Soluciones de los Ejercicios, MARP I

Mario Calvarro Marines

Índice general

Hoja 1

Ejercicio 1. (Iker Muñoz)

No será coste constante amortizado. Contraejemplo:

```
fun multiapilar (p: pila, k: nat)
    mientras k > 0 hacer
        apilar (p)
        k = k - 1
    fmientras
ffun
```

Coste $O(k)$. Si consideramos la secuencia de n llamadas a multiapilar tendremos:

$$\hat{c_i} = \frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{i=1}^{n} k = k.$$

Ejercicio 2. (Sergio García)

Sí:

$$\underbrace{1\ldots\underbrace{10\ldots0}_{k-1}}_{n} \to^{\text{incrementar}} \underbrace{1\ldots\underbrace{01\ldots1}_{k-1}}_{n} \to^{\text{decrementar}} \underbrace{1\ldots\underbrace{10\ldots0}_{k-1}}_{n} \to \ldots n \text{ operaciones.}$$

Incrementar y decrementar: $O(k) \Rightarrow n$ operaciones $O(nk)$.

Ejercicio 3. (Alba Bautista)

Hemos de calcular $\frac{\sum_{j=1}^{n} c_j}{j}$. Lo hacemos primero para $j = 2^n$.

Consideramos $C_{in} = \{2^i : 0 \leq i \leq n\}$ y $\overline{C}_{in} = (1, \ldots, 2^n)\ \ C_n$.

$$\sum_{i=1}^{\infty} C_i = \sum_{i \in C_n} C_i + \sum_{i=1}^{n} C_i = \sum_{i=1}^{\infty} 2^i + \sum_{i \in C_n} 1 = 2^{n+1} - 1 + \mid \overline{C}_n \mid\mid \overline{C}_n \mid = \mid 1 \ldots 2^n \mid - \mid C_n \mid =$$

$$= 2^n - (n+1) \Rightarrow \sum_{i=1}^{j} C_i = \left(2^{n+1} - 1\right) + (2^n - n - 1) = 3 \cdot 2^n - n - 2 \Rightarrow \frac{\sum_{i=1}^{j} C_i}{2^n} \leq 3 \in O(1)$$

De forma similar si tomamos $j = 2^n + j' : j' \in 1, \ldots, 2^n - 1$.

$$\Rightarrow \sum_{i=0}^{j} 3 \cdot 2^n - n - 2 + j \Rightarrow \frac{\sum_{i=0}^{j} C_i}{j} \leq 3 \in O(1)$$

Ya que $\begin{cases} 2^n \leq j \\ j' \leq j \end{cases}$

Ejercicio 4. (Juan Fonseca)

Utilizamos el método del potencial para calcular el coste amortizado:

$\Phi(D_i)$ elementos en la lista tras operación i-ésima

- Añadir un número:

$$\hat{c}_i = c_1 + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (k+1) - k = 2 \in O(1)$$

- Reducir-lista:

$$\hat{c}_i = c_1 + \Phi(D_i) - \Phi(D_{i-1}) = k + 1 + 1 - k = 2 \in O(1)$$

Ejercicio 5. (Marta Vicente)

```
proc contar(C[0, ..., k - 1] de {0, 1}, E/S posSig: nat)
    j := 0
    mientras j < k AND j < posSig AND C[j] = 1
        C[j] := 0
        j := j + 1
    fmientras

    si j < k AND j < posSig
        C[j] := 1
    fsi

    si j < k AND j = posSig
        C[j] := 1
        posSig := posSig + 1
    fsi
fproc
```

```
proc resetear (E/S posSig: nat)
    posSig := 0
fproc
```

$$valor(c) = \sum_{j=0}^{\text{posSig}} 2^j C[j]$$

Ejercicio 6. (Juan Diego Barrado)

DISCLAMER: Este ejercicio no está ni de cerca completo. Recomiendo no utilizar este documento para estudiarlo.

Buscar:

Para cada A_i, hacer búsqueda binaria tiene coste en $O\left(\log 2^i\right) = O\left(i\right)$.

