Tema 7

Colas de prioridad

7.1. Plantillas de clases y funciones en C++

7.2. El TAD Cola de Prioridad

7.3. El montículo binario (heap)

Plantillas de clases y funciones en C++

Hasta ahora hemos dotado de cierto grado de genericidad a una clase mediante una definición de tipo:

```
typedef int TBL;
```

que obliga a modificar la definición de la clase y recompilarla para adaptar la clase al tipo específico que necesitamos.

▷ C++ proporciona un mecanismo explícito para definir clases y funciones genéricas.

Una **plantilla de clase o de función** es un modelo para crear clases o funciones en el que se parametrizan uno o más tipos o valores.

```
template <class TBABB>
class ArbolBB;

template <class T>
int SeparaIntervalo
(T* vec, int n, T inf, T sup);
```

```
template <class TBL>
class Lista {
  public:
    ...
  private:
    ...
};
```

Plantillas de clases y funciones en C++

Declaración y definición de plantillas de clase

▷ Una declaración o definición de plantilla empieza con `template`.

▷ A continuación se da una **lista de parámetros de plantilla**, separados por comas y delimitados por < y >. No puede estar vacía.

▷ Un parámetro de plantilla puede ser de tipo o de expresión constante.

Parámetro de tipo. Se compone de `class` o `typename` y un identificador.

- ◇ `class` y `typename` son equivalentes en esta lista de parámetros.
- ◇ Cualquier tipo predefinido de C++ (`int`, `float`, `char*`, etc.) o definido por el usuario podrá ser un argumento válido como parámetro de tipo.
- ◇ Un parámetro de tipo es un especificador de tipo en toda la definición de la plantilla. Puede usarse en los mismos contextos que un tipo en una clase.

Parámetro de expresión constante. Es una declaración normal de un parámetro. El nombre del parámetro representa un valor constante.

Plantillas de clases y funciones en C++

Transformación de la clase `Vector` (de enteros) en una plantilla de clase.

```
class Vector {
  public:
    typedef int TBV;
    explicit Vector(int cap_inicial=0)
        { capacidad=cap_inicial; elementos=new TBV[cap_inicial]; }
    ...
};

template <class TBV>
class Vector {
  public:
    explicit Vector(int cap_inicial=0)
        { capacidad=cap_inicial; elementos=new TBV[cap_inicial]; }
    ...
};      // El resto de la definición de la plantilla es igual que la de la clase Vector
```

Plantillas de clases y funciones en C++

Tras la lista de parámetros de plantilla, se da la declaración o definición de la plantilla de clase, exactamente igual que una declaración o definición normal de clase.

Dentro de la definición de la plantilla de clase, su nombre puede utilizarse como especificador de tipo, igual que se hacia hasta ahora con el nombre de una clase.

▷ Cada ocurrencia de `Vector` dentro de su definición es un abreviatura de `Vector<TBV>`.

▷ Esta abreviatura sólo puede utilizarse aquí y dentro de la definición de sus miembros.

Cuando se utiliza fuera de su definición, se debe especificar la lista de parámetros completa: `Vector<TBV>`.

Plantillas de clases y funciones en C++

Instanciación de plantillas de clase

Una definición de plantilla de clase sirve como modelo para la generación automática de instancias de tipo específico de la clase.

Cuando después se escribe:

```
Vector<double> vd;
```

se crea automáticamente una clase `Vector` cuyos componentes son de tipo `double`.

▷ El resultado es equivalente a reescribir el texto de la definición de la plantilla de clase sustituyendo `TBV` por `double` y quitando `template <class TBV>`.

La creación de una clase concreta a partir de una plantilla genérica de clase se llama **instanciación de la plantilla**.

```
Vector<int> vi;        ⇐ vector de enteros
Vector<char*> vpc;     ⇐ vector de punteros a caracteres
Vector<bool> *pvb;     ⇐ puntero a vector de booleanos
```

Plantillas de clases y funciones en C++

Cada instanciación de una plantilla de clase es una clase independiente

▷ Los objetos de la clase Vector<int> no tienen permiso de acceso a los miembros no públicos de Vector<char*>.

La **lista de argumentos de plantilla** se separa por comas y se delimita por < y >, asociándose los tipos y valores específicos dados en la instanciación a los parámetros de plantilla por su posición.

▷ Vector<int> es un nombre de clase, e <int> es su lista de argumentos.

Una instancia de una plantilla de clase puede utilizarse en un programa donde pueda utilizarse cualquier clase, y sus objetos se declaran y usan como los de cualquier clase.

```
void Ordena(Vector<long> & vl) { ... }

Vector<float> *pvf = new Vector<float>[100];
```

Plantillas de clases y funciones en C++

Métodos de las plantillas de clase

Se pueden definir dentro de la definición de la plantilla de clase (inline), o fuera.

Si un método se define fuera, debe hacerse con una sintaxis especial:

```
template <class TBV>
inline
Vector<TBV>::Vector(int cap_inicial) {
  capacidad = cap_inicial;
  elementos = new TBV[cap_inicial];
}
```

Se debe escribir `template` con la misma lista de parámetros que la plantilla de clase, para que el método sea una plantilla de función con los mismos parámetros.

Se debe cualificar el nombre del método (constructor, en este caso) con el nombre de la plantilla de clase acompañada de la lista de parámetros de plantilla.

En el cuerpo del método, los parámetros de plantilla se pueden usar sin cualificar.

Plantillas de clases y funciones en C++

```
template <class TBV>
void Vector<TBV>::Inicia(int pri, int fin, TBV val) {
  ...
}

template <class TBV>
Vector<TBV> & Vector<TBV>::operator=(const Vector & der) {
  ...
}

template <class TBV>
void Vector<TBV>::ModificaCapacidad(int nueva_cap) {
  ...
}

template <class TBV>
void Vector<TBV>::Mostrar(ostream & fsal) const {
  ...
}          // El código de cada uno de estos métodos de la plantilla es el mismo
           // que había en su correspondiente método de la clase Vector
```

Plantillas de clases y funciones en C++

Plantillas de función

Se declaran y definen igual que las plantillas de clase:

- ▷ `template` más la lista de parámetros.
- ▷ Después se declara o define la plantilla como una función normal, incluyendo el uso de los parámetros de plantilla.

La instanciación de plantillas de función es distinta de las de clase.

- ▷ En principio, no es necesario dar la lista de argumentos de plantilla, porque puede deducirse de los tipos de los argumentos de función.
- ▷ Cuando sea necesario puede darse una lista explícita de argumentos de plantilla.

Plantillas de clases y funciones en C++

```
template <class T>
int SeparaIntervalo(T* vec, int n, const T & inf, const T & sup) {
  int ult = -1;  T tmp;
  for (int i=0 ; i < n ; i++)
    if (inf <= vec[i] && vec[i] <= sup)
      { ult++;  tmp = vec[ult];  vec[ult] = vec[i];  vec[i] = tmp; }
  return(ult);
}
...
char vc[500];
double * vd = new double[3000];
...
int u1 = SeparaIntervalo(vc,500,'g','m');
int u2 = SeparaIntervalo(vd,3000,34.0945,7772.8097021);
int u3 = SeparaIntervalo<double>(vd,3000,34.0945,7772.8097021);
```

Plantillas de clases y funciones en C++

Las plantillas se pueden sobrecargar, conjuntamente con las funciones normales.

▷ El compilador elegirá la más apropiada para una llamada entre las funciones y las instancias factibles de las plantillas, según los argumentos y sus tipos.

```
template <class T>
int SeparaIntervalo(Vector<T> & vec, const T & inf, const T & sup) {
  int ult = -1;  T tmp;
  for (int i=0 ; i < vec.Capacidad() ; i++)
    if (inf <= vec[i] && vec[i] <= sup)
      { ult++;  tmp = vec[ult];  vec[ult] = vec[i];  vec[i] = tmp; }
  return(ult);
}
...
long * vl1 = new long[1000];
Vector<long> vl2(2500);
...
int u4 = SeparaIntervalo(vl1,1000,999999,4999999);
int u5 = SeparaIntervalo(vl2,999999,4999999);
```

Plantillas de clases y funciones en C++

Amistad en las plantillas de clase

Conoceremos sólo la declaración de amistad de una plantilla de función en una de clase en la que ambas utilizan la **misma lista de parámetros de plantilla**.

```
template <class TBV> class Vector {
  friend ostream & operator<< <TBV>(ostream &, const Vector<TBV> &);
  ...
};
template <class Tipo>
ostream & operator<<(ostream & fsal, const Vector<Tipo> & v) {
  fsal << "[";
  if (v.capacidad > 0) {
    for (int i=0 ; i < v.capacidad-1 ; i++)
      fsal << v.elementos[i] << ", ";
    fsal << v.elementos[v.capacidad-1];
  }
  fsal << "]";
}
```

Una instancia de tipo específico de la plantilla de clase sólo da acceso a instancias del mismo tipo de la plantilla de función.

