista R conteniendo los elementos repetidos de L . Si no hay elementos repetidos R será la lista vacía.

$L = \{5, 2, 3, 2, 5, 5, 4\}$

$R = \{5, 2\}$

queue<int> multiIntersection (queue<int> q1,
queue<int> q2)

```
{  
    queue<int> q;  
    while (!q1.empty() && !q2.empty()) {  
        if (q1.front() == q2.front()) {  
            q.push(q1.front());  
            q1.pop();  
            q2.pop();  
        }  
        else if (q1.front() < q2.front())  
            q1.pop();  
        else q2.pop();  
    }  
    return q;  
}
```

```
multiset<int> multi_intersection (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;
```

```
}
```

```
template <class T>
list <T> repetidos(list <T> l)
{
    set <T> elem_dif;
    set <T> rep;
    list <T> l_rep;
    for (typename list <T>::iterator it =
        l.begin(); it != l.end(); ++it)
    {
        if (elem_dif.find(*it) == elem_dif.end())
            elem_dif.insert(*it);
        else
            rep.insert(*it);
    }
    for (typename set <T>::iterator it = rep.begin();
        it != rep.end(); ++it)
        l_rep.push_back(*it);
    return l_rep;
}
```

```
int orden (list<int> L)
```

```
    {  
        bool es_ascendente = true,  
        es_descendente = true;
```

```
        list<int>::iterator it1 = L.begin();
```

```
        list<int>::iterator it2 = L.begin();
```

```
        if (L.empty()) return 0;
```

```
        else {
```

```
            ++it2;
```

```
            while ((es_ascendente || !es_descendente)
```

```
                if (it2 != L.end())
```

Función orden que
devuelve 1 si la
lista de entrada

esta ordenada de forma

ascendente de principio

a fin, 2 si lo está

de forma descendente

y 0 si no está

ordenada de

ninguna

forma

```
            if ((*it1) < (*it2))
```

```
                es_descendente = false;
```

```
            else if ((*it1) > (*it2))
```

```
                es_ascendente = false;
```

```
            ++it1;
```

```
            ++it2;
```

```
}
```

```
        if (es_ascendente) return 1;
```

```
        else if (es_descendente) return 2;
```

```
        else return 0;
```

```
}
```

Usando el TDS lista, construir una función que tenga como entrada dos listas y devuelva 1, si la primera está contenida en la segunda (tiene los mismos elementos, consecutivos y en el mismo orden) y 0 en otro caso. P. ej: $l1 = \langle 1, 2, 3 \rangle$ } ①
 $l2 = \langle 0, 1, 2, 3 \rangle$

Pa partir de la

1: secuencia

Generalizar!!!

$l1 = \langle 1, 2, 3 \rangle$ } ②
 $l2 = \langle 1, 2, 2, 3 \rangle$ } ③

bool secuencia (list<int> & l1, list<int> & l2)

↳ list<int> iterador :: p, q;

int x;

$x = l1.\text{front}();$

$p = l1.\text{begin}();$

$q = l2.\text{find}(l2.\text{begin}(), l2.\text{end}(), x);$

while ($p \neq l1.\text{end}()$)

if ($*p \neq *q$)

return false;

else {
++p;

++q;

↳

return true;

5

Se dice que una pila es inversa a una cola cuando el listado de los elementos de la pila coincide con el listado de los de la cola en orden inverso.

Usando las clases `stack` y `queue` de la STL diseñar una función para determinar si una pila y una cola dada son inversas

template <class T>

bool sonInversas (<stack<T>, s, queue<T> q)

{

if (s.size() != q.size()) return false;

else {

stack<T> aux;

while (!q.empty())

{ aux.push (q.front());

q.pop();

}

while (!aux.empty())

{ if (aux.top() != s.top())

return false;

aux.pop();

s.pop();

}

return true;

}

```

void Pila::anitar()
{
    assert (nElem > 0);  $O(1)$ 
    nElem--;
    if (nElem <= reservados/4) Resize (reservados/2);
}

```

- Comportamiento de operaciones consecutivas poner/quitar cuando el vector está completo:

* Al poner un elemento con el vector lleno, se redimensiona al doble. Si inmediatamente después quitamos un elemento, el vector tendría más de la mitad del tamaño libre y volvería a redimensionarse a la mitad.

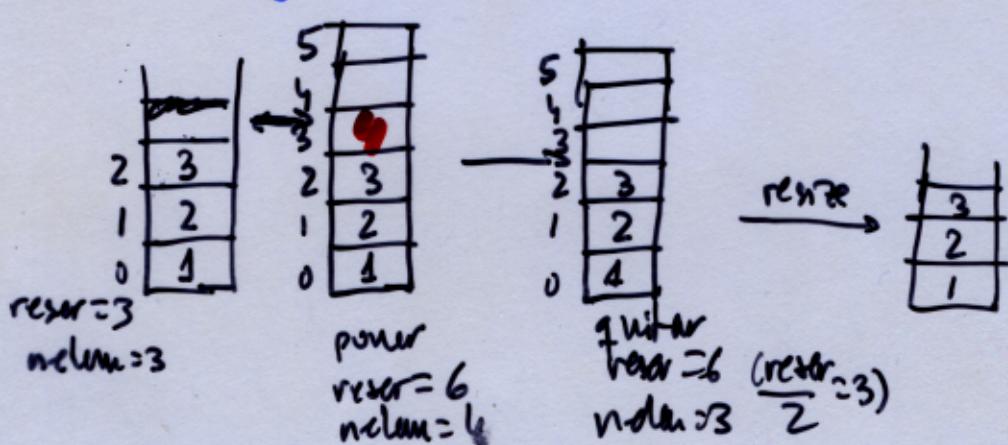
* Si continuamos con operaciones poner - quitar - poner - quitar - etc., cada una tendría un costo $O(n)$.

```
void Pila::Poner (T c)
```

```

    if (nElem == reservados) Resize (2 * reservados);
    datos [nElem] = c;
    nElem--;
}

```



y vuelve a
empatarse

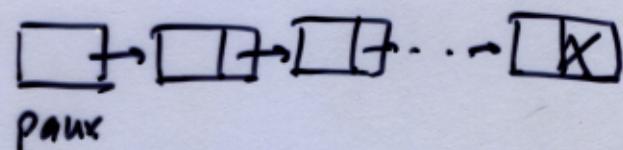
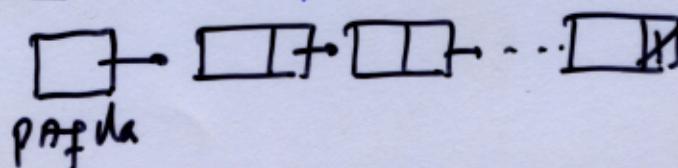
Pila & Pila:: operator = (const Pila & p)

↳ Pila paux(p);
CeldaPila *aux;
aux = this → primera;
this → primera = paux.primera;
paux.primera = aux;
return *this;

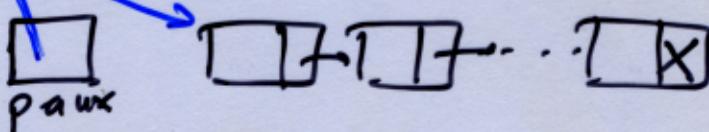
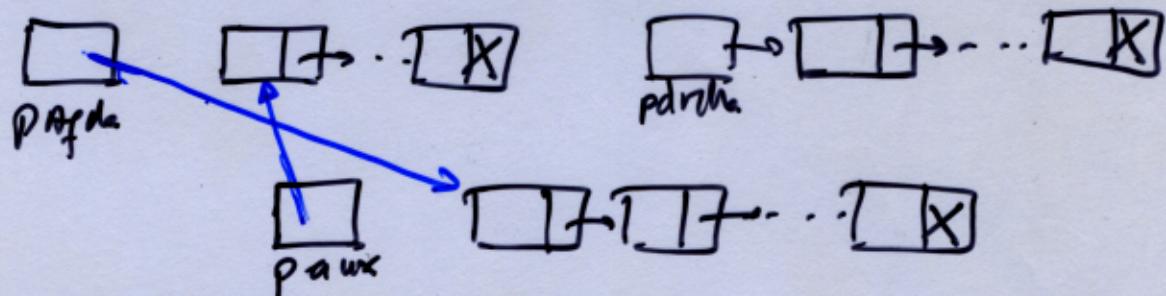
↳

hacemos pizqda = pdrecha (asignacion de 2 pilas)

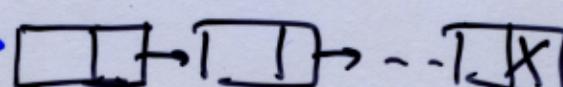
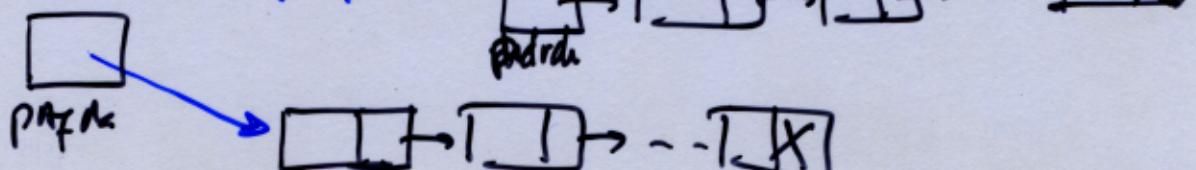
Paso 1 (crea paux como copia de la pila dcha)



Paso 2 (Intercambia los contenidos de paux y pizqda)

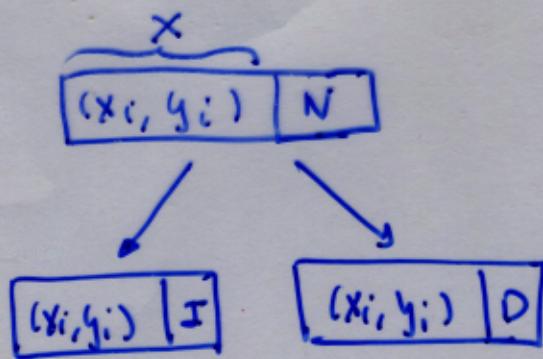


Paso 3 (Destruye paux)



(Al destruirse paux se destruyen los id's de la pila pizqda y pdrecha queda con un conjunto de celdas idéntico a pdrecha)

ARBOL CARTESIANO

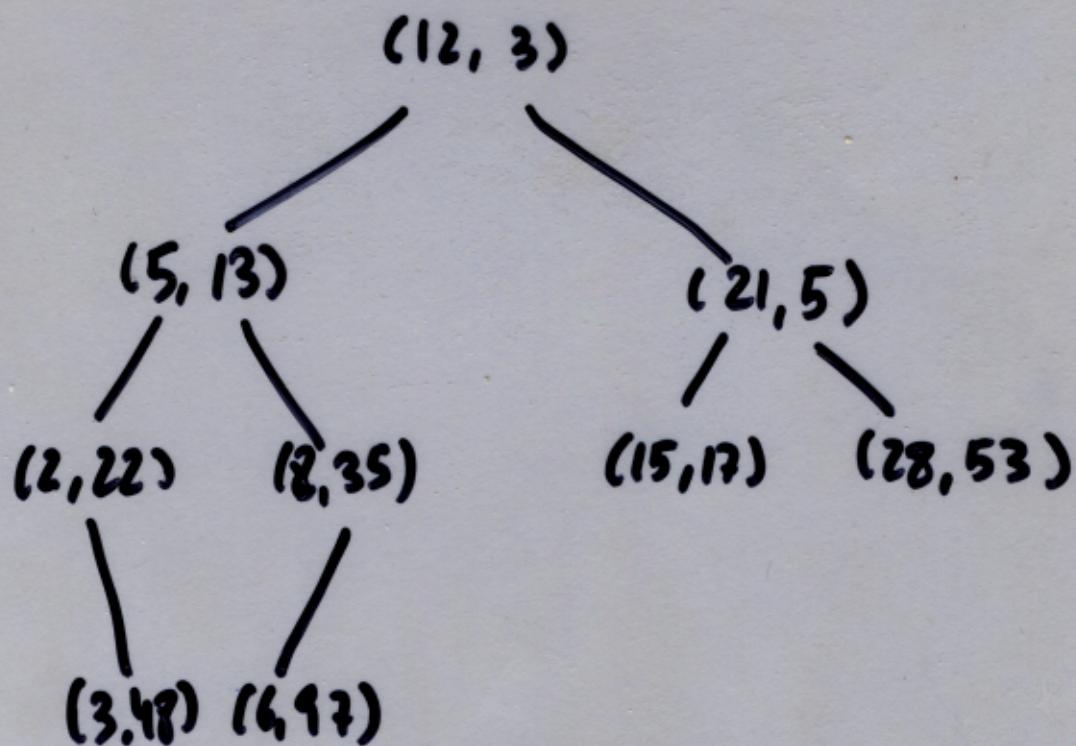


$x_S < x_N$ & & $x_N < x_D$ \rightarrow "ARB"