En el caso peor, el elemento que buscamos está en A_{k-1}, luego $\sum_{i=0}^{k-1} i = \frac{(k-1)k}{2} \in O\left(k^2\right) = O\left(\log^2\left(n\right)\right)$.

Insertar: Cuando insertamos n elementos como mucho viajan al nivel $\log\left(n\right)$ vaciamos A_0 de 1 elemento $\frac{n}{2}$ veces, A_1, 2 elementos $\frac{n}{4}$ veces y A_j, 2^j elementos $\frac{n}{2^{j+1}}$ veces $\Rightarrow$

$$T\left(n\right) = \sum_{j=0}^{\log n} 2^j \cdot \frac{n}{2^{j+1}} = \frac{n}{2}\log n.$$

Ejercicio 7. (Leonardo Macías)

Base: $b\left[0,\ldots,n-1\right] \in \mathbb{Z}^n$ con $b\left[i\right] \geq 2$.

Contador: $v\left[0,\ldots,n-1\right]$ con $0 \leq v\left[i\right] < b\left[i\right]$ y $v\left[0\right]$ menos significativo.

```
proc incrementar (b: base, v: contador)
    i := 0
    mientras i < n AND v[i] = b[i] - 1 hacer
        v[i] := 0
        i := i + 1
    fmientras
    si i < n hacer
        v[i] := v[i] + 1
    fsi
fproc
```

$v\left[0\right]$ cambia n veces, $v\left[1\right]$ cambia $\frac{n}{b[0]}$ veces, $\ldots$, $v\left[i\right]$ cambia $\frac{n}{\prod_{j=0}^{i} b[j]}$ veces.

M. Agregación: Nº Cambios: $\sum_{i=0}^{n} \frac{n}{\prod_{j=0}^{i} b[j]}$

Como $\forall j \in \{0,\ldots,n-1\} : b\left[j\right] \geq 2$:

$$\sum_{i=0}^{n} \frac{1}{\prod_{j=0}^{i} b\left[j\right]} \leq \sum_{i=0}^{n} \frac{1}{2^i} \leq \sum_{i=0}^{\infty} \frac{1}{2^i} = 2$$

Coste amortizado:

$$\frac{1}{n}\sum_{i=0}^{n} \frac{n}{\prod_{j=0}^{i} b\left[j\right]} = \sum_{i=0}^{n} \frac{1}{\prod_{j=0}^{i} b\left[j\right]} \leq 2$$

Ejercicio 8. (Beñat Pérez)

a) Utilizamos dos pilas una de los elementos y otra de máximos.

Operaciones:

```
proc apilar(p1, p2: pila; k: ent)
    si !p2.empty AND k <= p2.top =>
        p2.push(k)
    fsi
    p1.push(k)
fproc
```

```
proc maximo (p2: pila; k: ent)
    si !p2.empty =>
        return p2.top
    fsi
fproc
```

```
proc desapilar(p1, p2: pila)
    si !p1.empty =>
        si p1.top = p2.top =>
            p2.pop
        fsi
    fsi
    p1.pop
fproc
```

b) Operaciones:

```
proc anyadir(p1: pila; k: ent)
    si !p1.empty =>
        p1.push(k, max(k, p1.top)
    si no =>
        p1.push(k, max)
    fsi
fproc
```

```
func max(p1, p2: pila)
    return max(p1.top.b, p2.top.b)
fproc
```

```
proc desapilar(p1, p2: pilas; k: ent)
    si !p2.empty =>
        si !p1.empty =>
            p2.push(p1.top.a, p1.top.a)
            p1.pop
            mientras !p1.empty hacer
                p2.push(p1.top.a, max(p1.top.a, p2.top.b)
                p1.pop
            fmientras
            p2.pop
        fsi
    si no =>
        p2.pop
    fsi
fproc
```

En el caso peor tendremos $O(n)$. Sin embargo, amortizado de n operaciones nos saldrá $O(1)$ por operación.