Plantillas de clases y funciones en C++

Cuando se instancie `Vector` con un determinado tipo, como por ejemplo:

```
struct Punto3D { double x, y, z; };
Vector<short> vec1;
Vector<Punto3D> vec2;
```

es responsabilidad del programador asegurarse de que ese tipo esté provisto del operador de salida <<, ya que es necesario en:

```
fsal << v.elementos[i] << ", ";
```

Así, no se obtendrá error de compilación al escribir:

```
cout << vec1;   ⇐ Correcto
cout << vec2;   ⇐ Error: Punto3D no tiene definido <<
```

Si el tipo no dispone del operador de salida <<, puede instanciarse `Vector`, y no habrá ninguna pega mientras no se utilice el operador de salida de `Vector`.

Plantillas de clases y funciones en C++

Tipos anidados en plantillas de clases

Se puede anidar la definición de una plantilla de clase dentro de la definición de otra.

```
template <class TBP>
class Pila {
 public:
  ...
 private:
  struct Nodo {
   TBP elem;    Nodo * sig;
   Nodo(const TBP& e, Nodo* s)
    { elem = e;  sig = s; }
  };
  typedef Nodo * Enlace;
  Enlace top;
  int talla;
  ...
};
```

La clase anidada se define de esta manera

```
template <class TBP>
class Pila {
 public:
  ...
 private:
  template <class TBP> struct Nodo {
   TBP elem;    Nodo * sig;
   Nodo(const TBP& e, Nodo* s)
    { elem = e;  sig = s; }
  };
  typedef Nodo<TBP> * Enlace;
  Enlace top;
  int talla;
  ...
};
```

... y debe entenderse como si se realizase esta asociación

Plantillas de clases y funciones en C++

▷ Aunque la definición de la plantilla anidada no va precedida de `template <class TBP>`, es una plantilla de manera automática y el parámetro `TBP` se puede utilizar dentro de su definición.

▷ Se asocia cada instanciación de la clase contenedora para un tipo específico con la instanciación de la clase anidada para el mismo tipo específico.

También se pueden anidar definiciones de tipos (`typedef`) y enumeraciones (`enum`) en una plantilla de clase.

▷ Un tipo público anidado puede referenciarse fuera de la plantilla, pero debe ser referenciado como miembro de una instancia, y no de la plantilla.

Por ejemplo, si `Enlace` fuese público debería referenciarse como:

```
Pila<int>::Enlace ptraux;
```

y no como:

```
Pila::Enlace ptraux;
```

Plantillas de clases y funciones en C++

Compilación de plantillas de clase

Cuando el compilador procesa la definición de una plantilla de clase se guarda una representación interna, pero no genera código correspondiente a la misma.

Cuando se instancia una plantilla de clase para un tipo específico, entonces sí se genera código correspondiente a esta instancia, utilizando la representación interna guardada previamente.

Para ello siempre es necesario que la definición de la plantilla haya sido procesada antes de que la plantilla tenga que instanciarse.

Para asegurar que esto se realiza en todo fichero que requiera una instanciación de la plantilla, conviene:

▷ colocar la definición de plantillas de clase en ficheros de cabecera `.h`, e

▷ incluirlos en los ficheros necesarios (como se hace con la definición de clases).

Plantillas de clases y funciones en C++

Respecto a los métodos, miembros de clase y tipos anidados de una plantilla de clase, el estándar C++ ofrece dos posibilidades:

▷ Añadir su definición a la definición de la plantilla de clase en el fichero de cabecera.

▷ Dar su definición en un fichero de código separado `.c++` (igual que se hace con la definición de los métodos, miembros de clase y tipos anidados de una clase).

Esta segunda posibilidad aún no es factible utilizarla. No está soportada por muchos compiladores, y por algunos sólo parcialmente.

Usaremos la primera forma de definición de miembros de la plantilla de clase.

▷ Aunque ello tiene la desventaja de no encapsular los detalles de la implementación, tendremos la ventaja de poder compilar directamente y de la misma forma en cualquier compilador.

Plantillas de clases y funciones en C++

Cuando se define una plantilla de clase o sus miembros, se debe tener en cuenta qué identificadores utilizados dependen de algún parámetro de la plantilla y cuáles no.

▷ Los nombres que no dependen de parámetros de la plantilla se resuelven al procesar la definición de la plantilla.

▷ Los que sí dependen de algún parámetro de la plantilla se resuelven al instanciar la plantilla para algún tipo específico.

En ambos casos, los identificadores utilizados deberán estar declarados o definidos antes de que se necesiten.

Plantillas de clases y funciones en C++

`vector`, de la STL de C++, ofrece las operaciones de `Vector` y muchas más.

```
#include <vector>
```

- ▷ Constructor: `vector<double> v(64), u, x(28,-13.987);`

  Se puede iniciar el vector con el valor dado como segundo argumento. Si no, lo inicia al valor por defecto del tipo.

- ▷ Destructor
- ▷ Constructor copia: `vector<double> w(v);`
- ▷ Asignación: `u = v;`
- ▷ Indexación (para lectura y escritura de elementos): `v[i] = w[j] + u[k];`
- ▷ Consulta de su talla: `v.size()`
- ▷ Modificación de su talla: `w.resize(17);` `x.resize(1024,-0.001);`

El TAD Cola de Prioridad

Una **cola de prioridad** es una generalización de los conceptos de pila y cola en la que, tras entrar en la cola, **la salida de los elementos** no se basa necesariamente en el orden de llegada sino que **se basa en un orden definido entre ellos**.

La extracción de elementos de una cola de prioridad puede buscar minimizar o maximizar el valor del elemento saliente.

▷ La interpretaremos en el sentido de minimizar. Para maximizar tan sólo hay que cambiar el sentido de las comparaciones.

En el modelo básico de una cola de prioridad la operaciones que consideramos son:

▷ insertar en la cola,

▷ obtener el elemento mínimo, y

▷ eliminar el elemento mínimo.

La realización de estas operaciones no requiere mantener ordenada (ni total ni parcialmente) la colección de elementos en la cola.

El TAD Cola de Prioridad

Algunas aplicaciones importantes de las colas de prioridad:

▷ **Gestión de procesos** en un sistema operativo. Los procesos no se ejecutan uno tras otro en base a su orden de llegada. Algunos procesos deben tener prioridad (por su mayor importancia, por su menor duración, etc.) sobre otros.

▷ Implementación de **algoritmos voraces**, los cuales proporcionan soluciones globales a problemas basándose en decisiones tomadas sólo con información local. La determinación de la mejor opción local suele basarse en una cola de prioridad.

  ◇ Algunos **algoritmos sobre grafos**, como la obtención de caminos o árboles de expansión de mínimo coste, son ejemplos representativos de algoritmos voraces basados en colas de prioridad.

▷ Implementaciones eficientes de **algoritmos de simulación** de sucesos discretos. En éstas, el avance del tiempo no se gestiona incrementando un reloj unidad a unidad, sino incrementado sucesivamente el reloj al instante del siguiente suceso.

El TAD Cola de Prioridad

TAD Cola de Prioridad

Elementos: En el conjunto de elementos $\mathcal{X}$ hay definido un **orden lineal** $<$:

$$\forall x, y \in \mathcal{X},\ x \neq y \Rightarrow (x < y \wedge y \not< x) \vee (y < x \wedge x \not< y).$$

$\mathcal{CP}$ es el conjunto de multiconjuntos finitos de elementos de $\mathcal{X}$.

(En un multiconjunto los elementos pueden repetirse. P.e.: $\{7, 4, 7, 2, 7, 4\}$.)

Operaciones: Dados $CP \in \mathcal{CP}$ y $x \in \mathcal{X}$:

Crear() : Devuelve $\emptyset$.

Destruir(CP) : Elimina CP.

EstáVacía(CP) : Devuelve **cierto** si $CP = \emptyset$, y **falso** si $CP \neq \emptyset$.

Longitud(CP) : Devuelve $|CP|$ (cardinalidad del multiconjunto).

El TAD Cola de Prioridad

Operaciones (cont.):

Mínimo(CP**)** : **Si** $CP \neq \emptyset$, **entonces** devuelve $y \in CP$ tal que $\forall z \in CP$, $y \leq z$; en otro caso, error.

Insertar(CP**,**x**)** : Devuelve $CP' = CP \cup \{x\}$.

(En la unión de multiconjuntos $A \cup B$, las ocurrencias de los elementos de A se añaden a las de B. P.e.: $\{7,4,7,2,7,4\} \cup \{4,5,1,4\} = \{7,4,7,2,7,4,4,5,1,4\}$.)

EliminarMín(CP**)** : **Si** $CP \neq \emptyset$, **entonces** devuelve $CP' = CP - \{$**Mínimo(**CP**)**$\}$; en otro caso, error.

(En la diferencia de multiconjuntos $A-B$, para cada elemento de A se descuentan tantas ocurrencias como haya en B de ese elemento. Si en B hay las mismas o más ocurrencias de un elemento que en A, éste no está en $A - B$. P.e.: $\{7,4,7,2,7,4\} - \{4,2,7,2,4\} = \{7,7\}$.)