$y_N < y_I$ & & $y_N < y_D$ \rightarrow "APD"

(8, 35), (21, 5), (15, 17), (2, 22), (12, 3)

(28, 53), (13, 48), (6, 97), (5, 13)



Usando el TDA ABR construir una función que tenga como entrada un $ABR <int>$ y que devuelva el número de nodos con un valor de etiqueta dentro de un intervalo $[a, b]$ a, b enteros

int contar (int, a, b, ABR <int> T)

{

if ($T.\text{raiz}() == 0$) return

else if ($T.\text{etiqueta}(T.\text{raiz}()) > b$)

return contar (a, b, T.\text{izqda}(T.\text{raiz}(), T))

else if ($T.\text{etiqueta}(T.\text{raiz}()) < a$)

return contar (a, b,

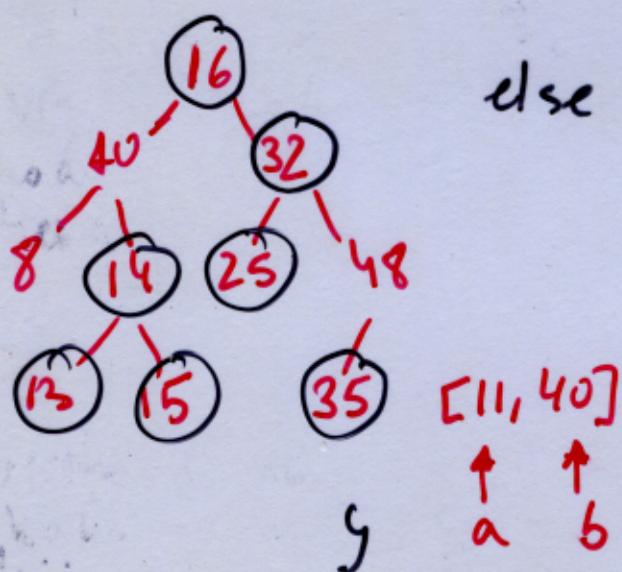
T.\text{drcha}(T.\text{raiz}(), T))

else return $1 + \text{contar}(a, b,$

T.\text{izqda}(T.\text{raiz}(), T))

+ contar (a, b,

T.\text{drcha}(T.\text{raiz}(), T)),



Construir una función que devuelva para un ABRS,
el nivel del nodo con mayor valor de la clave

int nivel (ABRS <T> A)

↳ Nodo u;

n = A.raiz();

contador = 0;

while (n != nodonulo)

↳ n = A.dcha(n),

contador++;

↳

return contador;

}

Sea A un árbol binario con n nodos. Se define el ancestro común más cercano (AMC) entre 2 nodos r y w como el ancestro de mayor profundidad que tiene tanto a r 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 2 nodos r y w y como salida el $AMC(r, w)$.

nodo AMC (ArbolBinario AT , r ab, nodo r , nodo w)

{

vector <nodo> rnodos;

while ($r \rightarrow \text{padre}() \neq 0$) {

rnodos.insert ($r \rightarrow \text{padre}()$);

$r = r \rightarrow \text{padre}()$;

}{

bool find = false;

vector <nodo>::iterator it;

while ($find \neq \text{true} \& w \rightarrow \text{padre}() \neq 0$) {

$it = rnodos.find (w \rightarrow \text{padre}())$;

if ($it \neq rnodos.end()$)

$find = \text{true}$;

else $w = w \rightarrow \text{padre}()$;

}{ if ($find$) return *it;

else return nodo();

}

Algoritmo

ArbolBinario <int>:: Nodo AMI (Const ArbolBinario
liat > & q,

Const ArbolBinario <int>:: Nodo v,

Const ArbolBinario <int>:: Nodo w)

}

ArbolBinario <int>:: Nodo p;

for (p = q. padre (v); p != nodo_nulo; p = a. padre (p))

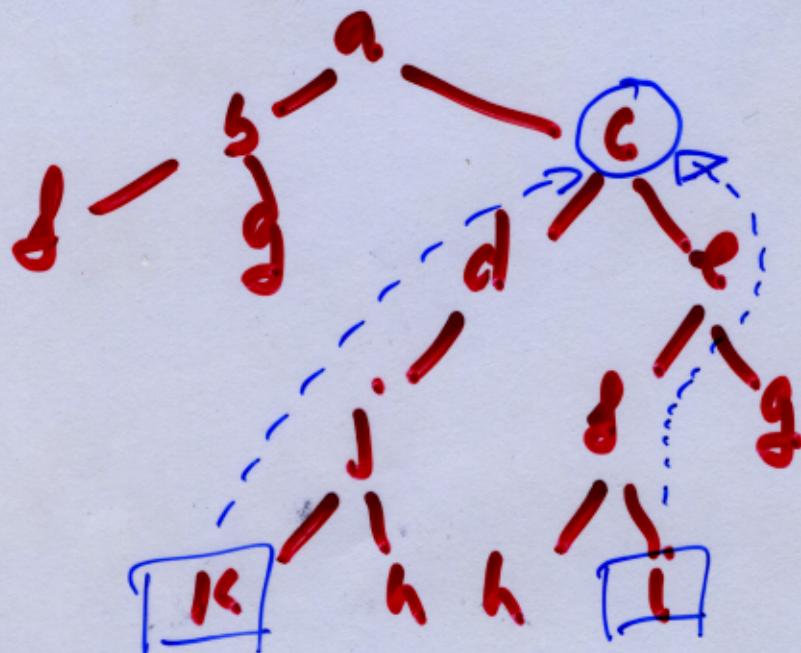
while (w != nodo_nulo)

if (p == q. padre (w))

return q. padre (w);

else w = a. padre (w);

}



Dado un arbol 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 (biutree<int> &ab, int k,
biutree<int>::nodo n)

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

Pre: 4, 9, 24, 33, 21, 74, 63

0	1	2	3	4	5	6
4	9					

0	1	2	3	4	5	6
4	9		24			

0	1	2	3	4	5	6
4	9		24	33		

0	1	2	3	4	5	6
4	9	21	24	33		

0	1	2	3	4	5	6
4	9	21	24	33	74	

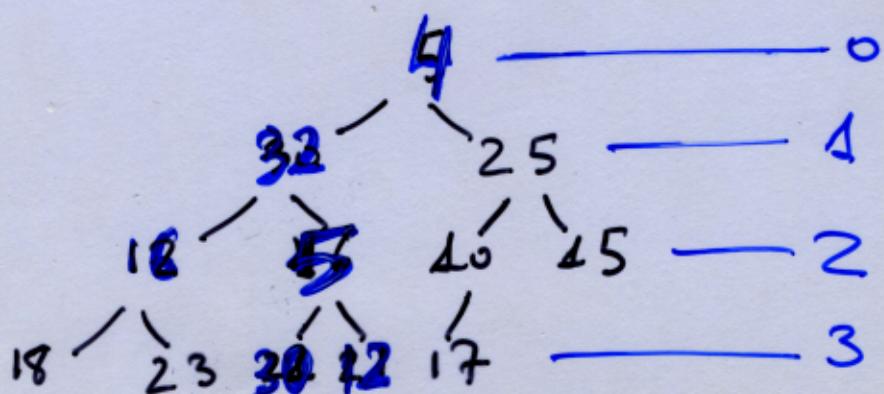
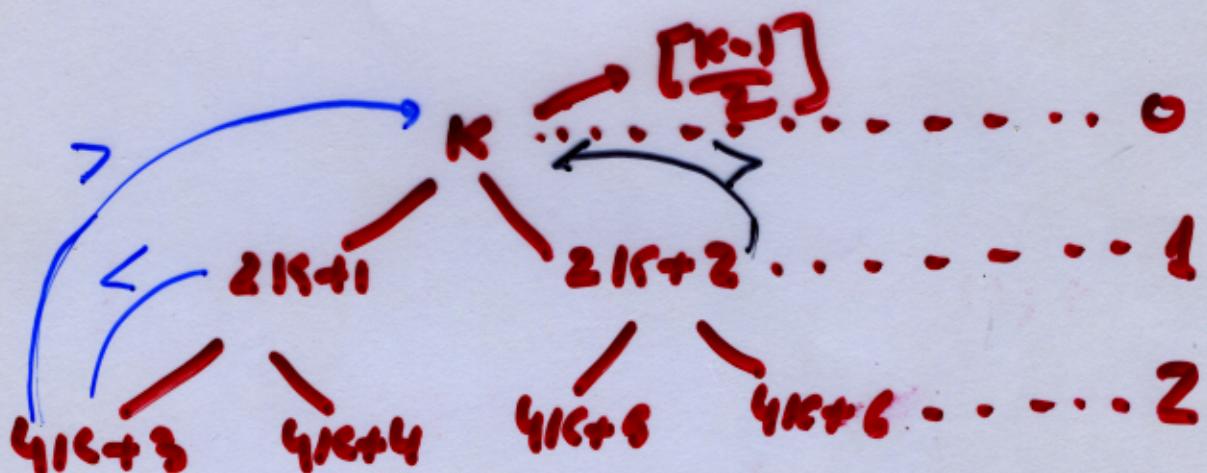
0	1	2	3	4	5	6
4	9	21	24	33	74	63