Por el método de contabilidad. Asumamos que apilar tiene coste 3 (poner el elemento en p_1, quitarlo de p_1 y ponerlo en p_2). Así, solamente nos queda que desapilar tiene coste 1, ya que habría que hacer una sola operación.

Además, la función máximo es constante.

Con todo, nos queda que el coste amortizado es $O(1)$.

Ejercicio 9 (Lucía Alonso)

Tenemos una cola C, creamos una doble cola llamada *maximos* para llevar el control del máximo de C_1:

$$maximos[i] : 0 \leq i < n : \forall j : i < j \leq n : maximos[i] \leq maximos[j]$$

- En la llamada a *borrar* () elemento de C_1 debemos comprobar si es 1^{er} máximo.
- En la llamada a *insertar* () elemento de C_1 debemos borrar todos los elementos de máximo menores que él.

Ejemplo:

$$\begin{aligned} C = 5,4,3,2,1; \quad & maximos = 5,4,3,2,1 \Rightarrow insertar\,(3) \\ C = 5,4,3,2,1,3; \quad & maximos = 5,4,3,3 \Rightarrow remove\,() \\ C = 4,3,2,1,3; \quad & maximos = 4,3,3 \Rightarrow insert\,(6) \\ C = 4,3,2,1,3,6; \quad & maximos = 6. \end{aligned}$$

```
proc insertar(cola: C1, doble cola : maximos, elemento: x)
    c.push(x)
    mientras !maximos.empty() AND maximos.back() < x
        maximo.pop_back()
    fmientras
    maximos.push_back(x)
fproc
```

```
proc borrar(cola: C1, doble cola: maximos)
    si c.front() = maximos.front()
        maximos.pop_front()
    fsi
    c.pop()
fproc
```

```
{!maximo.empty()}
func getMax(doble cola: maximo)
    return maximo.front()
ffunc
```

Coste amortizado. Por el método del potencial.

- Caso peor: Se añade el máximo de C y *maximos* está ordenada decrecientemente:

  $$c_i = 4 + n \in O\,(n) \Rightarrow \text{Lineal.}$$

  En la siguiente llamada, cola vacía:

  $$c_{i+1} = 4 + 1 \in O\,(1) \Rightarrow \text{cte.}$$

- Método del potencial:

  $$\Phi\,(D_i) = \text{longitud de la doble cola de máximos.}$$

  Al empezar con C vacía se cumple $\forall i : \Phi\,(D_i) - \Phi\,(D_0) > 0$.

  - En la llamada a $getMax\,()$:

    $$\hat{c}_i = c_i + \Phi\,(D_i) - \Phi\,(D_{i-1}) = 1 \in O\,(1)$$

    Al ser una operación de consulta.
  - En la llamada $borrar\,()$:

    $$\hat{c}_i = c_i + \Phi\,(D_i) - \Phi\,(D_{i-1}) = \begin{cases} 2 \text{ si no se borra el máximo} \\ 2 + m - 2 - m = 1 \text{ c.c} \end{cases}$$

  - En la llamada a $insertar\,()$:

    $$\hat{c}_i = c_i + \Phi\,(D_i) - \Phi\,(D_{i-1}) = 4 + b_i + (k - b_i) - k = 4 \in O\,(1)$$

  Es decir, el coste amortizado es constante.

Ejercicio 10 (Javier Saras)

Por el método del potencial.

$$\Phi(D_i) = M + m$$

Con $\Phi(D_0) = 0$ y $\Phi(D_i) \geq 0$.