El TAD Cola de Prioridad

Ejemplos:

$CP =$**Crear()**	**Insertar(**CP**,**32**)**	**Insertar(**CP**,**13**)**	**Mínimo(**CP**)**$= 13$
$\{\,\}$	$\{\,32\,\}$	$\{\,32, 13\,\}$	$\{\,32, 13\,\}$
EliminarMín(CP**)**	**Insertar(**CP**,**7**)**	**Insertar(**CP**,**26**)**	**Insertar(**CP**,**7**)**
$\{\,32\,\}$	$\{\,7, 32\,\}$	$\{\,7, 32, 26\,\}$	$\{\,7, 32, 7, 26\,\}$
Mínimo(CP**)**$= 7$	**EliminarMín(**CP**)**	**Insertar(**CP**,**18**)**	**Insertar(**CP**,**18**)**
$\{\,7, 32, 7, 26\,\}$	$\{\,7, 32, 26\,\}$	$\{\,7, 32, 26, 18\,\}$	$\{\,7, 18, 32, 26, 18\,\}$
Longitud(CP**)**$= 5$	**EliminarMín(**CP**)**	**EliminarMín(**CP**)**	**Mínimo(**CP**)**$= 18$
$\{\,7, 18, 32, 26, 18\,\}$	$\{\,18, 32, 26, 18\,\}$	$\{\,18, 32, 26\,\}$	$\{\,18, 32, 26\,\}$

El TAD Cola de Prioridad

Eficiencia de diferentes implementaciones básicas

Costes en el peor caso:	**Insertar**	**Mínimo**	**EliminarMín**
Lista	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Lista con acceso al mínimo	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Lista ordenada	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Árbol Binario de Búsqueda	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
• en el caso promedio:	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Árbol AVL (equilibrado)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Montículo binario (heap)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

El TAD Cola de Prioridad

Implementaciones eficientes de colas de prioridad

Operaciones adicionales requeridas habitualmente:

- ▷ **Formar** una cola de prioridad a partir de n elementos dados inicialmente.
- ▷ Cambiar la prioridad de un elemento dado de la cola.
- ▷ Eliminar un elemento dado de la cola (no el mínimo).
- ▷ **Combinar** dos colas para formar una única cola con los elementos de ambas.

Costes en el peor caso:	**Insertar**	**Mínimo**	**EliminarMín**	**Combinar**	**Formar**
Montículo binario (heap)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Montículo a la izquierda	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Cola binomial	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Otras implementaciones ofrecen costes asintóticos amortizados iguales o más eficientes (unitarios para algunas operaciones): montículos oblicuos y montículos de Fibonacci.

El montículo binario (heap)

El **montículo binario** es la implementación específica más básica y habitual de una cola de prioridad.

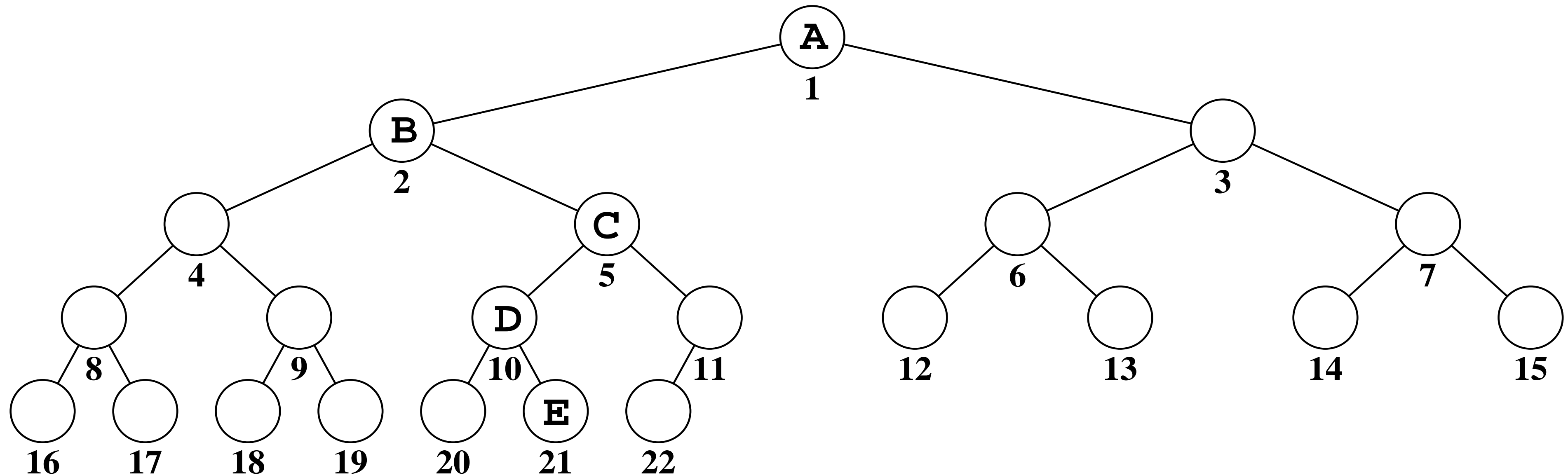
También es frecuente referirse a él por su nombre en inglés: "(binary) **heap**".

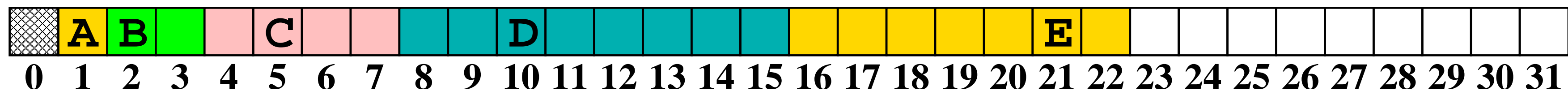
Un heap es un árbol binario con dos propiedades adicionales: estructural y de orden.

Estructura de heap. Un heap es un árbol binario casi completo con n nodos.

- ▷ Un árbol binario casi completo con n nodos se puede almacenar en un vector con n o más componentes.
- ▷ Numerando los nodos del árbol binario casi completo por niveles, de arriba a abajo y de izquierda a derecha, cada nodo i del heap se almacena en la posición i del vector.
- ▷ No se necesitan punteros para acceder a los hijos o al padre de un nodo.
- ▷ Principal problema: se debe estimar a priori el tamaño máximo del heap.

El montículo binario (heap)





Dado un nodo cualquiera i, $1 \leq i \leq n$:

▷ Su hijo izquierdo está en $2i$ (si tiene; es decir, si $2i \leq n$).

▷ Su hijo derecho está en $2i + 1$ (si tiene; es decir, si $2i + 1 \leq n$).

▷ Su padre está en $\lfloor i/2 \rfloor$ (si tiene; es decir, si $i > 1$).

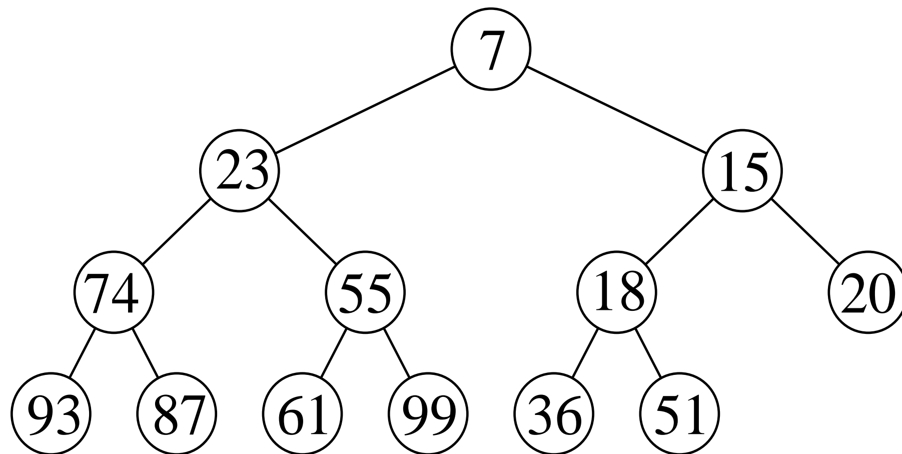
El montículo binario (heap)

Orden de heap. En un heap, el elemento situado en cada nodo es menor o igual que el elemento situado en cualquiera de sus hijos (si el nodo tiene algún hijo).

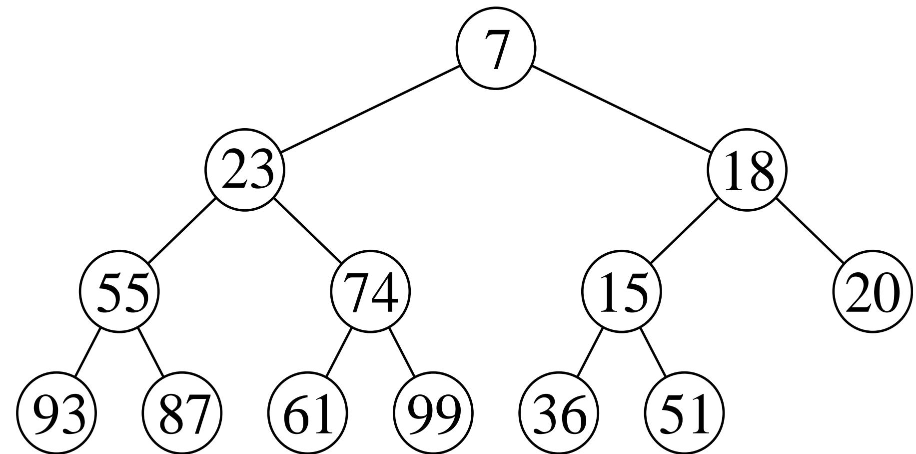
▷ El elemento mínimo siempre está en la raíz.