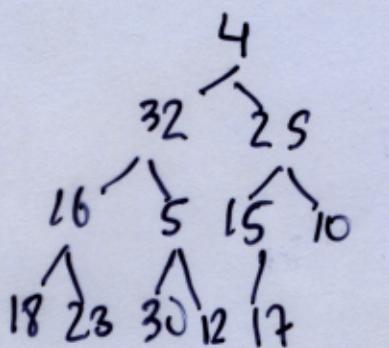
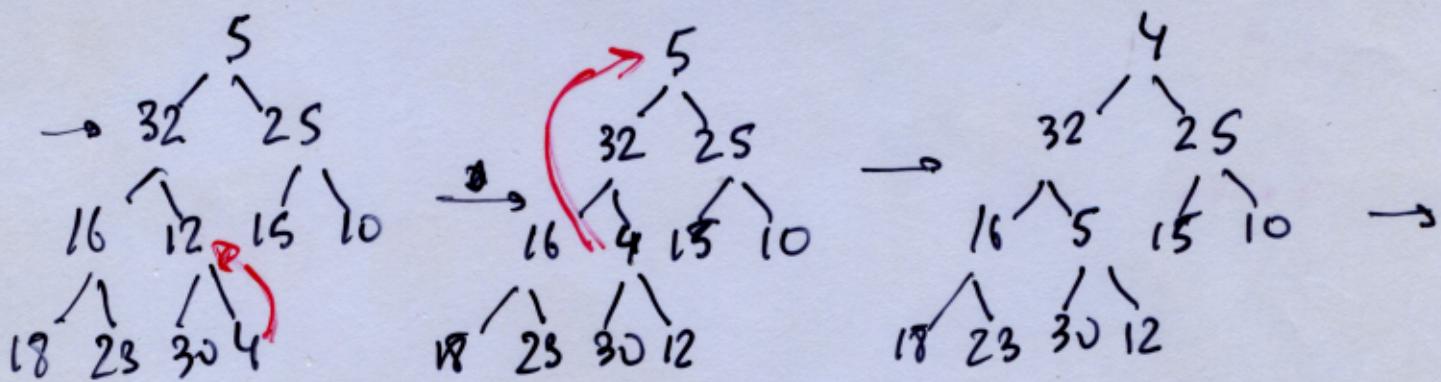
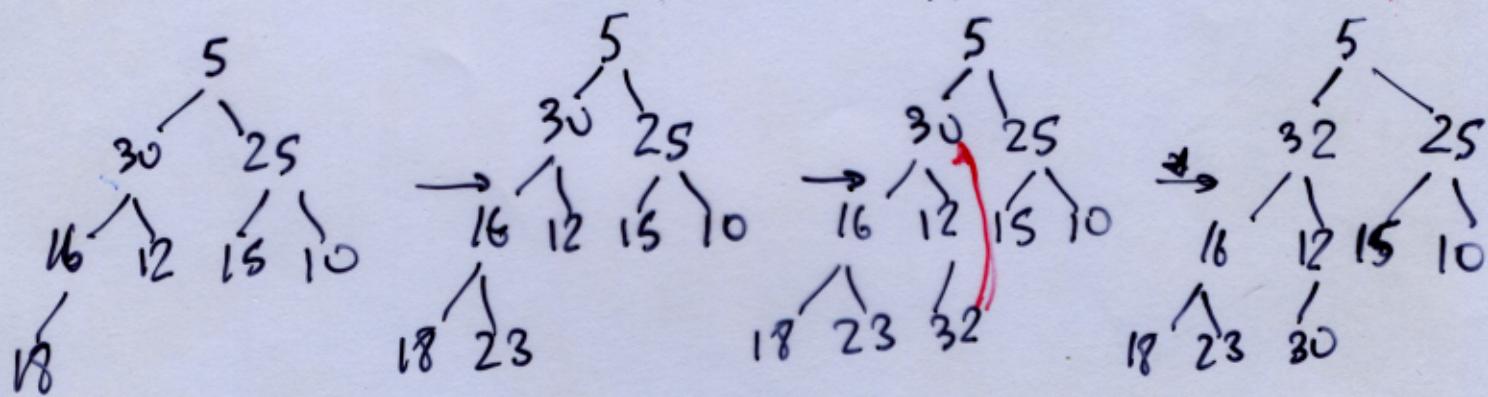
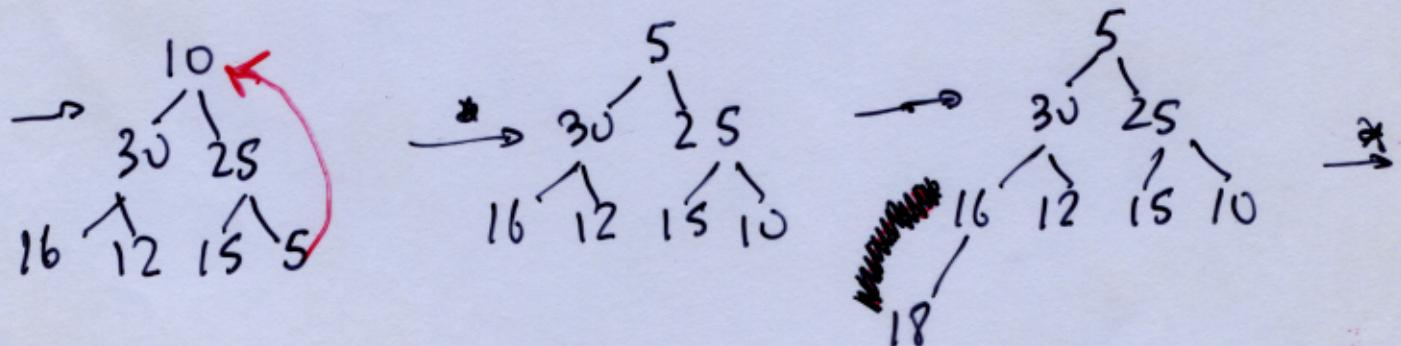
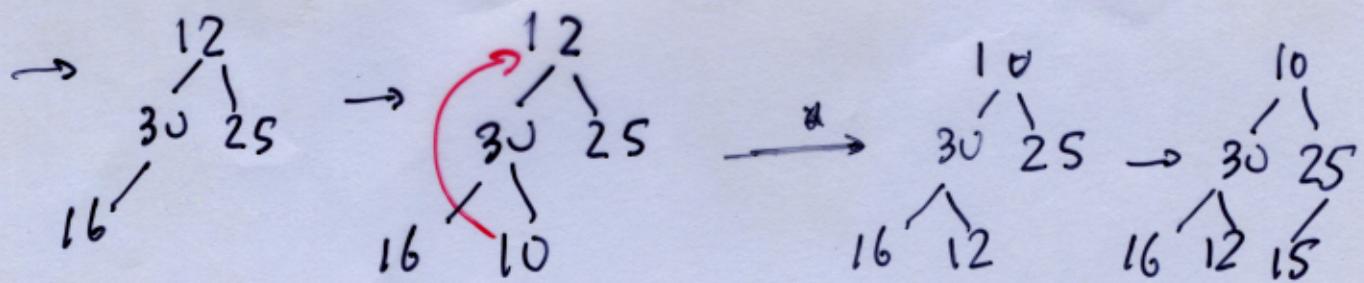
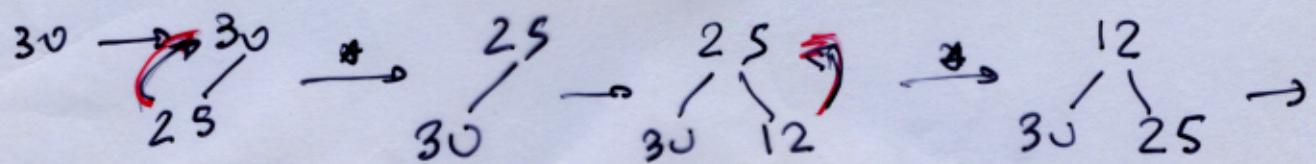
4
9
21
24
33
74
63

Post: 24, 33, 9, 74, 63, 21, 4

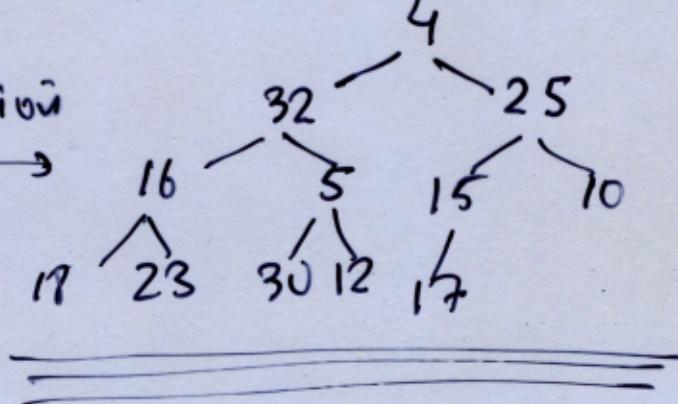
Un heap-doble es una estructura que permite realizar las operaciones eliminar-minimo y eliminar-maximo en $O(\log_2 n)$. Tiene la propiedad de que si nodo i a profundidad par tiene una clave menor que la del padre y mayor que la del abuelo (cuando existen) y si nodo i a profundidad impar tiene una clave mayor que la del padre y menor que la del abuelo (cuando existen) con las hojas empujadas a la raíz.

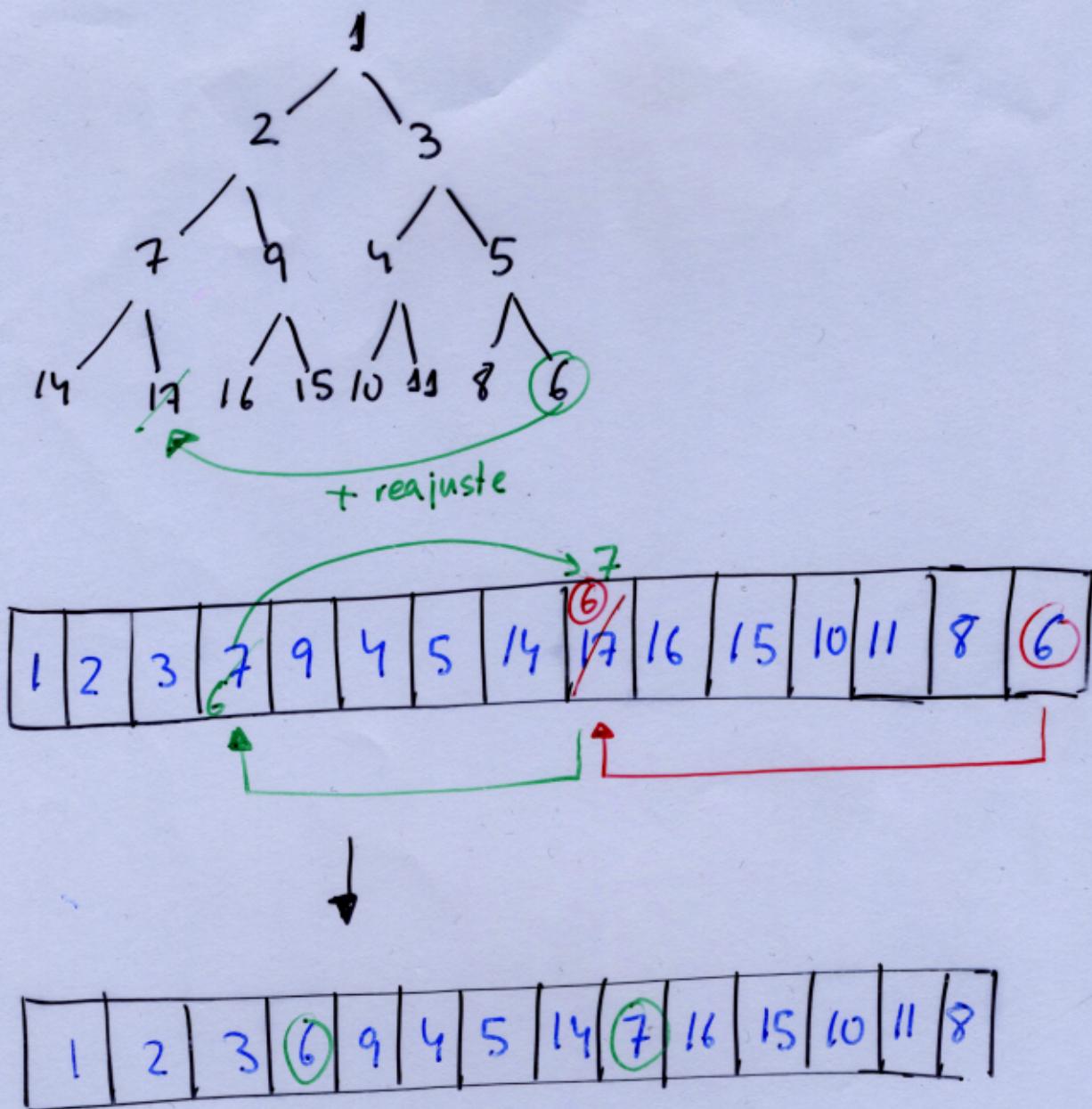
Algoritmo para insertar una clave. Optar por para crear un heap-doble con $\{30, 25, 12, 16, 10, 15, 5, 18, 23, 32, 4, 17\}$