- *maximo* ():
$$\hat{c_i} = 1 + M + m - M - m = 1$$
- *minimo* ():
$$\hat{c_i} = 1 + M + m - M - m = 1$$
- *eliminar* ():
  - Máximo: $\hat{c_i} = 2 + M - 1 + m - M - m = 1$.
  - Mínimo: $\hat{c_i} = 2 + M + m - 1 - M - m = 1$.
  - Ninguno: $\hat{c_i} = 1 + M + m - M - m = 1$.
- *insertar* ():
  - Máximo: $\hat{c_i} = m + 3 + 1 + m + 1 - (M + m) = 5$
  - Mínimo: $\hat{c_i} = m + 3 + M + 1 + 1 - (M + m) = 5$
  - Ninguno: $\hat{c_i} = a + b + 3 + M - a + 1 + m - b + 1 - (M + m) = 5$

Ejercicio 11 (Javier Amado)

```
class multiconj {
    valores: array <int>
    n: int
    +constructor vacio() {
        valores := new int[max]
        n := 0;
    }

    +insertar(v: int) {
        si (n < max) {
            valores[n] = v
            n := n + 1
        fsi
    }
    +elimMayores {
        nborrar := ceil(n/2)
        m := mediana(valores/n)
        i := 0; elim = 0; j = 0;
        mientras i < n
            si valores[i] > m
                valores [i] := null
                elim := elim + 1
            fsi
            i := i + 1
        fmientras
        mientras j < n AND elim < nborrar
            si valores[j] = m
                valores[j] := null
                elim := elim + 1
            fsi
            j := j + 1
        fmientras
        i := 0; j := 0
        mientras j < n
            si valores[j] != null
                valores[i] = valores[j]
```

```
                i := i + 1
            fsi
        fmientras
        n := n - elim; f elim mayores?
    }
}
```

Coste por el método del potencial

$$\Phi(D_i) = 2n$$

$$\hat{c}_{insertar} = c_{insertar} + (\Phi(D_i) - \Phi(D_{i-1})) = 1 + 2 = 3 \in O(1)$$

$$\hat{c}_{elimMayores} = c_{elimMayores} + (\Phi(D_i) - \Phi(D_{i-1})) = n - 2\lfloor n/2 \rfloor = \begin{cases} 0 \text{ si } n \text{ es par} \\ 1 \text{ si } n \text{ es impar} \end{cases}$$

Hoja 2

Ejercicio 1. (Natalia Rodríguez)

```
func esEA (t: arbolbinario) dev (h: ent, b: bool)
   caso vacio?(t) entonces (0, true)
       !vacio?(t) entonces sea (l, x, r) = desc(t)
                     (hl, bl) = esEA(l)
                     (hr, br) = esEA(r)
                     en(max(hl, hr) + 1, bl AND br AND |hl - hr| <= 1)
   fcaso
ffunc
```

Ejercicio 2. (Dylan Hewitt)

```
func zurdo(T: arbolBinario) dev (esZurdo: bool, nNodos: nat)
  caso vacio?(t) entonces (true, 0)
    !vacio?(t) entonces sea (iz, x, dr) = desc(t) en
    caso vacio?(iz) AND vacio?(dr) entonces (true, 1)
      !vacio?(dr) OR !vacio?(dr) sean:
              (esZurdoIz, nNodosIz) = zurdo(iz)
              (esZurdoDr, nNodosDr) = zurdo(dr)
         entonces:
         (esZurdoDr AND esZurdoIz AND nNodosDr < nNodosIz,
         nNodosIz + nNodosDr + 1)
    fcaso
  fcaso
ffunc
```

Ejercicio 3. (Laura Rodrigo)

Dibujo

Ejercicio 4. (Pablo García)

(Dibujo)

Sea T_1, T_2 completos con $h_{T_1} = h_{T_2} + 1$

T_1, T_2 son equilibrados por ser completos, T también por serlo T_1, T_2 y $\|h_{T_1} - h_{T_2} = \|h_{T_2} + 1 - h_{T_2}\| = 1 \leq 1$. Sean n_{T_i} el número de nodos de T_i :

$$n_{T_1} = \sum_{i=0}^{h_{T_1}-1} 2^i = \frac{1-2^{h_{T_1}}}{1-2} = 2^{h_{T_1}} - 1 = 2^{h_{T_2}+1} - 1$$
$$n_{T_2} = \sum_{i=0}^{h_{T_2}-1} 2^i = \frac{1-2^{h_{T_2}}}{1-2} = 2^{h_{T_2}} - 1.$$