▷ En cada subárbol se verifica el orden de heap.

▷ El elemento máximo está en una hoja.



heap



no es heap

El montículo binario (heap)

```
1  #include "errores.h"                                // Plantilla  heap.h
2  #include <vector>
3  using namespace std;
4  template <class TBH>     // TBH debe tener definido el operador <
5  class Heap {
6    public:
7      explicit Heap(int maxelems=1000)
8        { maxheap=maxelems; elems=new TBH[maxheap+1]; talla=0; }
9      ~Heap() { delete [] elems; }
10     Heap(const Heap & der) { elems = 0; operator=(der); }
11     Heap & operator=(const Heap & der);
12     void Vaciar() { talla = 0; }
13     bool EstaVacio() const { return(talla == 0); }
14     bool EstaLleno() const { return(talla == maxheap); }
15     int Longitud() const { return(talla); }
16     const TBH & Minimo() const;
17     void Insertar(const TBH & item);
18     void EliminarMin();
19     void FormarHeap(const vector<TBH> & items);
```

El montículo binario (heap)

```
20   private:
21     TBH *elems;
22     int talla, maxheap;
23     void Hundir(int hueco);
24 };

25 template <class TBH>
26 Heap<TBH> &
27 Heap<TBH>::operator=(const Heap & der) {
28   if (this != &der) {
29     delete [] elems;
30     maxheap = der.maxheap;
31     elems = new TBH[maxheap+1];
32     talla = der.talla;
33     for (int i=1 ; i <= talla ; i++)
34       elems[i] = der.elems[i];
35   }
36   return(*this);
37 }
```

El montículo binario (heap)

Minimo.

```
38 template <class TBH>
39 const TBH &
40 Heap<TBH>::Minimo() const {
41   if (talla == 0) Error("Minimo");
42   return(elems[1]);
43 }
```

Tamaño del problema
n: número de elementos en el heap

$$t_{\texttt{Minimo}}(n) = \Theta(1)$$

Insertar.

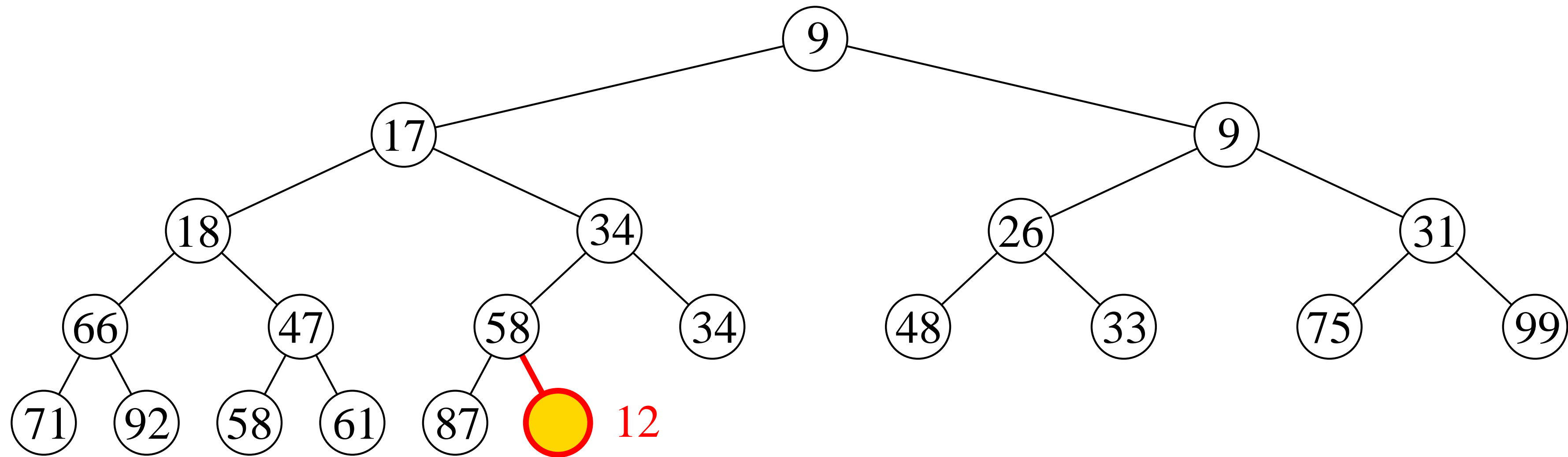
En un heap de n elementos, al insertar uno nuevo se deberá utilizar la posición $n+1$ del vector para mantener la estructura de heap.

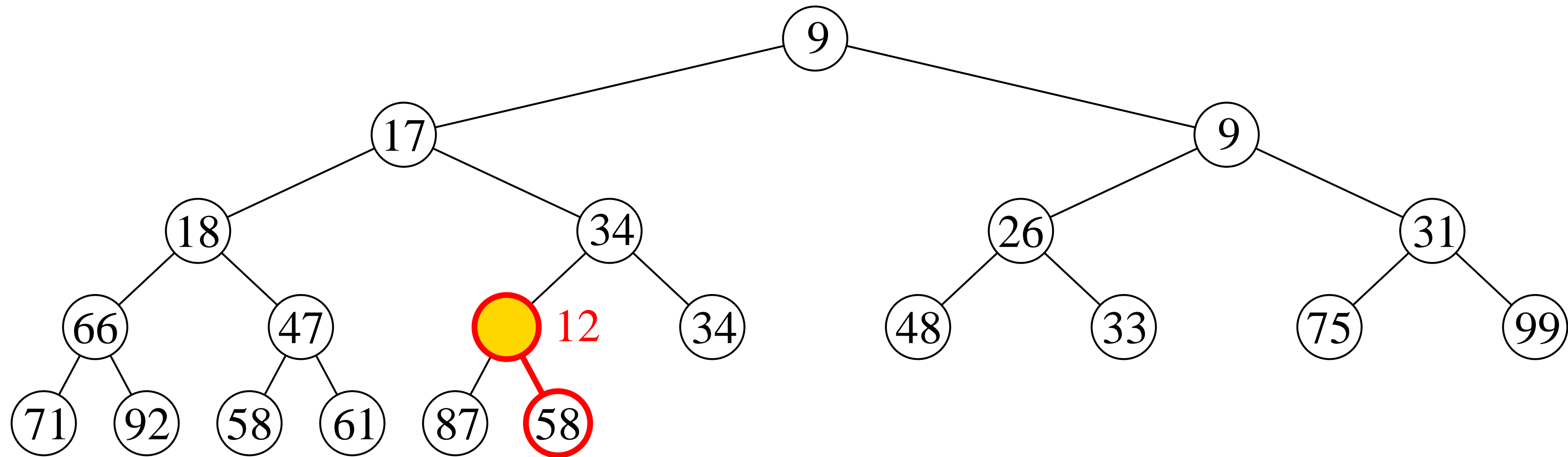
Si se preserva el orden de heap al situarlo en el hueco (nodo $n+1$), fin de la inserción.

Si no, el nuevo elemento será menor que el elemento que está en el padre del hueco.
$\Rightarrow$ Se baja el elemento del nodo padre al hueco y ahora el hueco queda en el padre.

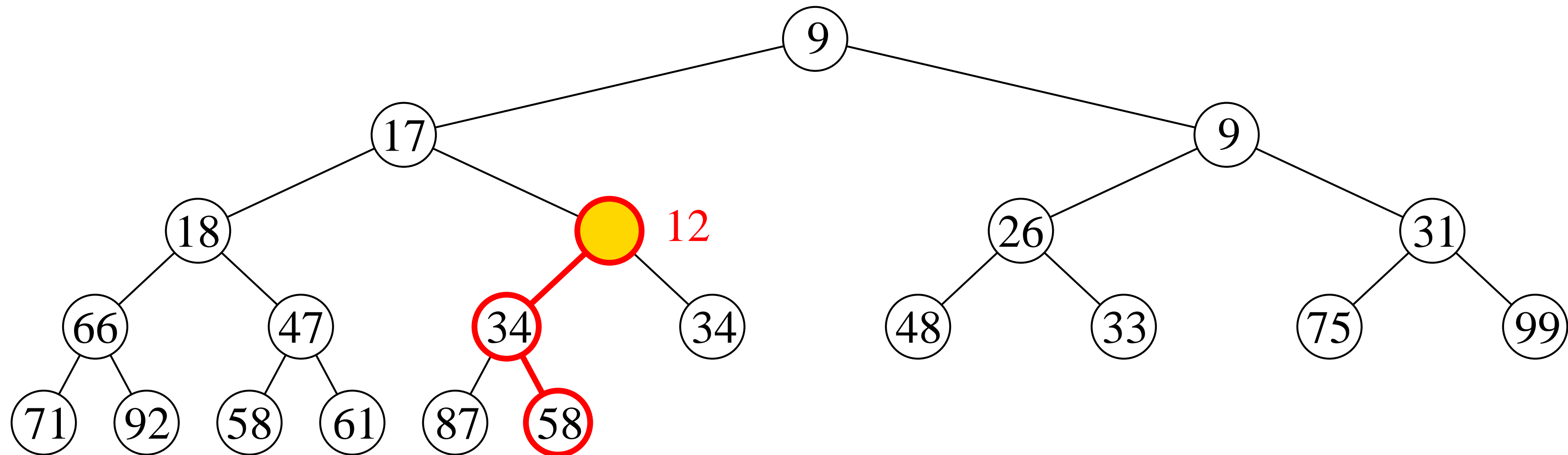
Este proceso se repite hasta colocar el nuevo elemento en un hueco. En el peor caso, el hueco ascenderá hasta la raíz.