Solución

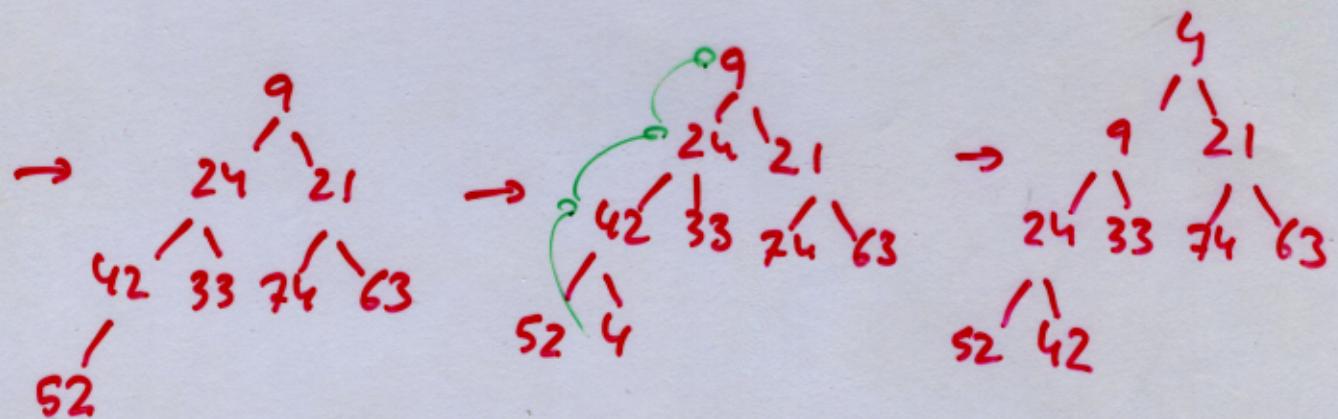
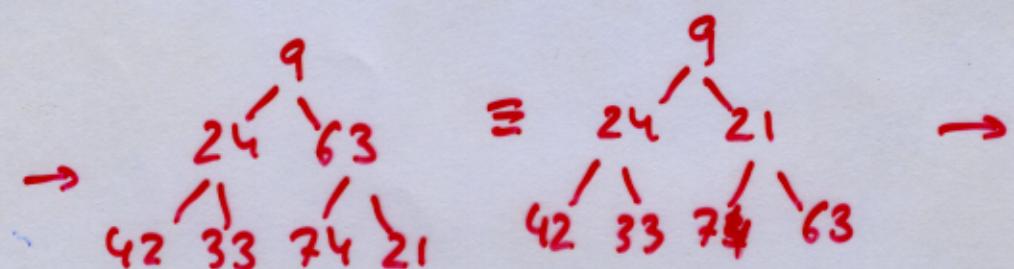
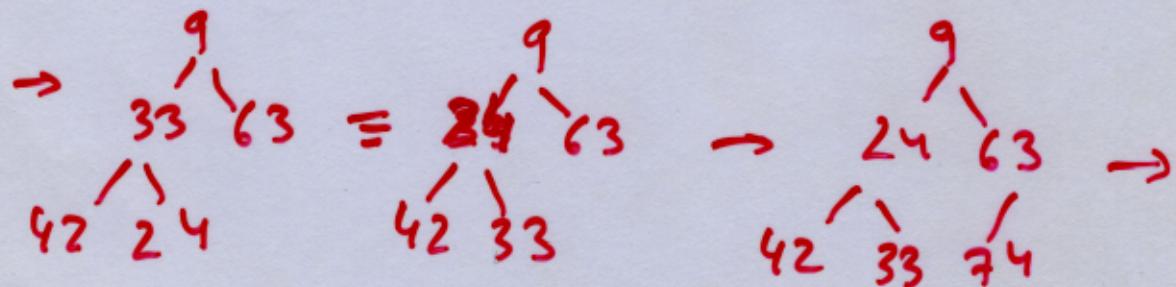
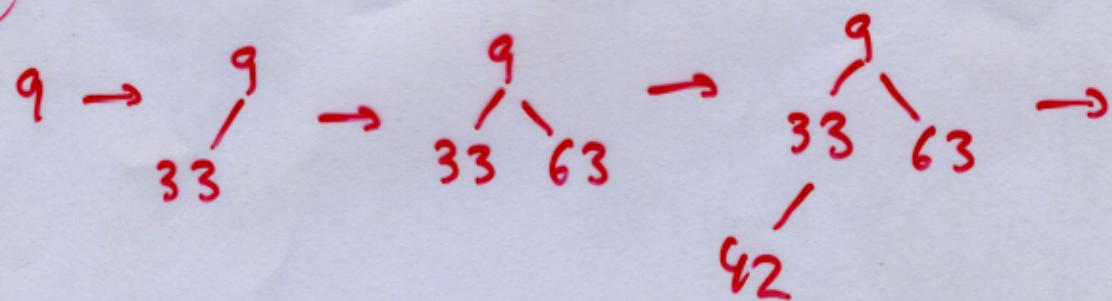




1. Recorrido por niveles (cola de nodos), buscando el máximo
2. "borrar el máximo" sustituyéndolo por el último elemento del último nivel
3. Reajustar el Apomín

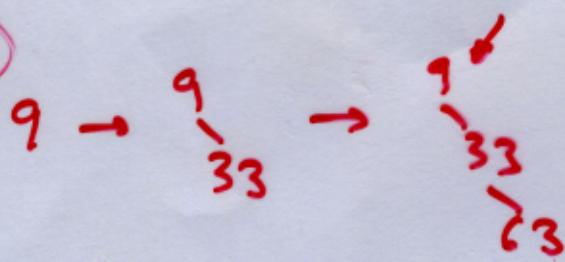
{9, 33, 63, 42, 24, 74, 21, 52, 43}

(APQ)

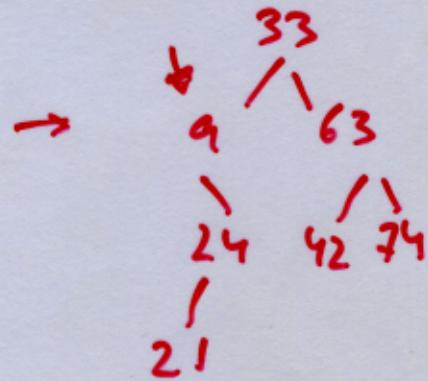
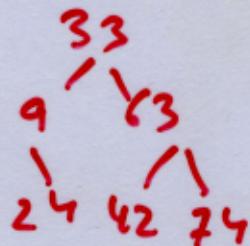
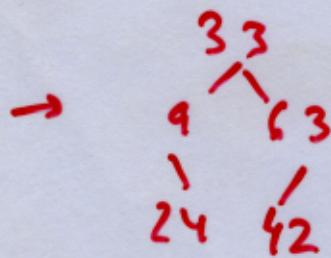
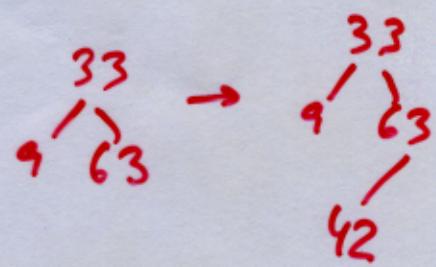


$\{9, 33, 63, 42, 24, 74, 21, 52, 4\}$

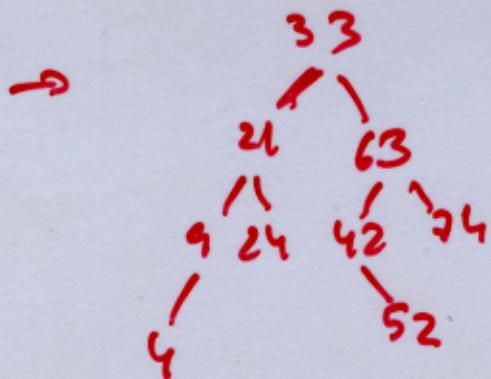
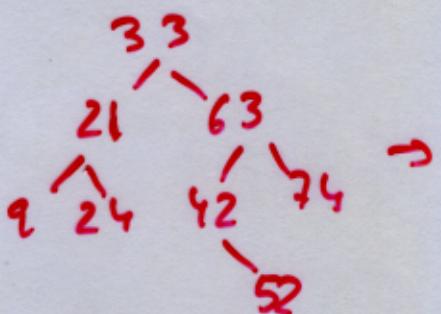
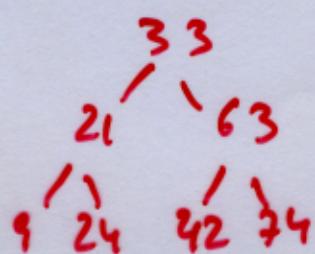
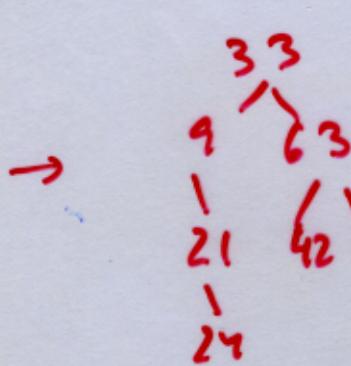
AVL



$R_{SI}(9)$



$R_{OI}(9)$

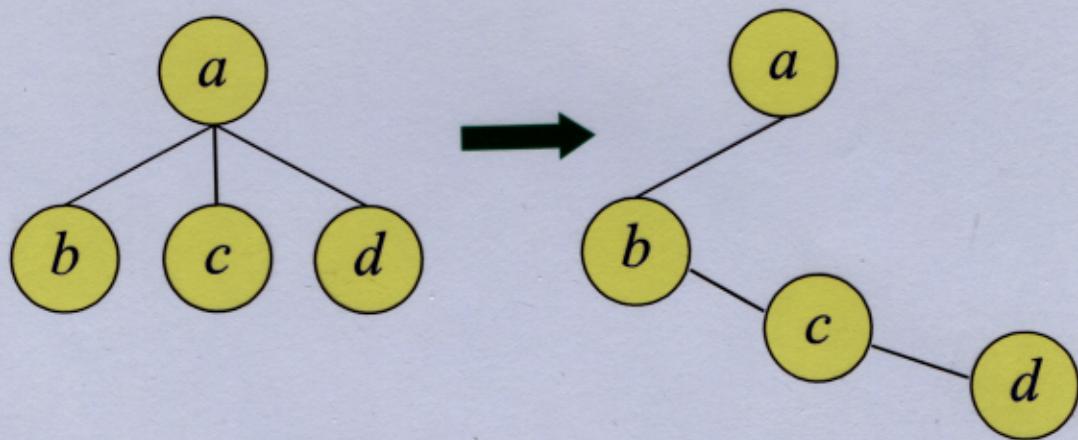


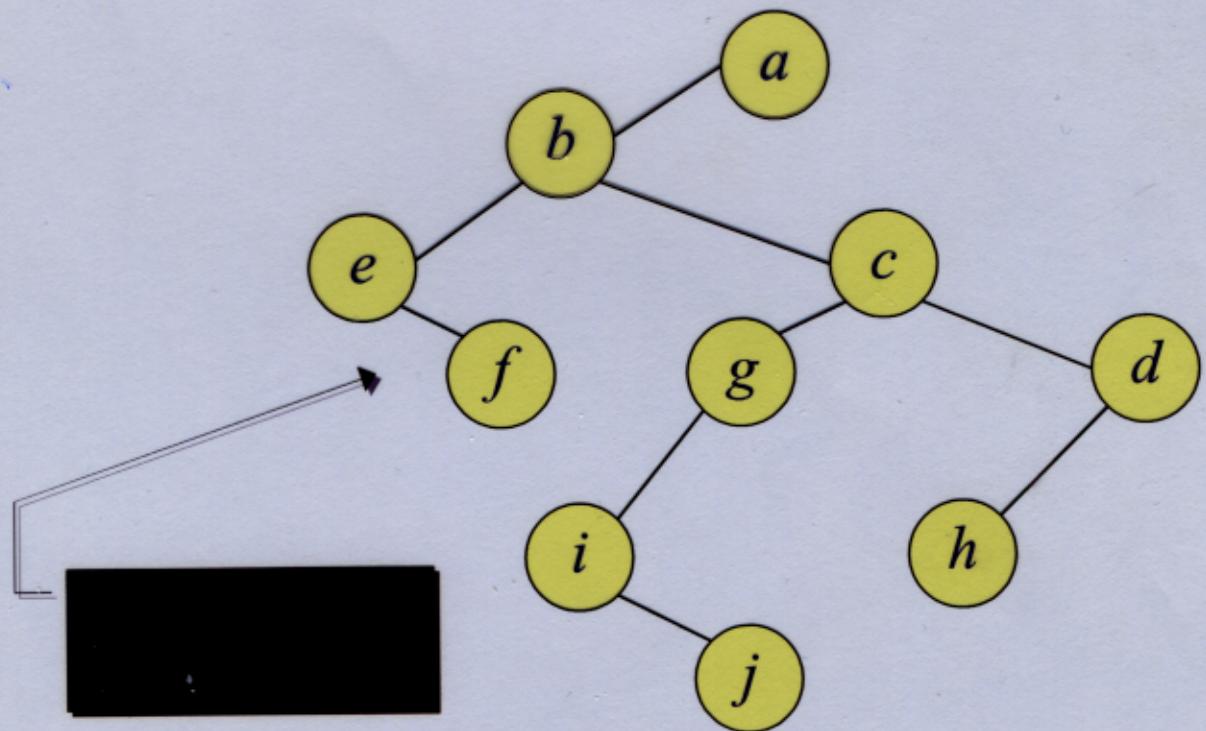
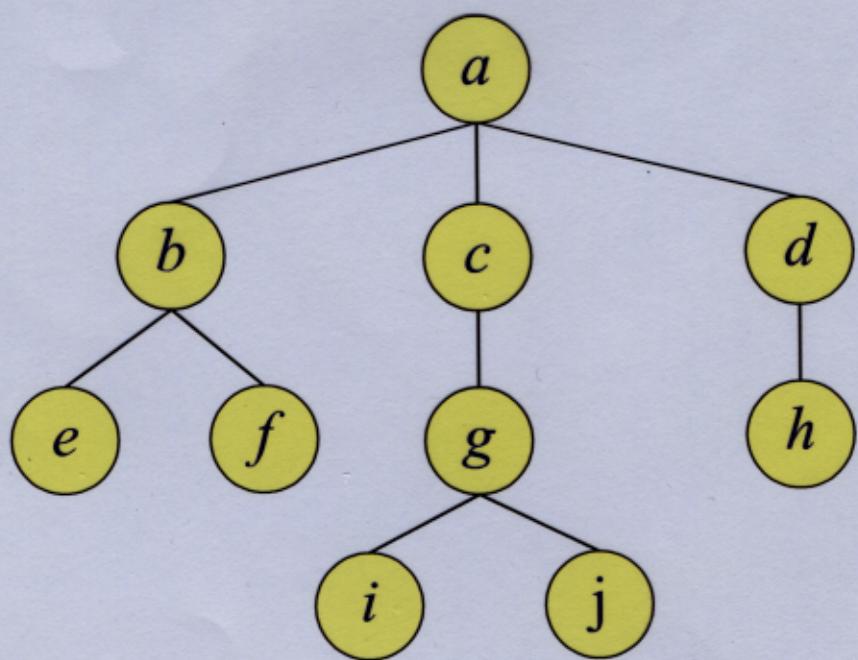
Representación de árbol n-arios: *Basada en nodos binarios*

