Soluciones de los Ejercicios, MARP I

Mario Calvarro Marines

Índice general

1. Hoja 1 **5**

1.1. Ejercicio 1. (Iker Muñoz) 5

1.2. Ejercicio 2. (Sergio García) 5

1.3. Ejercicio 3. (Alba Bautista) 5

1.4. Ejercicio 4. (Juan Fonseca) 6

1.5. Ejercicio 5. (Marta Vicente) 6

1.6. Ejercicio 6. (Juan Diego Barrado) 6

1.7. Ejercicio 7. (Leonardo Macías) 7

1.8. Ejercicio 8. (Beñat Pérez) 7

1.9. Ejercicio 9 (Lucía Alonso) 8

1.10. Ejercicio 10 (Javier Saras) 10

1.11. Ejercicio 11 (Javier Amado) 10

2. Hoja 2 **13**

2.1. Ejercicio 1. (Natalia Rodríguez) 13

2.2. Ejercicio 2. (Dylan Hewitt) 13

2.3. Ejercicio 3. (Laura Rodrigo) 13

2.4. Ejercicio 4. (Pablo García) 13

2.5. Ejercicio 5. (Iker Muñoz) 14

2.6. Ejercicio 6. (Virginia Chacón) 14

2.7. Ejercicio 7. (Pablo García) 14

2.8. Ejercicio 8. (??) 15

2.9. Ejercicio 9. (Antonio Parrilla) 15

2.10. Ejercicio 22. (Alejandro Ysasi) 15

3. Hoja 3 **17**

3.1. Ejercicio 5. (María del Mar Ramiro) 17
3.2. Ejercicio 7. (Mario Calvarro) 17

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 $O\left(k\right)$. 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:

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

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

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 $\overline{C}_{in} = (1, \ldots, 2^n)\ \ C_n$.

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

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

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

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

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:

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

- Añadir un número:

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

- Reducir-lista:

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

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
```

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

Ejercicio 6. (Juan Diego Barrado)

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

<u>Buscar</u>:

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 $\sum_{i=0}^{k-1} i = \frac{(k-1)k}{2} \in O\left(k^2\right) = O\left(\log^2\left(n\right)\right)$.

<u>Insertar</u>: 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.

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

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

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

<u>Coste amortizado</u>:

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

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 <u>potencial</u>

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

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

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

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:

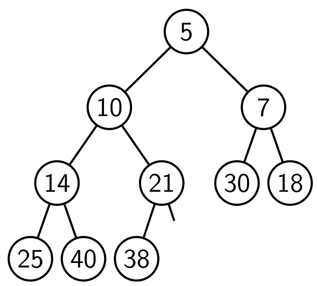
```
fun decrecer(v : monticulo, p : posicion, c : clave)
  si p >= 0 && p < v.size() && c < v[p]
    v[p] := c
    flotar(v, p)
  fsi
ffun
```

```
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
```

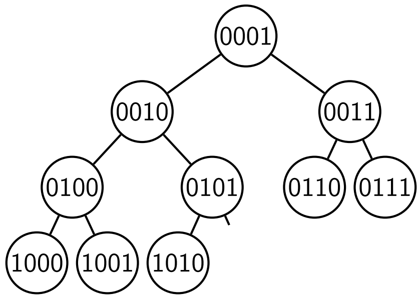
En un montículo de máximos se cambian flotar y hundir.

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:

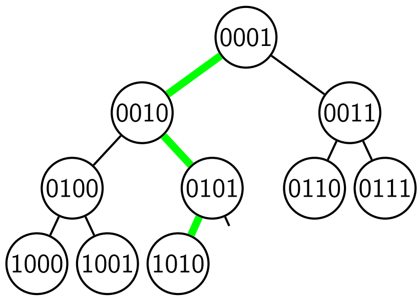


En el árbol presentado tenemos diez nodos diferentes por lo que la representación de sus posiciones en binario irán desde 0001_2 hasta 1010_2, es decir, sustituyendo los valores de los nodos por su posición en binario:



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,

```cpp
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.