Entonces:

$$\begin{aligned} n_{T_1} - n_{T_2} &= 2^{h_{T_2}+1} - 1 - 2^{h_{T_2}} + 1 \\ &= 2^{h_{T_2}} \cdot 2 - 2^{h_{T_2}} \\ &= 2^{h_{T_2}} \Rightarrow \\ \lim_{h_{T_2} \to \infty} n_{T_1} - n_{T_2} &= \lim_{h_{T_2} \to \infty} 2^{h_{T_2}} = \infty. \end{aligned}$$

Es decir,

$$\forall C > 0,\ \exists m \in \mathbb{N} : h_{T_2} \leq m \Rightarrow n_{T_1} - n_{T_2} > C.$$

Ejercicio 5. (Iker Muñoz)

Supongamos un desequilibrio LL ($h_{iz} = h_{der} + 2$)

(*Dibujo*)

Por reducción al absurdo.

- Si $h_{ii} = h_{id} = h \Rightarrow$ No hay desequilibrio.
- Si $h_{ii} = h_{id} = h + 1 \Rightarrow$ Antes de insertar ya había un desequilibrio, es decir, no se cumplía la precondición.
- $h_{id} \neq h_{ii}$ (Análogo RR)

Insertamos y hacemos una rotación:

(*Dibujo*)

La rotación disminuye en 1 la altura del árbol y tenemos la altura original.

El elemento ya está insertado y el árbol es AVL, por lo que no es necesario más rotaciones.

Ejercicio 6. (Virginia Chacón)

(*Dibujo*)

Ejercicio 7. (Pablo García)

Muy extenso.

Ejercicio 8. (??)

para transformar un AVL en conjunto, lo recorremos en inorden que nos va a dar una lista ordenada.

```
vector conjunto_arbol(arbol a)
    vector v;
    inorden(v, a);
    return v;
```

```
vector inorden(vector v, arbol a)
    if (a != null) then
        inorden(v, a->izq);
        v.push_back(a->root());
        inorden(v, a->der);
    end if;
```

```
vector union_conjunto(const vector& a, const vector& a1)
    int i = 0, j = 0;
    vector union;
    while (i < a
```

Ejercicio 9. (Antonio Parrilla)

Ejercicio 22. (Alejandro Ysasi)

Blanrrojinegro = rojinegro + color azul (solo hijo de rojo).

Ahora tenemos 3 colores con las siguientes restricciones:

- No puede haber 2 azules seguidos (nodo azul solo tiene hijos azules).
- Un nodo rojo solo puede ser hijo de negro.
- No puede haber 2 rojos seguidos (nodo rojo solo tiene hijo negro o azul).
- Azul solo es hijo de rojo.

Por tanto, la máxima cantidad de colores en un nodo interno son 1 negro, 2 rojos, 4 azules.

Jerárquicamente: negro $>$ rojo $>$ azul.

$2-3-\ldots-8$ árbol permite nodos de hasta 7 claves y 8 hijos.

- Si se inserta un 2-nodo $\Rightarrow$ 3 nodo
- ...
- Si se inserta un 8-nodo $\Rightarrow$ 2 nodo +5 nodo +4 nodo.
- Blanrrojinegro $\Rightarrow$ Ahora se inserta con azul y no con rojo (correspondientes operaciones)
- $2, 3, 4$ nodos igual que rojinegro

Hoja 3

Ejercicio 5. (María del Mar Ramiro)

Montículo de mínimos:

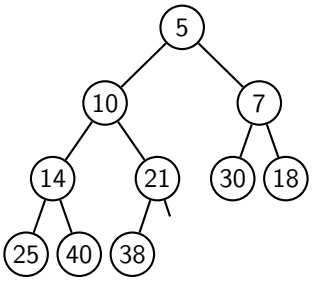
```
fun decrecer(v : monticulo, p : posicion, c : clave)
  si p >= 0 && p < v.size() && c < v[p]
    v[p] := c
    flotar(v, p)
  fsi
ffun
```

```
fun aumentar(v : monticulo, p : posicion, c : clave)
  si p >= 0 && p < v.size() && v[p] < c
    v[p] := c
    hundir(v, p)
  fsi
ffun
```