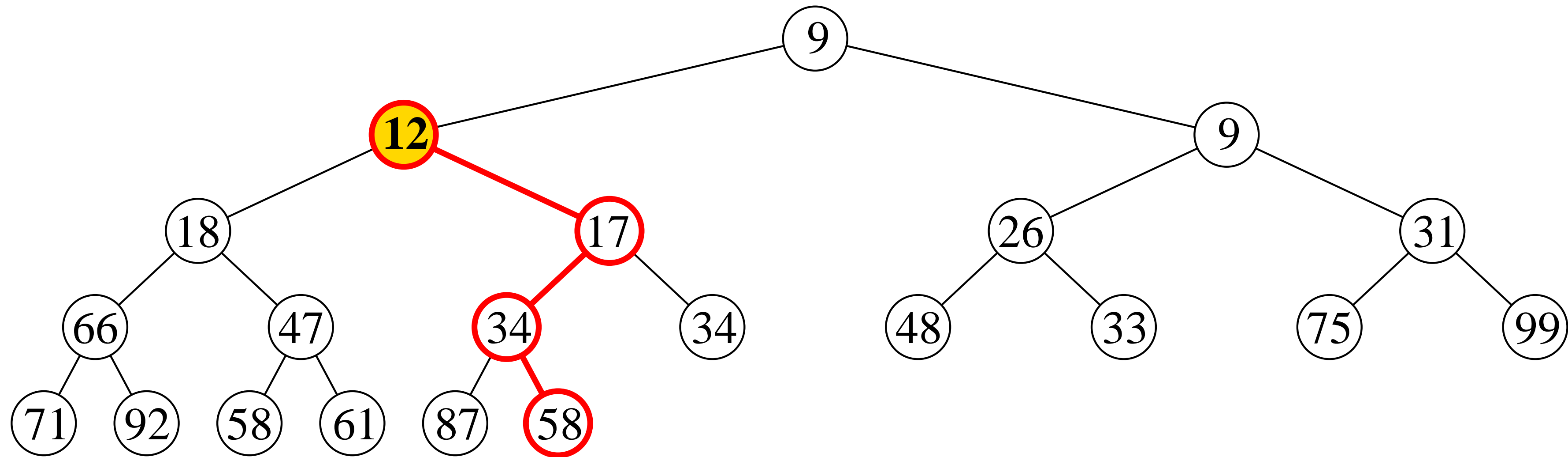
El montículo binario (heap)





El montículo binario (heap)





El montículo binario (heap)

Partiendo desde el nodo $n+1$ al insertar, la violación del orden de heap sólo puede aparecer en el camino desde este nodo a la raíz, e irá trasladándose a lo largo de él.

Si el elemento de i es menor que el de $\lfloor i/2 \rfloor$, al intercambiarlos queda el menor como padre de dos mayores. El del hermano ($i-1$ ó $i+1$) ya era mayor que el del padre.

Si el elemento de i debe seguir subiendo a $\lfloor \lfloor i/2 \rfloor /2 \rfloor$, el que baje a $\lfloor i/2 \rfloor$ será menor que los que había en $\lfloor i/2 \rfloor$ e $i-1$ ó $i+1$. Al inicio, el orden de heap se cumplía.

```
44 template <class TBH>
45 void
46 Heap<TBH>::Insertar(const TBH & item) {
47   if (talla == maxheap) Error("Insertar");
48   talla++;
49   int hueco;
50   for (hueco=talla ; hueco > 1 && item < elems[hueco/2] ; hueco /= 2)
51     elems[hueco] = elems[hueco/2];
52   elems[hueco] = item;
53 }
```

En promedio, el coste de insertar es constante.

$$t_{\texttt{Insertar}}(n) = \Omega(1), \mathcal{O}(\log n)$$

El montículo binario (heap)

EliminarMin.

En un heap de n elementos, al eliminar uno se deberá inutilizar la posición n del vector, dejándolo con $n - 1$ elementos, para mantener la estructura de heap.

Como el elemento que se elimina es el mínimo y éste está en la raíz, inicialmente tenemos un hueco en la raíz y el elemento que estaba en la posición n por recolocar.

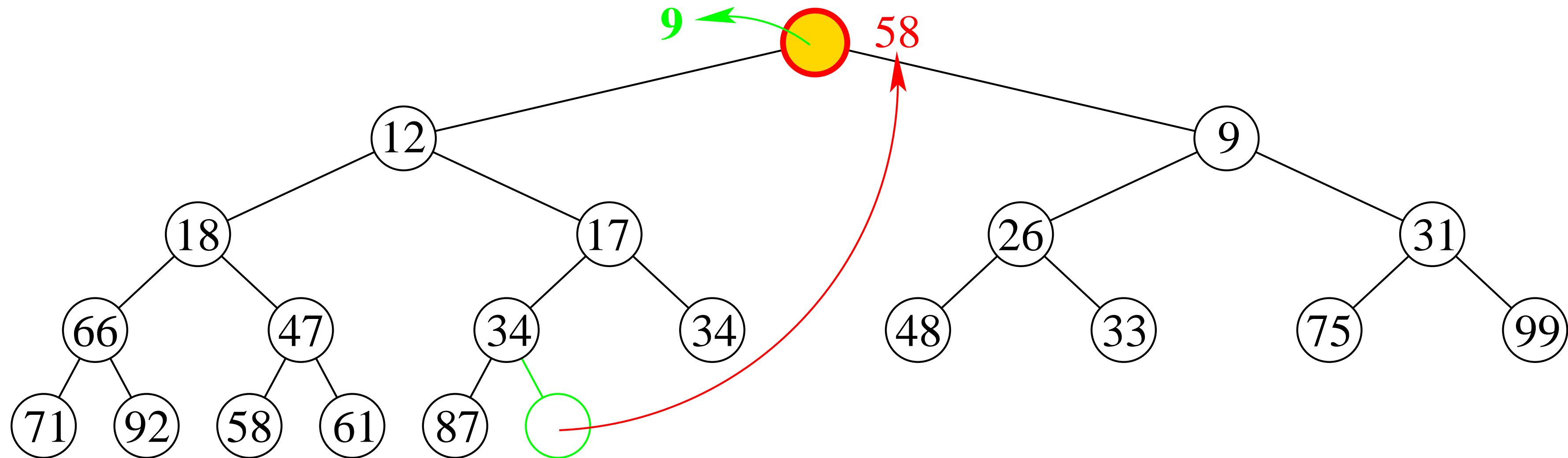
Si se preserva el orden de heap al situar el elemento en el hueco, fin de la eliminación.

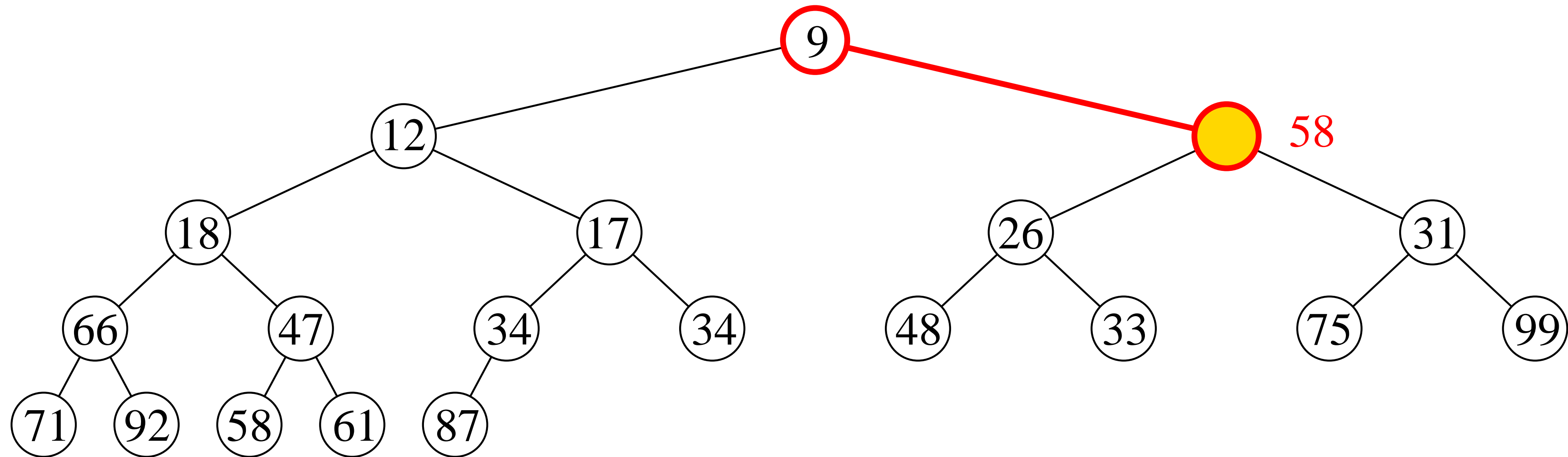
Si no (lo más probable), el elemento por recolocar será mayor que alguno de los elementos que están en los hijos del hueco.