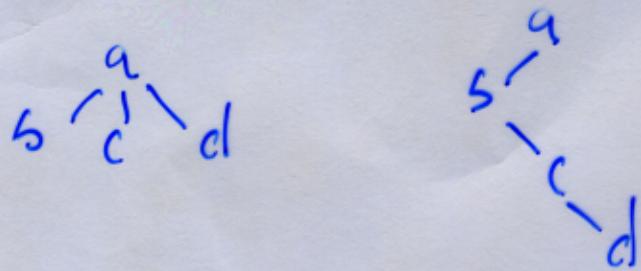
Los nodos del árbol n-ario se representan mediante nodos binarios.

Para representar el árbol, los nodos binarios se enlazan de la forma siguiente:

- primer hijo a la izquierda (del padre)
- hermanos a la derecha (del hermano anterior)







Pre: $\boxed{a, b, c, d} \rightarrow \boxed{a, b, c, d}$

In: $b, a, c, d \rightarrow \boxed{b, c, d, a}$

Post: $\boxed{b, c, d, a} \rightarrow d, c, b, a$

Pre (General) = Pre (binario)

Post (General) = In (binario)

bool inclusion (const conjunto & (1),
const conjunto & (2))

{
for (wust &B33; wust iterator p = (1).begin();
p != (1).end(); ++p)

if !(2).pertenece (*p) return false;
return true;}

}

¿ n diferentes con las claves $1, 2, \dots, n-1, n$?

$n=1$

(1)

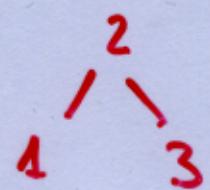
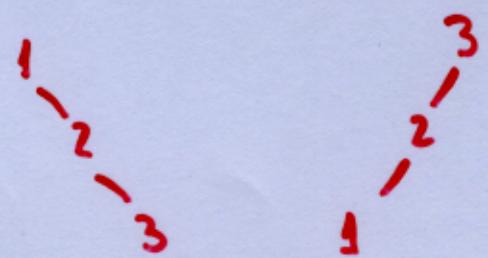
1

$n=2$

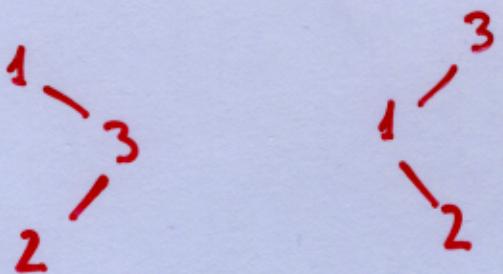


(2)

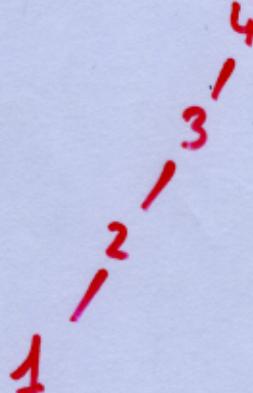
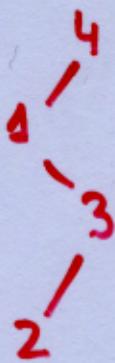
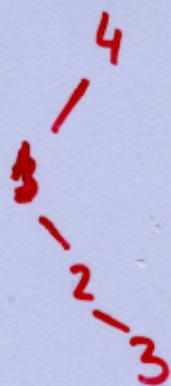
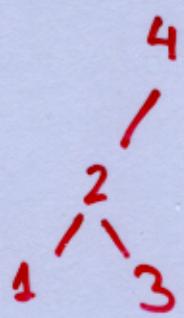
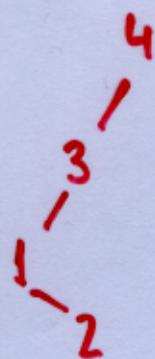
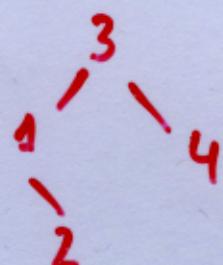
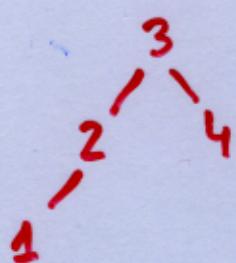
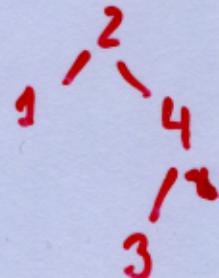
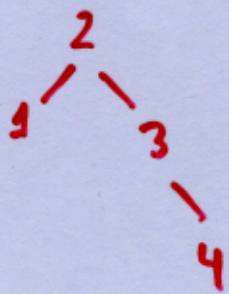
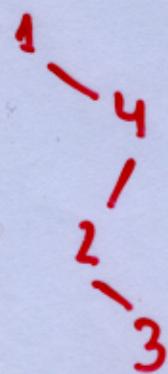
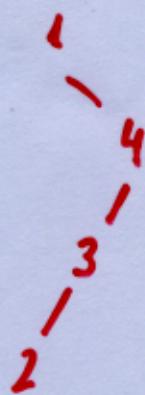
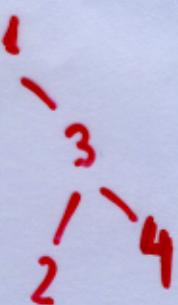
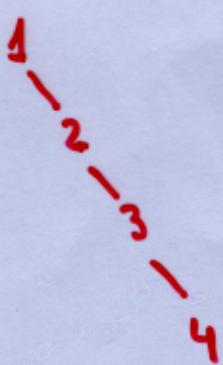
$n=3$



(5)



$u=4$

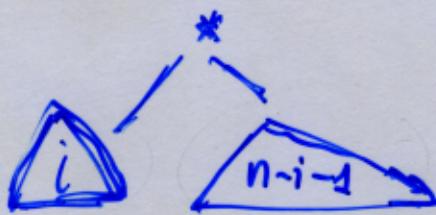


(14)

¿Cuántos para un n arbitrario?

n : $\Delta B B$ diferentes con las claves $\{1, 2, \dots, n\}$

$$\Delta(n) = \sum_{i=0}^{n-1} A(i) * A(n-i-1)$$



$$A(0) = 1, A(1) = 1, A(2) = 2$$

$$A(3) = A(0) * \Delta(2) + A(1) * \Delta(1) + A(2) * \Delta(0) = 5$$

$$A(4) = A(0) * \Delta(3) + A(1) * \Delta(2) + A(2) * \Delta(1) + A(3) * \Delta(0) = 14$$

$$\Delta(5) = A(0) * \Delta(4) + A(1) * \Delta(3) + A(2) * \Delta(2) + A(3) * \Delta(1) + A(4) * \Delta(0) = 42$$

$$\begin{aligned} \Delta(6) = & A(0) * \Delta(5) + A(1) * \Delta(4) + A(2) * \Delta(3) + A(3) * \Delta(2) + \\ & + A(4) * (A(1) + A(5) * \Delta(0)) = 132 \end{aligned}$$

$$\Delta(3) = \sum_{i=0}^2 \Delta(i) * A(3-i-1)$$

$$\boxed{\Delta(7) = 429}$$

$$\Delta(4) = \sum_{i=0}^3 \Delta(i) * A(4-i-1)$$

$$\Delta(5) = \sum_{i=0}^4 \Delta(i) * A(5-i-1)$$

$$\Delta(6) = \sum_{i=0}^5 \Delta(i) * A(6-i-1)$$

Construir un iterador para una clase diccionario
cuya definición privada es:

~~template <class T, class V>~~

class diccionario {

private:

list <data <T, V> > datos;

};

};

con data definido como:

template <class T, class V>

struct data {

T clave;

list <V> info_asoci;

};

Diccionario <string, string> D;

Clave { Programa

- Plan, proyecto o declaracion de algo que se plantea hacer
- Distribucion de materias de un curso
- Anuncio del reporto y anuncio tecnico de un espectaculo

```
class iterator {
private:
    typename list<data<T,U>>::  
    iterator punt;
public:
    iterator() {};
    iterator & operator++() {
        punt++;
        return *this;
    }
    iterator & operator--() {
        punt--;
        return *this;
    }
    bool operator==(const iterator &it) {
        return it.punt == punt;
    }
    bool operator!=(const iterator &it) {
        return it.punt != punt;
    }
    data<T,U> & operator*() {
        return *punt;
    }
    friend class Diccionario;
    friend class const_iterator;
```

```
iterator begin() {
    iterator it;
    it.punt = datos.begin();
    return it;
}
```

```
iterator end() {
    iterator it;
    it.punt = datos.end();
    return it;
}
```

ostream & operator << (ostream & os, const Diccionario<string, string> & D) {

Diccionario<string, string> :: const_iterator it;
for (it = D.begin(); it != D.end(); ++it) {
 list<string> :: const_iterator it_s;

os << endl << (*it).clave << endl << "información asociada: " << endl;

for (it_s = (*it).info_asoci.begin(); it_s != (*it).info_asoci.end(); ++it_s) {
 os << (*it_s) << endl;

os << "fin de informacion" << endl;
return os;

Disponemos del TDRs matriz de enteros (se almacenan los datos por filas) y se quiere definir un iterador que itere por columnas sobre los elementos pares de la matriz. Para ello hay que implementar los operadores `++` y `*`, así como las funciones `begin()` y `end()` en la clase matriz. P. ej. si la matriz M ~~es~~:

$$\begin{bmatrix} 5 & 4 & 3 \\ 2 & 4 & 2 \\ 9 & 0 & 2 \\ 8 & 9 & 1 \end{bmatrix} \quad \downarrow$$

ejecutando el siguiente código

Matriz M;

⋮⋮⋮

Matriz::iterator its

for (it = M.begin(); it != M.end(); ++it)

cout << *it;

se imprimirá sobre la salida estandar:

2, 8, 4, 0, 2, 2

```
class Matrix {
    int ** datos;
    int nf, nc;
}

public:
    class iterator {
        private:
            int * d;
        public:
            int & operator*() const;
            iterator & operator++();
        };
    };

    iterator end();
    iterator begin();
}

Matrix::iterator Matrix::begin() {
    Matrix::iterator it;
    it.d = & datos[0][0];
    if (*it.d) % 2 == 0 return it;
    else ++it; // pasa al siguiente par
    return it;
}
```

Matrix: iterator & operator $\text{operator } ++()$:

```
int f = (d - & (datos[0][0])) / nc; // fila donde
// apunta
int c = (d - & (datos[0][0])) % nc; // columna donde
// apunta
while (true) {
    if (f >= nf - 1 && c >= nc - 1) // ya hay mas
        // elementos
        {
            d = & (datos[nf - 1][nc - 1]);
            return *this; // devolvemos end
        }
    else {
        if (f < nf - 1) {
            f = f + 1; // en la misma columna, el siguiente
            d = & (datos[f][cc]);
            if (*d % 2 == 0) return *this;
        }
        else { // hemos recorrido toda la columna y
            f = 0; // pasamos a la siguiente columna
            c = c + 1;
            d = & (datos[f][c]);
            if (*d % 2 == 0) return *this;
        }
    }
}
```

friend class Matrix;

construir una clase guia_de_teléfonos que de
soporte al manejo de información del tipo:

Susana Amiba 958665544

Antonio Santillan 653453111

Carlos Pineda 606112233

Carolina Fuentes 958334455

Fernando Pum 654234567

: : : = : (sin nombres repetidos)

Con funciones para:

- Devolver un tipo asociado a un nombre
- Insertar un nuevo elemento en la guia
- borrar un elemento de la guia
- Imprimir la guia

```
#include <map>
#include <iostream>
#include <string>
```

```
class Alia_Tlf {
```

```
private:
```

```
map<string, string> datos;
```

```
public:
```

```
string & operator[] (const string & nombre)
```

```
    { return datos[nombre]; }
```

```
}
```

```
pair<map<string, string>::iterator, bool>
```

```
insert (pair<string, string> p)
```

```
{
```

```
pair<map<string, string>::iterator, bool> ret;
```

```
ret = datos.insert(p);
```

```
return ret;
```

```
}
```

```
void borrow (const string & number)
```

```
}
```

```
map <string, string>:: iterator it_low =  
    datos.lower_bound (number),
```

```
map <string, string>:: iterator it_upper =  
    datos.upper_bound (number),
```

```
datos.erase (it_low, it_upper);
```

```
}
```

```
friend ostream & operator << (ostream & os,  
                                GUIA_TIFF & g)
```

```
{
```

```
map <string, string>:: iterator it;
```

```
for (it = g.datos.begin();
```

```
    it != g.datos.end(); ++it) {
```

```
    os << it->first << "\t" <<
```

```
    it->second << endl;
```

```
}
```

```
return os;
```

```
}
```

```
class iterator {
```

```
private:
```

```
map<string, string>::iterator it;
```

```
public:
```

```
iterator& operator++() {
```

```
    ++it;
```

```
}
```

```
iterator& operator--() {
```

```
    --it;
```

```
    }
```

```
pair<const string, string>& operator*() {
```

```
    return *it;
```

```
}
```

```
bool operator != (const iterator& i) {
```

```
    return i.it != it;
```

```
    }
```

```
bool operator == (const iterator& i) {
```

```
    return i.it == it;
```

```
    }
```

```
friend class Guia_Tlf;
```

```
};
```

```
iterator begin() {  
    iterator i;  
    i.it = datos.begin();  
    return i;  
}
```

```
iterator end() {  
    iterator i;  
    i.it = datos.end();  
    return i;  
}
```

Se quiere construir el tipo de dato vectorDisperso de string, que se caracteriza porque la mayoría de los elementos toman el mismo valor (que llamaremos valor por defecto). Para representar dicho valor es más eficiente almacenar solo los elementos distintos por lo que se propone la siguiente representación

```
class VectorDisperso {
```

```
private:
```

```
map<int, string> M; //map que almacena los  
string v_def; //valor por defecto
```

```
};
```

donde, para un vectorDisperso v , el elemento de la posición i -ésima del vector, $v[i]$ sería:

$$v[i] = \begin{cases} M[i] & \text{si } M.\text{find}(i) \neq M.\text{end}() \\ v_def & \text{en otro caso} \end{cases}$$

Implementar un método que cambie el valor por defecto del vector (teniendo en cuenta los elementos del vector disperso cuyo valor anterior fuese nr)

void VectorDisperso::cambiar_valor_defecto (const string & nr)

```
void vectorDisperso::cambiar_valor_defecto (const
{
    map<int, string>::iterator it;
    for (it = M.begin(); it != M.end(); )
    {
        if (M[it] == nv) // (it).second == nv
            M.erase(it);
        else it++;
    }
    v_def = nv;
}
```

Para gestionar un documento, se usa un TDA Documento. Este TDA tiene en su representación una tabla Hash en la que cada palabra del documento tiene asociada una lista ordenada con las posiciones en las que aparece la palabra en el mismo.

Implementar una función

`int Documento::min_distancia (string palabra1,
string palabra2)`

que devuelva la distancia mínima en la que aparecen las palabras `palabra1` y `palabra2` en el documento. Para la representación de la tabla Hash se usa hashing abierto

Usamos el TDS map

class Documento {

std::map<string, list<int>> tabla_hash;

}; ↑
unordered_map //Tabla hash STL

int Documento::min_distancia (string p1, string p2)

{

list<int> l1 = tabla_hash[p1];

list<int> l2 = tabla_hash[p2];

int minima = numeric_limits<int>::max();

for (list<int>::iterator it1 = l1.begin();
it1 != l1.end(); ++it1)

for (list<int>::iterator it2 = l2.begin();

it2 != l2.end(); ++it2)

{ int d = abs (*it1 - *it2);

if (d < minima) minima = d;

}

return minima;

}

Tenemos una clase Liga que almacena los resultados de los enfrentamientos de una liga de baloncesto

struct enfrentamiento {

 unsigned char eq1, eq2; //códigos de los equipos
 unsigned int puntos_eq1, puntos_eq2; //puntos
 //por cada equipo

};

class Liga {

private:

 list<enfrentamiento> res;

 //

};

Implementar una clase iteradora dentro de la clase Liga que permita recorrer los enfrentamientos en los que el resultado haya sido empate. Implementar los métodos begin() y end()

class iterator {

private:

list<enfrentamiento>::iterator it, final;

public:

iterator() { }

bool operator == (const iterator & i) const
{ return i.it == it; }

enfrentamiento & operator* ()

{ return (*it); }

struct enfrentamiento {

unsigned char eg1, eg2;

unsigned int puntos_eg1;

puntos_eg2;

iterator & operator++ ()

{ ++it; }

bool salir = false;

while (it != final & & ! salir)

{ if ((*it).puntos_eg1 == (*it).puntos_eg2)
 salir = true

 else ++it

 return *this

class Liga {

private:

list<enfrentamiento> res;

= . . .

friend class Liga;

iterator begin() { }

iterator i;

i.it = res.begin();

i.final = res.end();

if (!(*i.it).puntos_eg1
 == (*i.it).puntos_eg2)

 ++i

return i

};

iterator end() { }

iterator i;

i.it = res.end();

i.final = res.end();

return i;

};

se dispone de un contenedor de pares de elementos
{ clave, binree<int> }

definido como:

```
class contenedor {  
    private:  
        map<string, binree<int>> datos;  
    --  
}
```

Implementar un iterador que itere sobre los string
de longitud 4 y para los que el binree<int>
tenga una estructura de ABB

class contenedor

private:

map<string, biutree<int>> datos;

====:

5

item sobre string de size 4 / biutree<int> sea ABB.

bool esABB (biutree<int>::node n, int min, int max)

{

 if (n.null())
 return true;

 else
 if ((n->min < min || n->max > max)) return false;

 if (!n->left().null())
 if (n->left() < n->right()) return false;

 if (!n->right().null())
 if (n->right() > n->left()) return false;

 return esABB (n->left(), min, n->min) &&
 esABB (n->right(), n->max, max)

5

iterator begin()

iterator i;

i->it = datos.begin();

i->final = datos.end();

if (!((*(i->it)).first.size()) == 4 &&

 esABB ((*(i->it)).second.root(), min1, max1))

++ i;

return i;

iterator end() {

iterator i;

i->it = datos.end();

i->final = datos.end();

return i;

5

```
class Container {
```

```
private:
```

```
map<string, binTree<int>> datos;
```

```
public:
```

```
class iterator {
```

```
private:
```

```
map<string, binTree<int>>::iterator it, final;
```

```
public:
```

```
iterator();
```

```
bool operator==(const iterator & i) const
```

```
{ return i.it == it; }
```

```
bool operator!=(const iterator & i) const
```

```
{ return i.it != it; }
```

```
pair<const string, binTree<int>> operator*()
```

```
{ return *it; }
```

```
iterator & operator++()
```

```
{ ++it;
```

```
bool salir = false;
```

```
while (it != final && !salir)
```

```
{
```

```
if ((*it).first.size() == 4 &&
```

```
esABB((*it).second.left, (*it).second.right),
```

```
min(), max()))
```

```
salir = true;
```

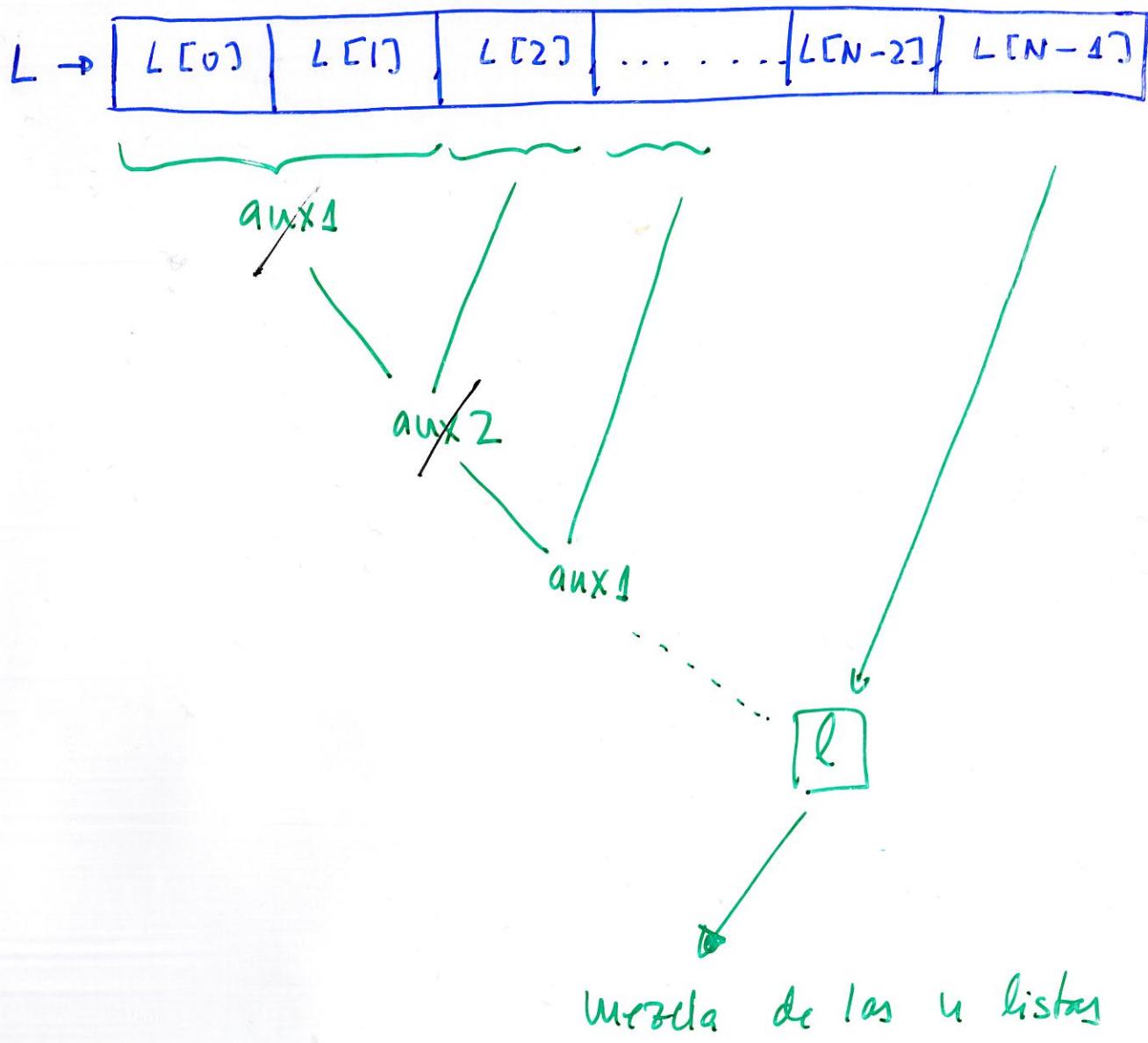
```
else ++it;
```

```
return *this;
```

```
friend class Container;
```

```
3
```

```
numeric_limits<int>::min(), max()
```



```

lista mezcla (tlista * listas, int n)
{
    tlista l; tlista aux1, aux2; int i;
    if (n==2) l = mezcla2 (listas [0], listas [1]);
    else {
        aux1 = mezcla2 (listas [0], listas [1]); /* mezcla de
                                                    los 2 primeros */
        for (i=2; i<n-1; i++) /* mezcla de los restantes
                                menos el último */
            if (i%2) { /* impar */
                aux1 = mezcla2 (listas [i], aux2);
                destruir (aux2);
            }
            else { /* par */
                aux2 = mezcla2 (listas [i], aux1);
                destruir (aux1);
            }
        if ((n-1)%2) /* mezcla del último */
        {
            l = mezcla2 (listas [(n-1)], aux2);
            destruir (aux2);
        }
        else {
            l = mezcla2 (listas [(n-1)], aux1);
            destruir (aux1);
        }
    }
    return l;
}