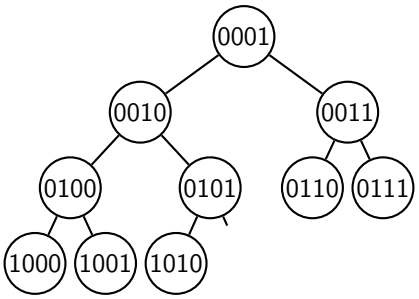
En un montículo de máximos se cambian flotar y hundir.

Ejercicio 7. (Mario Calvarro)

En primer lugar, estudiemos la relación entre la representación binaria de la posición de un nodo y el camino de decisiones que nos lleva a dicho nodo (decisión entre tomar el camino izquierdo o el derecho en los nodos que le preceden). Veámoslo con un ejemplo:

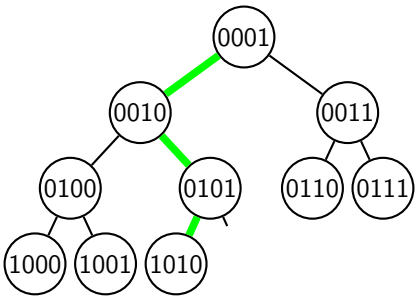


En el árbol presentado tenemos diez nodos diferentes por lo que la representación de sus posiciones en binario irán desde 0001_2 hasta 1010_2, es decir, sustituyendo los valores de los nodos por su posición en binario:



Lo primero que salta a la vista es que la posición del 1_2 más significativo representa el "nivel" en el que se encuentra el nodo empezando por el nivel 1. Por tanto, si el 1_2 más significativo se encuentra en el bit menos significativo se tendrá que es el nivel 1, si está en la siguiente más significativo se tendrá que es el nivel 2, etc.

Por otro lado, los siguientes bits nos indican las decisiones tomadas en los anteriores niveles, siendo un 0_2 camino izquierdo y 1_2 camino derecho. Por ejemplo, si tenemos el nodo $10 = 1010_2$ nos indica que en el camino seguido a la inversa fue izquierda, derecha e izquierda:



A su vez, podemos ver que que realizando un desplazamiento hacia la derecha hallamos directamente la posición del padre de un nodo, siempre que los bits que quedan libres a la izquierda los sustituyamos por un 0. Por ejemplo, si tenemos la posición $10 = 1010_2$ y realizamos un desplazamiento nos quedará $0101_2 = 5$ que es el nodo padre.

Esta operación es equivalente a la división entera por 2 que se realiza en un montículo de Williams implementado por vectores.

Con esto podemos demostrar por inducción que la posición $0 < p < m$, con $m =$ bit 1_2 más significativo, en la cadena de bits nos indica la decisión tomada al bajar del nivel $m - p$:

- Caso base: Si tenemos un árbol de 3 elementos la raíz será 01_2, su hijo izquierdo 10_2 y el derecho 11_2. Es decir, el bit menos significativo nos indicará si el camino seguido era el izquierdo (0) o derecho (1).
- Caso inductivo: Supongamos ahora que todos los nodos hasta el nivel p cumplen esta propiedad y tomemos un elemento cualquiera del nivel $p+1$. Si ahora realizamos un desplazamiento a la derecha nos quedará la disposición binaria del nodo del padre por lo que, aplicando la hipótesis de inducción, nos quedará que todos los bits hasta el $p+1$, que es el 1_2 que indica el nivel, menos el primero, que es el que se pierde con el desplazamiento, nos indican el camino seguido hasta el padre. Pero con el caso base sabemos que el primer bit nos indica si el hijo es el izquierdo o el derecho.