$\Rightarrow$ Se sube el menor de los elementos de los hijos al hueco y ahora el hueco queda en el nodo hijo en el que estaba este menor elemento.

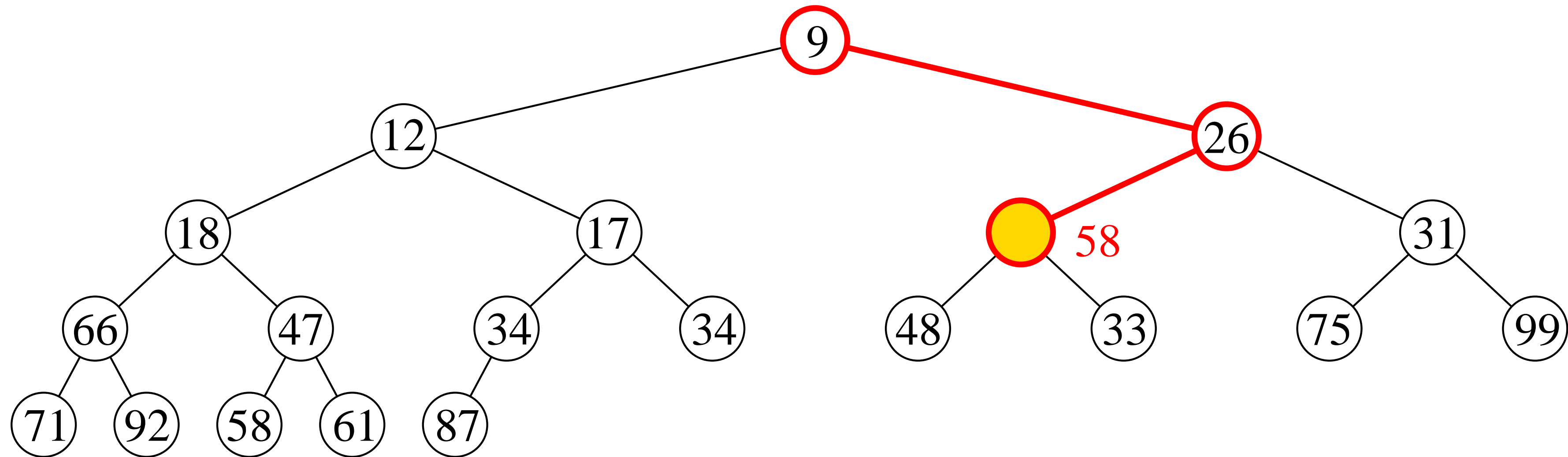
Este proceso se repite hasta recolocar el elemento pendiente en un hueco. En el peor caso, el hueco descenderá hasta las hojas.

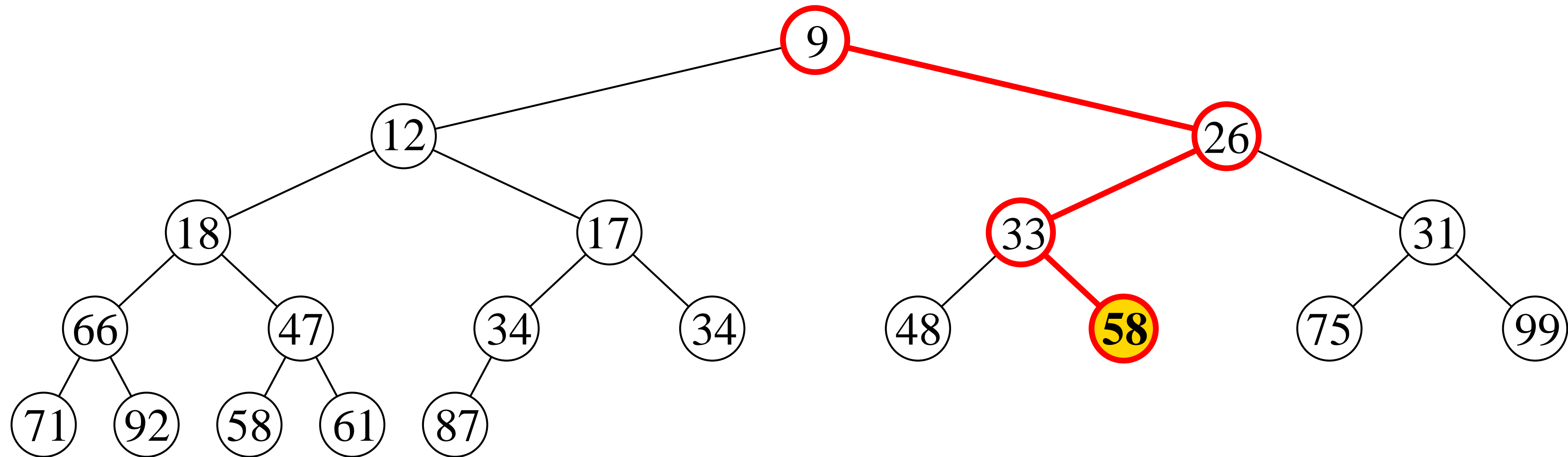
El montículo binario (heap)





El montículo binario (heap)





El montículo binario (heap)

Al eliminar, la violación del orden de heap sólo puede aparecer en el camino desde la raíz a las hojas, e irá trasladándose a lo largo de él.

Si el elemento de i es mayor que el de $2i$ y/o el de $2i+1$, al intercambiarlo con el menor de ellos queda el menor como padre de dos mayores.

Si el elemento de i debe seguir bajando a $2(2i)$, $2(2i)+1$, $2(2i+1)$ ó $2(2i+1)+1$, el que suba a $2i$ ó $2i+1$ será mayor que el que ahora está en i (antes en $2i$ ó $2i+1$). Al inicio, el orden de heap se cumplía.

```
54 template <class TBH>
55 void
56 Heap<TBH>::EliminarMin() {
57   if (talla == 0) Error("EliminarMin");
58   elems[1] = elems[talla];
59   talla--;
60   Hundir(1);
61 }
```

$$t_{\texttt{EliminarMin}}(n) \;=\; \Omega(1), \mathcal{O}(\log n)$$

El montículo binario (heap)

El método privado `Hundir` permite unificar la implementación de `EliminarMin` y `FormarHeap`. Ambas operaciones del heap se basan en el mismo proceso.

```
62 template <class TBH>
63 void
64 Heap<TBH>::Hundir(int hueco) {
65   TBH tmp = elems[hueco];
66   int hijo;
67   while (true) {
68     hijo = hueco*2;
69     if (hijo < talla && elems[hijo+1] < elems[hijo]) hijo++;
70     if (hijo > talla || !(elems[hijo] < tmp)) break;
71     elems[hueco] = elems[hijo];
72     hueco = hijo;
73   }
74   elems[hueco] = tmp;
75 }
```

$$t_{\texttt{Hundir}}(n) = \Omega(1), \mathcal{O}(\log n)$$

El montículo binario (heap)

FormarHeap.

Disponiendo de n elementos inicialmente, se puede formar un heap con ellos mediante n inserciones sucesivas partiendo de un heap vacío.

▷ Coste en el peor caso: $\mathcal{O}(n \log n)$

▷ Coste en el caso promedio: $\mathcal{O}(n)$

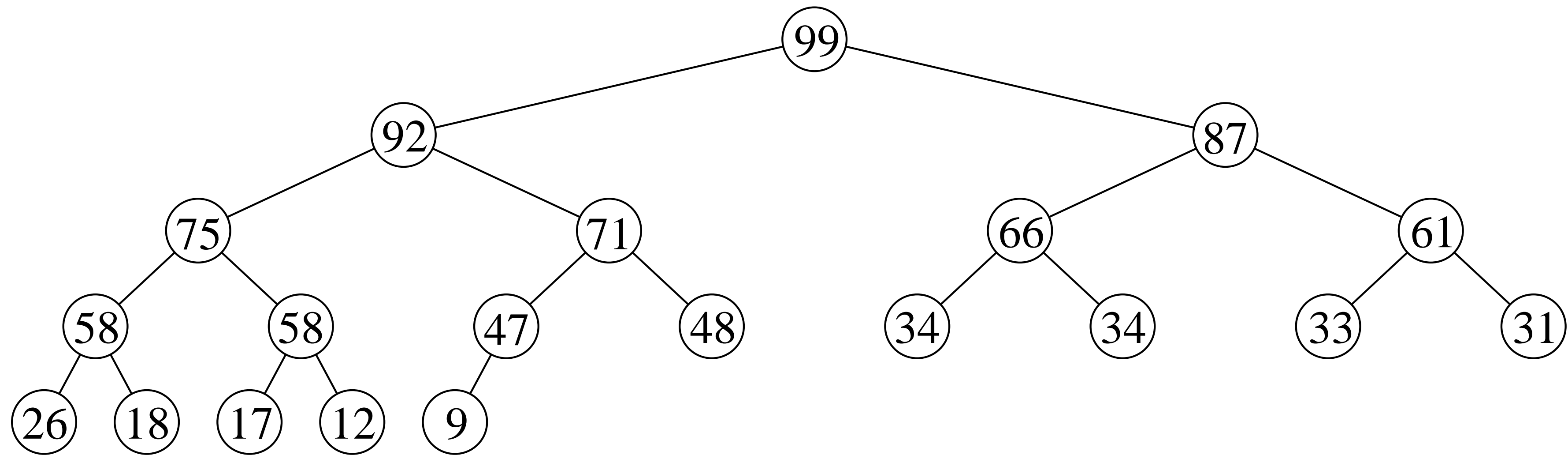
Aproximación alternativa:

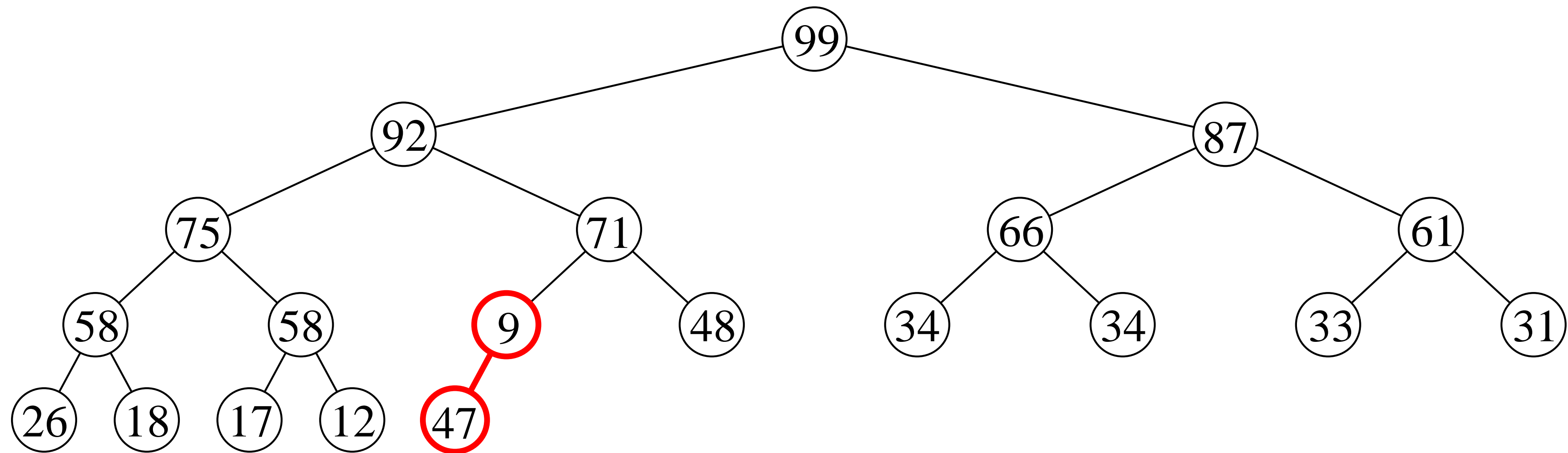
▷ Situar los elementos arbitrariamente en las n primeras posiciones del vector `elems` $\Rightarrow$ se crea un árbol binario con estructura de heap, aunque no con orden de heap.

▷ Considerando los nodos hoja como heaps de 1 nodo, comenzando desde el nodo $\lfloor n/2 \rfloor$ y procediendo nodo a nodo decrecientemente hasta la raíz, establecer el orden de heap en el subárbol con raíz en el nodo considerado.

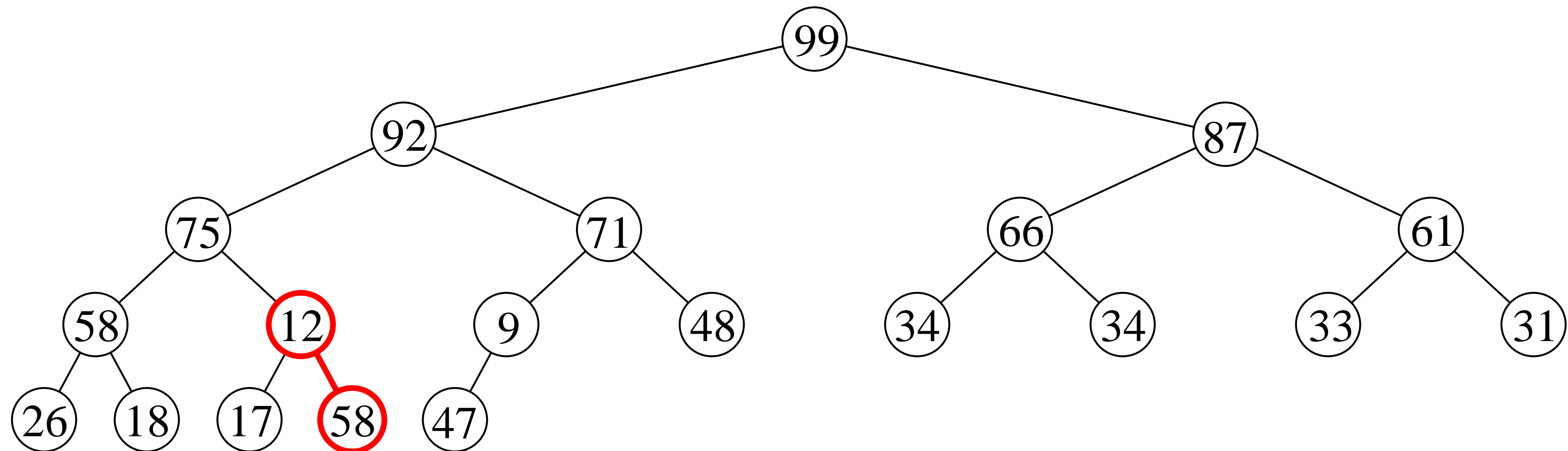
Para ello, se debe hundir el elemento del nodo considerado en su subárbol hasta que alcance su nivel adecuado.

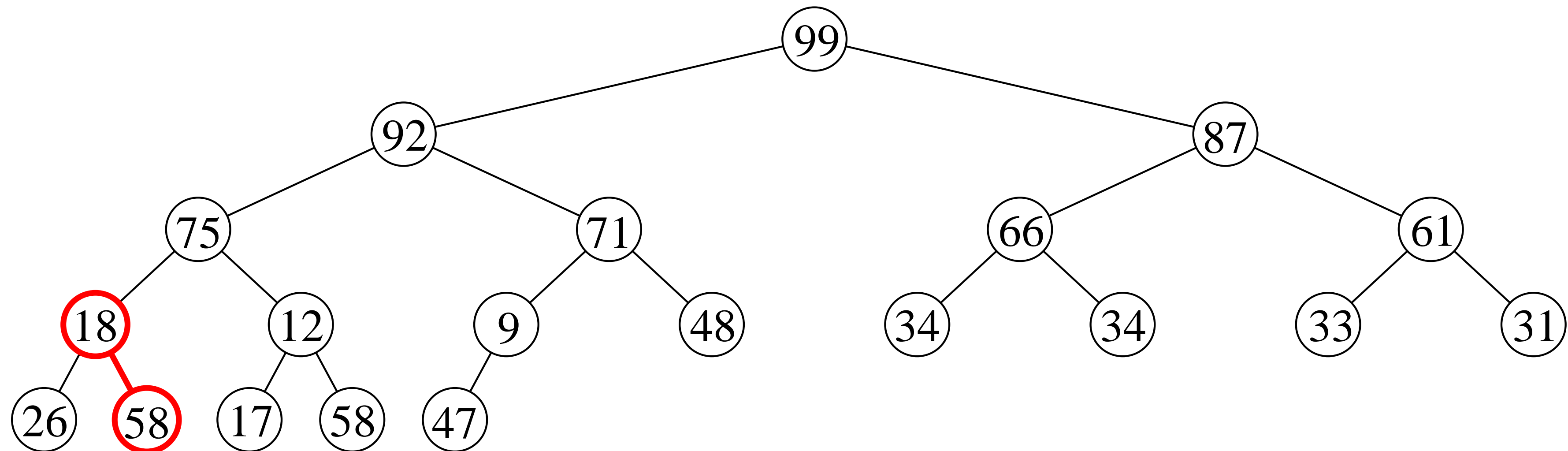
El montículo binario (heap)