```

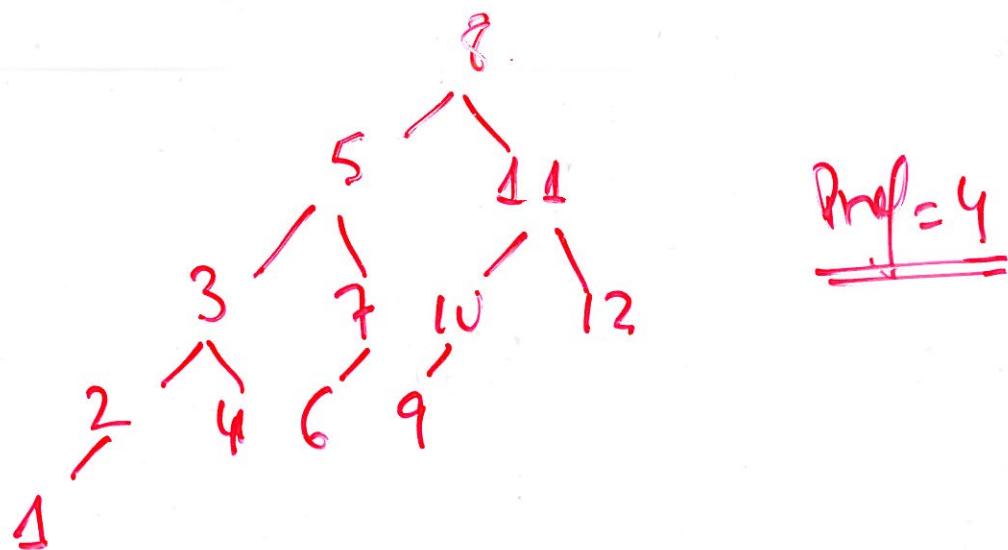
¿Son válidas estas funciones hash?

$$h(1\zeta) = [\zeta/n]$$

$$h(1<) = 1$$

$$h(k) = (k + \text{random}(n)) \% n$$

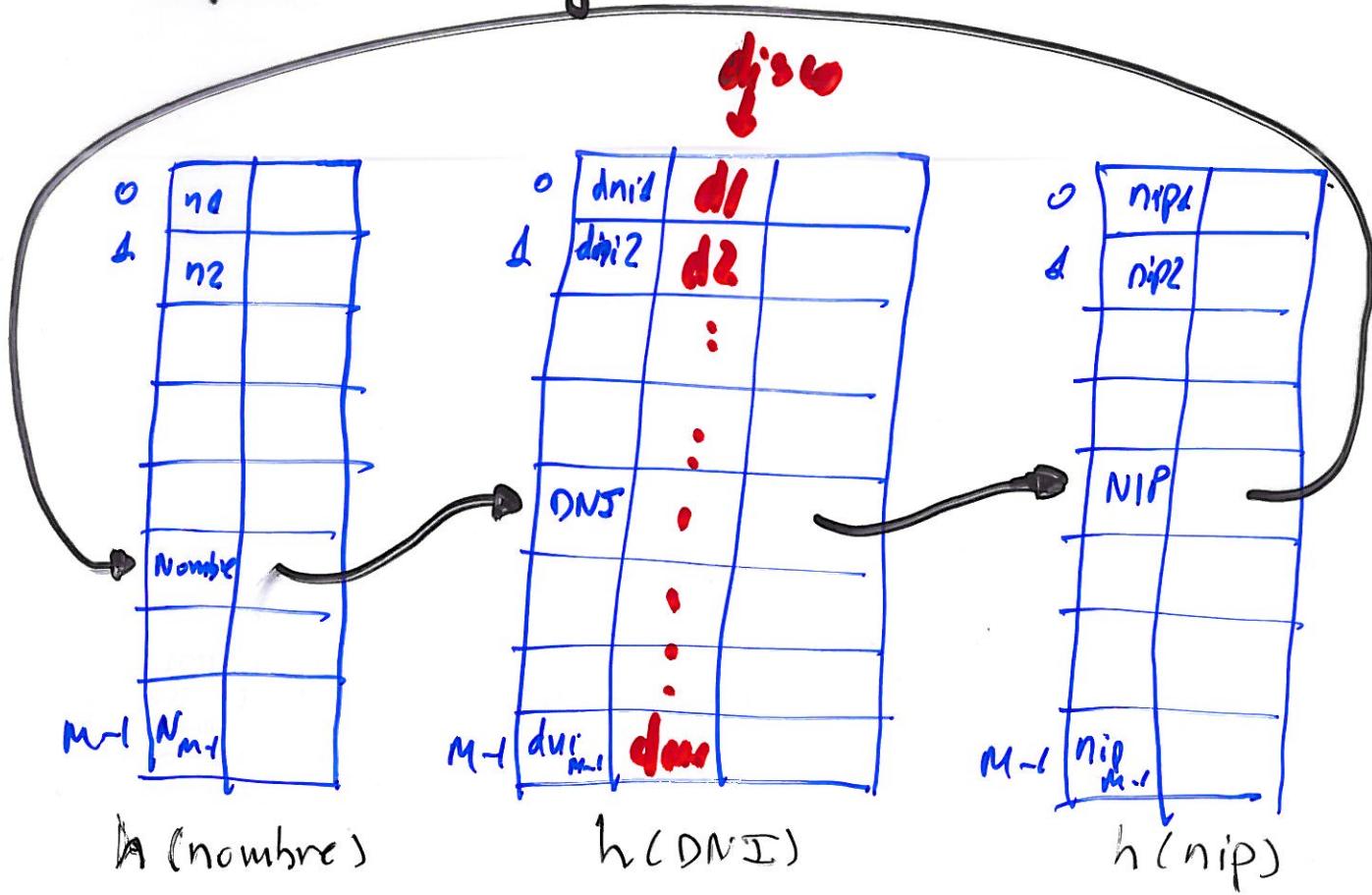
Construir un AVL que contenga como etiquetas los enteros del 1 al 12 y que tenga la mayor profundidad posible.



Junio 2010

5 b)

Los empleados de una empresa se representan en una base de datos por su nombre, DNI y número de identificación personal como identificadores únicos junto con la información adicional del empleado. Construir una estructura de tablas hash que permita acceder en un tiempo $O(1)$ en media al registro de un empleado por cualquiera de estos 3 campos, teniendo en cuenta que no hay espacio para duplicar los registros.



Febrero 2013

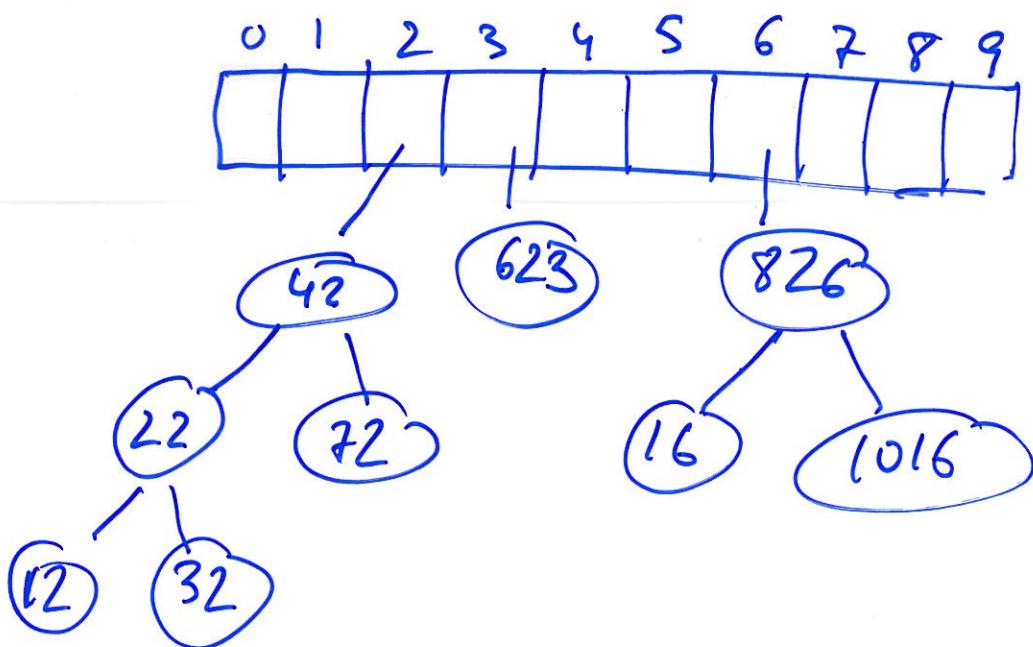
Problema 5

Supongamos que en una Table Hash hacemos uso en cada celda de AVL en lugar de listas,

- * ¿Cuál es la eficiencia (peor caso) de la función buscar?
- * Mostrar la tabla hash resultante de insertar los elementos:

x	16	72	826	1016	12	42	623	22	32
$h(x)$	6	2	6	6	2	2	3	2	2

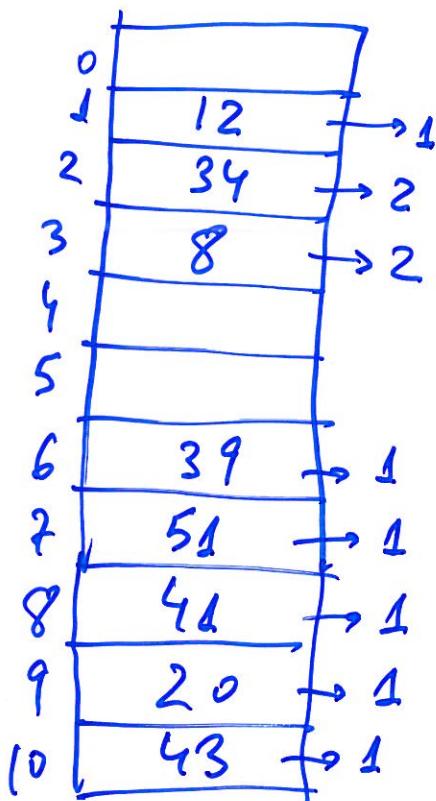
$2h(x) ?$



Insertar las claves $\{12, 41, 8, 34, 51, 20, 43, 39\}$ en una tabla hash cerrada de tamaño 11 usando como función hash $h(k) = k \% 11$ y para resolver colisiones, rehashing doble usando como función hash secundaria:

$$h_2(k) = 7 - (k \% 7) \quad ? \text{ Rendimiento?}$$

	12	41	8	34	51	20	43	39	
$h_1(k)$	4	8	8	1	7	9	10	6	
$h_2(k)$	2	1	6	1	5	4	6	3	



$$\text{Rend} = \frac{10}{8} = 1.25 \text{ int/clave}$$

rellenar la tabla con la eficiencia de las operaciones de los filos en las estructuras de datos de las columnas. ($T \rightarrow \text{int}$)

	Vector	Lista	ABR	DVL	Pila	T. Heaps
Encontrar mínimo						
Encontrar máximo						
# [a, b] a <= b						
Imprimir ordenad.						