Finalmente, para realizar la implementación de la inserción lo que haremos será seguir el camino que nos indica donde se insertará el nuevo elemento manteniendo la estructura de semicompletitud, es decir, el que nos viene dado por $dec2bin(n+1)$[1]. En el caso de encontrarnos por el camino un

[1]Suponemos que el tamaño del vector que devuelve será hasta el 1_2 más significativo.

nodo con un valor mayor al que se va insertar los cambiaremos y continuaremos con el valor del nodo cambiado. Con esto mantendremos la propiedad de mínimos. Por tanto,

```
void insertar(const T& x)
{
  //Caso Base
  if (n == 0)
    t.root() = new Node<T>(x);
  else
  {
    vector<bool> camino = dec2bin(n+1);
    T elemento = x;
    BinTree<T> arbol = t;

    //Como n > 0 => camino.size() >= 2
    for (size_t i = camino.size() - 2; i > 0; --i)
    {
      if (t.root() > x)
        swap(arbol.root(), x);

      if (camino[i])
        arbol = arbol->left();
      else
        arbol = arbol->right();
    }

    if (t.root() > x)
      swap(arbol.root(), x);

    if (camino[0])
      t.left() = new Node<T>(elem);
    else
      t.right() = new Node<T>(elem);
  }

  ++n;
}
```

El coste del algoritmo vendrá dado por el bucle principal que realiza un total de $\log(n)$ operaciones (el tamaño del vector de bits será $\log(n)$) todas ellas constantes. En definitiva, el coste será $O(\log(n))$ en el caso peor.

Ejercicio 8.(Adrián Carlos Skaczylo)

Insertar:

```
flotar(int i)
  int pos = i;
  Elem p, h, e;
  bool inv = false;
  while (pos != 1 && !inv)
  {
    p = array[pos/2];
    e = array[pos];

    if (pos % 2 = 0 && pos + 1 < array.size())
      h = array[pos + 1]
    else
      h = array[pos - 1]

    if (e.max <= p.min)
      array[pos] = p
      array[pos/2] = e
    else if (e.min >= p.max)
      Inv = true
      if (e.max > h.min && e.min < h.max)
        intercambiar(e, h)
    else
      intercambiar(e, p)
      intercambiar(e, h)

    pos = pos / 2;
  }
```

Eliminar:

```
hundir(int i)
  int pos = i
  Elem e
  int p_HI, p_HD, p_hMP
  bool inv = false;
  p_HI = 2 * pos //puede no existir

  while (!inv && p_HI <= size())
  {
    e = array[pos]
    p_HD = p_HI + 1
    if (p_HD <= size && masPeq(p_HD))
      p_hMP = p_HD
    else
      p_hMP = p_HI
      p_HD = p_HI

    if (masPeq(e, p_hMP))
      inv = true
    else if(masPeq(p_hMP, e)
      array[pos] = array[p_hMP]
      array[p_hMP] = e
      intercambiar(p_HI, p_HD)
    else
      intercambiar(p_hMP, e)
      intercambiar(p_HI, p_HD)

    pos = pos_hMD
    pos_HY = pos * 2
  }
```

Ejercicio 9. (Pablo Zamora)

Ejercicio 11. (Salvador Dzimah)

La secuencia será:

$$0 \quad -1 \quad 1 \quad -2 \quad 2 \quad -3 \quad 3 \quad -4 \quad 4$$

Ejercicio 15

Árbol semicompleto en el que los niveles de máximos y mínimos se van alternando.

Otra opción es dos árboles de Williams, uno de mínimos y otro de máximos que mantengan una referencia al otro para mantener la consistencia.

Otras opciones:

- Deaps.
- Máx-mín
- Intervalos

Ejercicio 23

Hoja 4

Ejercicio 1

No tiene sentido porque es no dirigido

Ejercicio 2

Matriz V^2 y lista de adyacencia $V \cdot E$.

Ejercicio 3

Si es por matriz es el producto normal (en binario). El coste será V^3.

Con listas de adyacencia será el coste respecto a V, E.

Ejercicio 4

Pensar con el grado de entrada/salida 0 o no 0 y aquellos con ambos 0.

Ejercicio 5

Usamos tabla hash de listas. Con la tabla hash solo tenemos la información positiva (actuando como un a matriz). Como los grafos son muy grandes en la práctica, las tablas hash pueden tener muchos conflictos.

Ejercicio 6

Usamos una matriz de adyacencia. Si hay sumidero solo hay uno. Recorremos la matriz buscándolo y, después, lo comprobamos. Fila todo unos y columna ceros. Al revés sabemos directamente que no es sumidero.

Ejercicio 8

Dibujos

Ejercicio 9

Dibujos Empezamos por el 1.

Ejercicio 10

Usamos una pila o una cola dependiendo de si es DFS o BFS.

Ejercicio 11

Clase.

Ejercicio 12

Ejercicio 9.9 libro.

Ejercicio 13

Mismo que antes pero por componentes Descartamos las aristas que generan un ciclo.

Ejercicio 14

9.10 del libro Grado de salida > 0 y el de entrada $= 0$. Solo puede haber uno. Primero buscamos los candidatos.

Ejercicio 15

Cada cambio de adyacencia se cambia el color. Vamos comprobando que se mantienen los colores correctos. Si hay contradicción no es 2-coloreable.

Ejercicio 16

Dibujo

Ejercicio 17

Búsqueda por el camino buscando el mínimo de las aristas. Descartamos aquellos que tienen "mala" capacidad.

Ejercicio 18

Caminos mínimos descartan ciclos. Formamos un subgrafo con las aristas que no forman ciclos. Con esto tenemos un recubrimiento. Con este algoritmo, el camino puede ser mínimo, pero no el árbol.

Ejercicio 19

Ejercicio 20

Variación de Dijkstra. No solo calculamos la distancia mínima, también el nº de caminos. Tendremos un vector de distancias y otro de nº de caminos mínimos. Cuando encontramos un camino mejor también tenemos que actualizar el resto de información.

Ejercicio 21

Otra variación, es caso de igualdad de coste nos quedamos con el camino con menos aristas. Necesitamos llevar información adicional. Debemos mantener correctamente los datos.

Ejercicio 22

Pensamos en el flujo de los ejercicios anteriores. No interesa la distancia mínima, sino la capacidad mejor.

Nos interesa saber el cuello de botella, si este vale, el resto también.

Ejercicio 23

Ejercicios 9.11, 9.12 libro.

Ejercicio 24

Dado árbol de recubrimiento, si empezamos podando las hojas (en ese orden), podemos conseguir lo que se pide.

Ejercicio 25

Concepto de grafo se extiende con la información adicional de si se puede conectar.

Ejercicio 26

Usamos como en Kruskal y Prim con colas de prioridad de mínimos. El test que se hace es comprobar cosas (union-find)

Ejercicio 27

Si es conexo, $\exists$ recubrimiento. Por finitud entre ellos, habrá uno mínimo. Con Kruskal tenemos el bosque de recubrimiento. En el bucle en vez de la condición $V-1$ vértices debemos cambiarlo en caso de que no tengamos todos los vértices (cola vacía o similar). Similar con Prim.

Ejercicio 28

Encontrar un ciclo de longitud mínima cuyo tamaño sea exactamente V.

Ejercicio 31

El árbol de recubrimiento es único. El orden puede ser distinto. Inducción. Debe cumplir el lema de corte.

Hoja 5

Para los ejercicios en general tenemos:

- Estrategia
- Implementación de la estrategia (normalmente ordenación, en pseudocódigo)
- Demostración de la corrección. Solemos usar el método de la reducción de diferencias.
- Coste del algoritmo. Suele ser el de la ordenación.

Ejercicio 1

Clases de equivalencia = TAD conjuntos disjuntos. Primero igualdades, después desigualdades. Con igualdades vamos actualizando el Union-Find buscando las clases de equivalencia. Con desigualdades si dos están en la misma clase, son insatisfactibles. Coste lineal.

Ejercicio 2

Tareas procesador transparencias, generalizado. 12.4 del libro.

Ejercicio 3

12.5 libro.

Ejercicio 4

12.10 libro.

Ejercicio 7

12.8

Ejercicio 8