	Vector orden	Lista orden	ABR	DVL	Pila	T. Heaps
Encontrar mínimo	$O(1)$	$O(1)$	$O(n)$	$O(n \log_2 n)$	$O(n)$	$O(n)$
Encontrar máximo	$O(1)$	$O(1)$	$O(n)$	$O(n \log_2 n)$	$O(n)$	$O(n)$
# [a, b] a <= b	$O(n \log_2 n)$	$O(n)$	$O(n)$	**	$O(n)$	$O(n)$
Imprimir ordenad.	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

* si un algoritmo especial

** $O(\max(\log_2 k, \# [a, b]))$

*** con heapsort P. ej.

Junio 2010

5.2.2

? Una de las siguientes funciones hash te parecerá más apropiada para insertar elementos en una tabla hash abierta / cerrada? ¿Por qué?

a) $h(k) = k \bmod (M \times N)$ M, N primos

b) $h(k) = k^2 \bmod M$ M primo

c) $h(k) = (M - (k \bmod M)) - 1$ M primo

(a) $M \times N$ no es primo

(b) OK

(c) OK

Aquí todos los posibles son válidos

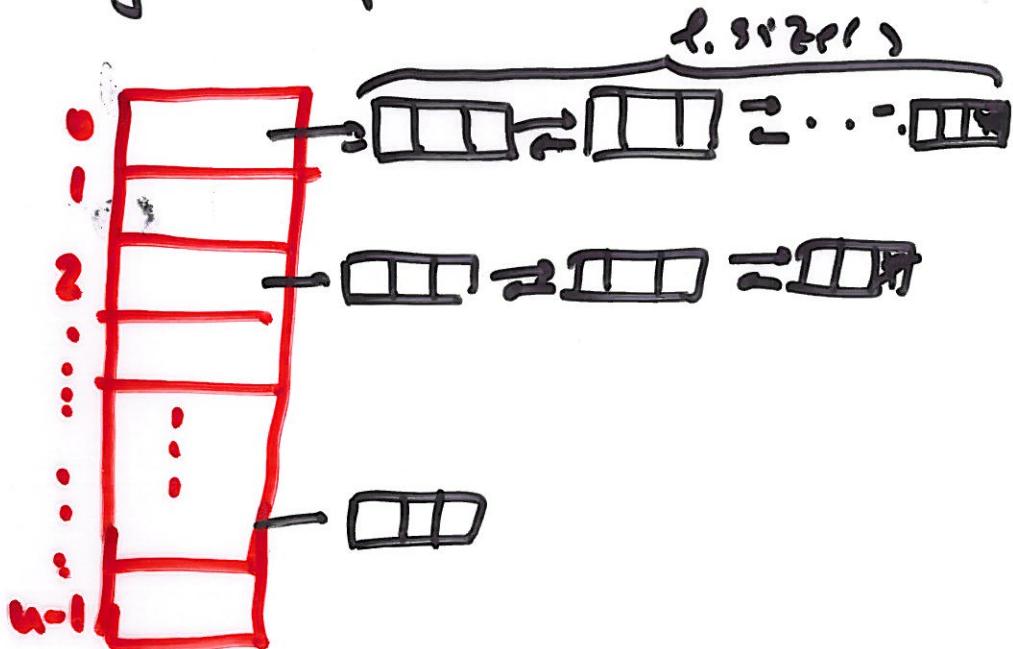
$$k \bmod M = 0 \rightarrow h(k) = M - 1$$

$$k \bmod M = M - 1 \rightarrow h(k) = (M - (M - 1)) - 1 = 0$$

Rango $[0, M-1]$

Sept - 2010

5.6 Se define el índice radial de una tabla hash abierta como el número de cubetas de la tabla multiplicado por el tamaño de la cubeta con mayor número de elementos. Construir un algoritmo para calcular dicho índice radial



Algoritmo

1. Determinar el tamaño del vector (número de cubetas): n ($U.size()$)
2. Activar la función $size()$ para cada una de las listas $U[i]$ y encontrar el $maxsize$
3. Multiplicar $n * maxsize$

$\{3, 10, 29, 66, 127\}$

$$h_1(k) = k \% 7$$

$$h_i(k) = [h_{i-1}(k) + h_0(k)] \% 7 \quad (i=2, 3, \dots)$$

$$h_0(k) = 1 + k \% 5$$

	3	10	29	66	127	1
$h_1(k)$	3	3	1	3	1	
$h_0(k)$	4	1	5	2	3	

$$h_1(3) = 3 \% 7 = 3$$

$$h_0(3) = 1 + 3 \% 5 = 4$$

$$h_1(10) = 10 \% 7 = 3$$

$$h_0(10) = 1 + 10 \% 5 = 1$$

$$h_1(29) = 29 \% 7 = 1$$

$$h_0(29) = 1 + 29 \% 5 = 5$$

$$h_1(66) = 66 \% 7 = 3$$

$$h_0(66) = 1 + 66 \% 5 = 2$$

$$h_1(127) = 127 \% 7 = 1$$

$$h_0(127) = 1 + 127 \% 5 = 3$$

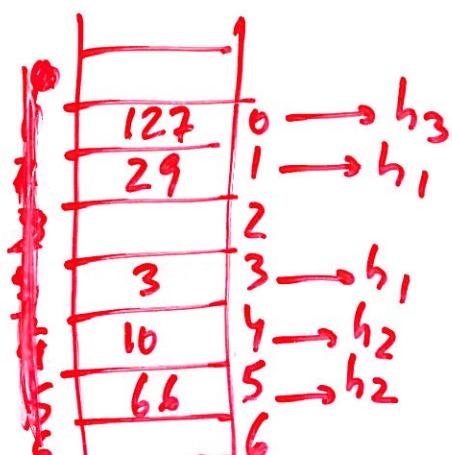
Colisiones

$$h_2(10) = [h_1(10) + h_0(10)] \% 7 = (3 + 1) \% 7 = 4$$

$$h_2(66) = [h_1(66) + h_0(66)] \% 7 = (3 + 2) \% 7 = 5$$

$$h_2(127) = [h_1(127) + h_0(127)] \% 7 = (1 + 3) \% 7 = 4$$

$$h_3(127) = [h_2(127) + h_0(127)] \% 7 = (4 + 3) \% 7 = 0$$



TEST

1. El TDA cola con prioridad no es más que un caso particular del TDA cola
2. El orden en que las hojas se listan en los recorridos pre, in y post en un árbol Binario es el mismo en los 3 casos
3. Un AVL puede reconstruirse de forma única dado su recorrido en inorden
4. Es imposible que un árbol Binario sea AVL y ÁRB a la vez
5. En un esquema de hashing doble nunca puede ocurrir que para 2 claves $k_1 \neq k_2$ ocurra simultáneamente que $\begin{cases} h_1(k_1) = h_1(k_2) \\ h_0(k_1) = h_0(k_2) \end{cases}$
6. Un analista tiene que resolver un problema de tamaño 100 y para ello utiliza un algoritmo $O(n^2)$. Uno de sus colaboradores consigue obtener una solución que es $O(n)$. El analista debe olvidarse de su primera solución y usar la de su colaborador de mejor eficiencia en tiempo

7. Si la eficiencia de un algoritmo viene dada por la función $f(n) = 1 + 2 + \dots + n$, ese algoritmo es $O(\max\{1, 2, \dots, n\})$ es decir $O(n)$

8. Puede hacerse esta definición
stack <int>: iterator P:

9. Un DPD puede reconstruirse de forma unívoca dado su recorrido en postorden

10. En un esquema de hashing doble es correcto usar como función hash secundaria la función:

$$h_0(x) = [(B-1) - (x \% B)] \% B \quad B \text{ primo}$$

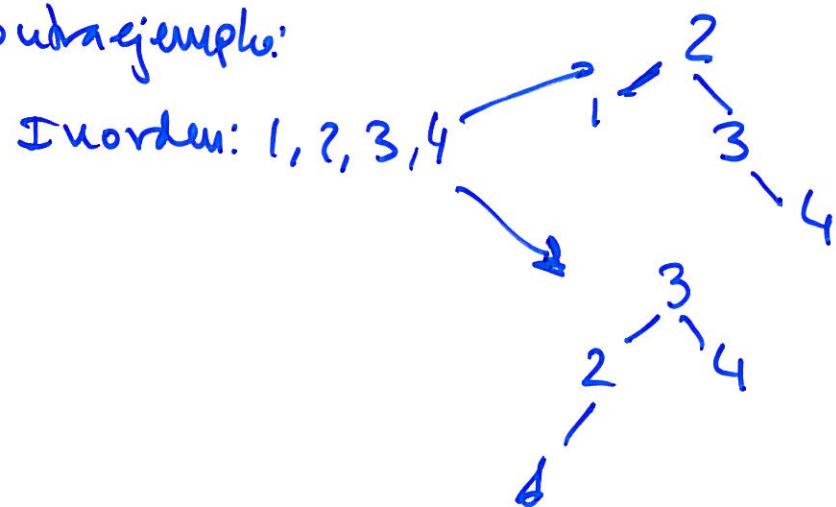
1. Falso

2. Verdadero. El orden en que los hijos se listan en los 3 recomidos es el mismo y en particular el orden de las hojas (siempre de izquierda a derecha)

3. Falso.

contraejemplo:

Inorden: 1, 2, 3, 4



4. Verdadero salvo casos degenerados (todas las claves iguales o árboles de profundidad 1). Las condiciones de definición de los 2 árboles dan lugar a una contradicción:

$$E(\text{padres}) < E(\text{izquierda}) \quad \text{y} \quad \text{APD}$$

$$E(\text{padres}) < E(\text{derecha})$$

de P

imposible

$$E(\text{padres}) > E(\text{izquierda})$$

$$E(\text{padres}) < E(\text{derecha})$$

AVL

5 Falso, si que puede ocurrir:

$$h(k) = k/0.5$$

$$h_0(k) = 3 - k/0.3$$

k	3	18
$h(k)$	3	3
$h_0(k)$	3	3

$$h(k) = k/0.7$$

$$h_0(k) = 2 + k/0.5$$

k	70	35
$h(k)$	0	0
$h_0(k)$	1	1

6 Falso. La eficiencia en espacio y las constantes pueden influir en ejemplos de tamaño pequeño

7. Falso. El algoritmo es $\Theta(n^2)$

8 Falso. Una pila no tiene iteradores

9. Verdadero. Por la condición geométrica de los APO

10. No es una función correcta porque puede llegar a tener el valor 0.

4. Para cada función $f(n)$ y cada tiempo t de la tabla siguiente, determinar el mayor tamaño de un problema que puede ser resuelto en un tiempo t (suponiendo que el algoritmo para resolver el problema tarda $f(n)$ microsegundos, es decir, $f(n) \times 10^{-6}$ sg.)

$f(n)$	t				
	1 sg.	1 h.	1 semana	1 año	1000 años
$\log_2 n$	$\approx 10^{300000}$	$10^{10^{12}}$	$10^{1.82 \cdot 10^{10}}$	$10^{94.9 \cdot 10^{11}}$	$10^{9.5 \cdot 10^{45}}$
n	10^6	$3.6 \cdot 10^8$	$6048 \cdot 10^8$	$\approx 3,15 \times 10^{23}$	$3.15 \cdot 10^{15}$
$n \log_2 n$	62.74	$\approx 1,33 \times 10^8$	$1.77 \cdot 10^{10}$	$7.98 \cdot 10^{11}$	$6.41 \cdot 10^{14}$
n^3	$3.3 \cdot 10^5$	1533	8457	31594 146645	$3.1594 \cdot 10^5$
2^n	19	31.7	39.1	44.8	54.8
$n!$	10	12	15	17	19