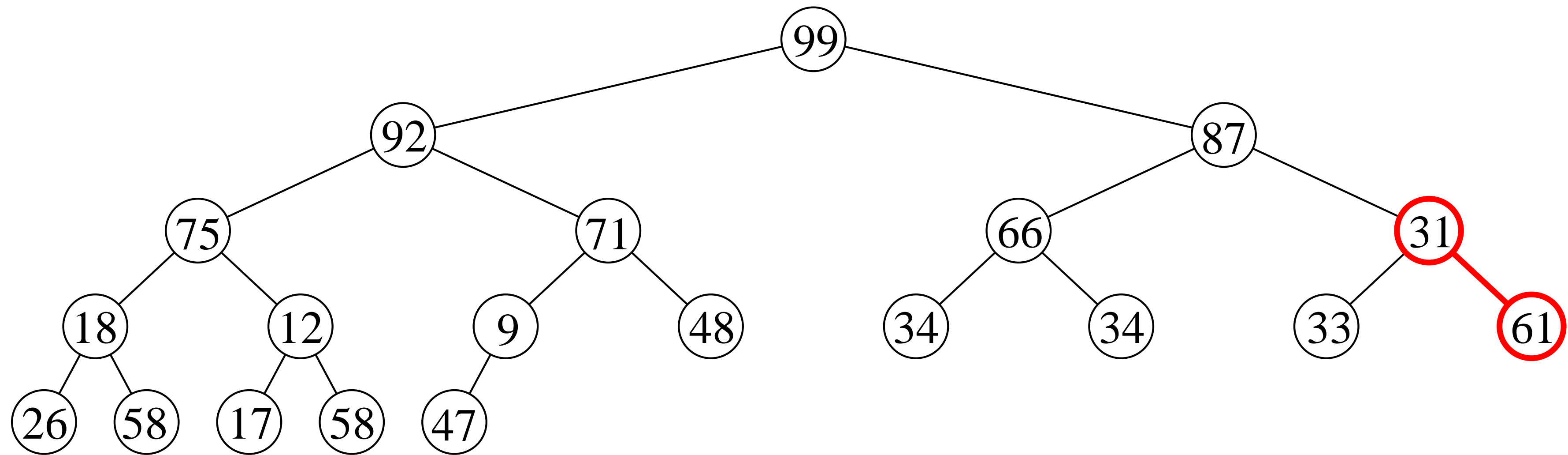


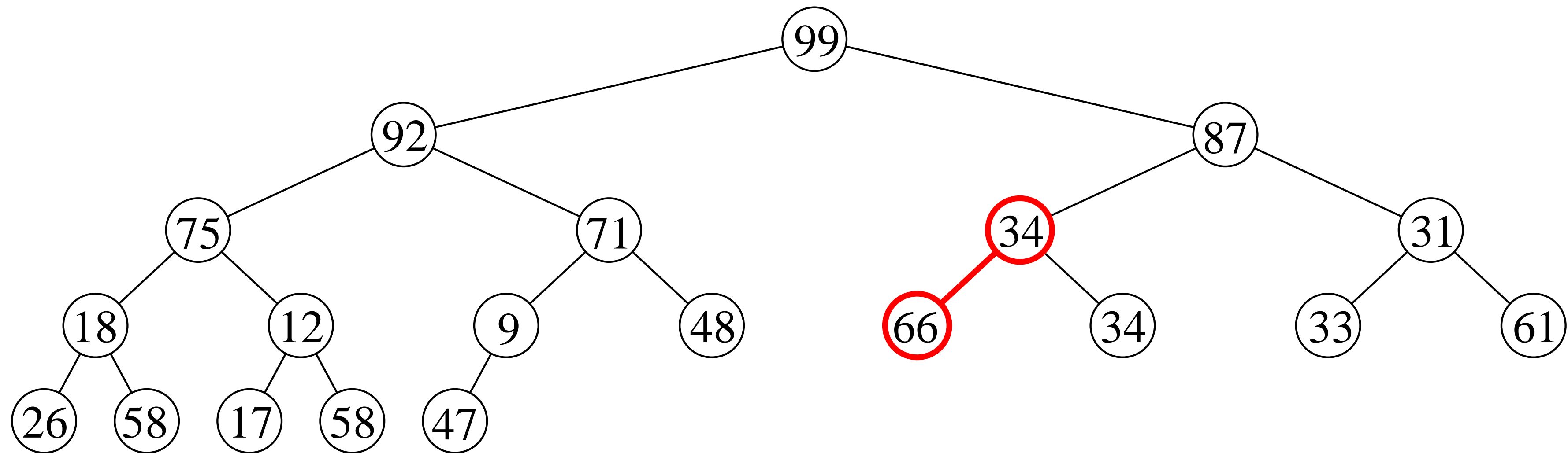
El montículo binario (heap)



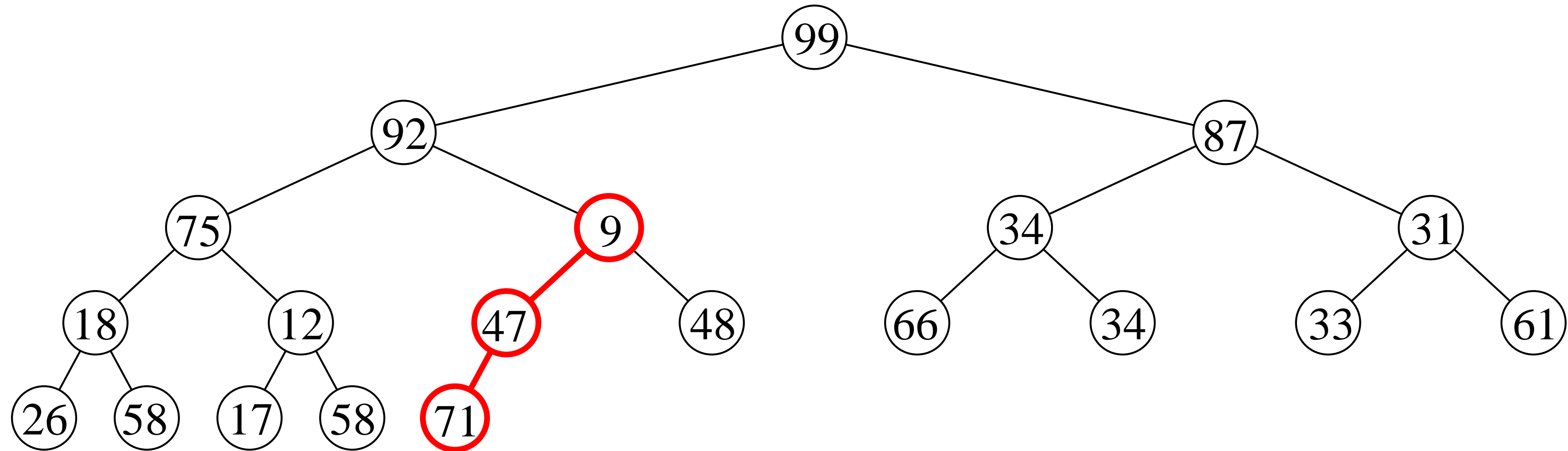


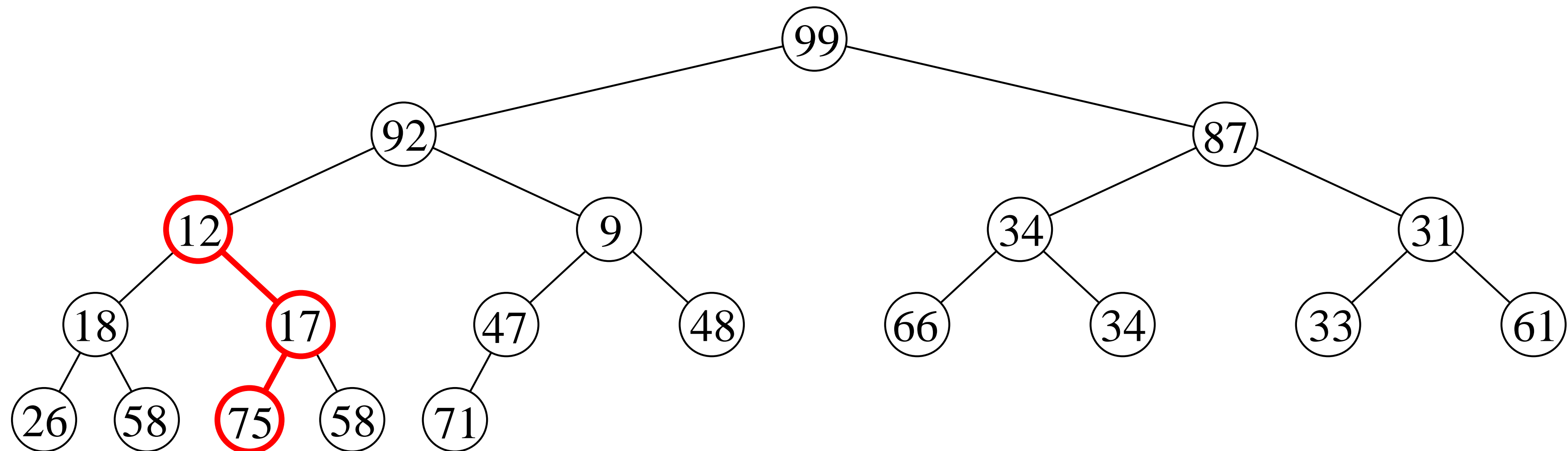
El montículo binario (heap)



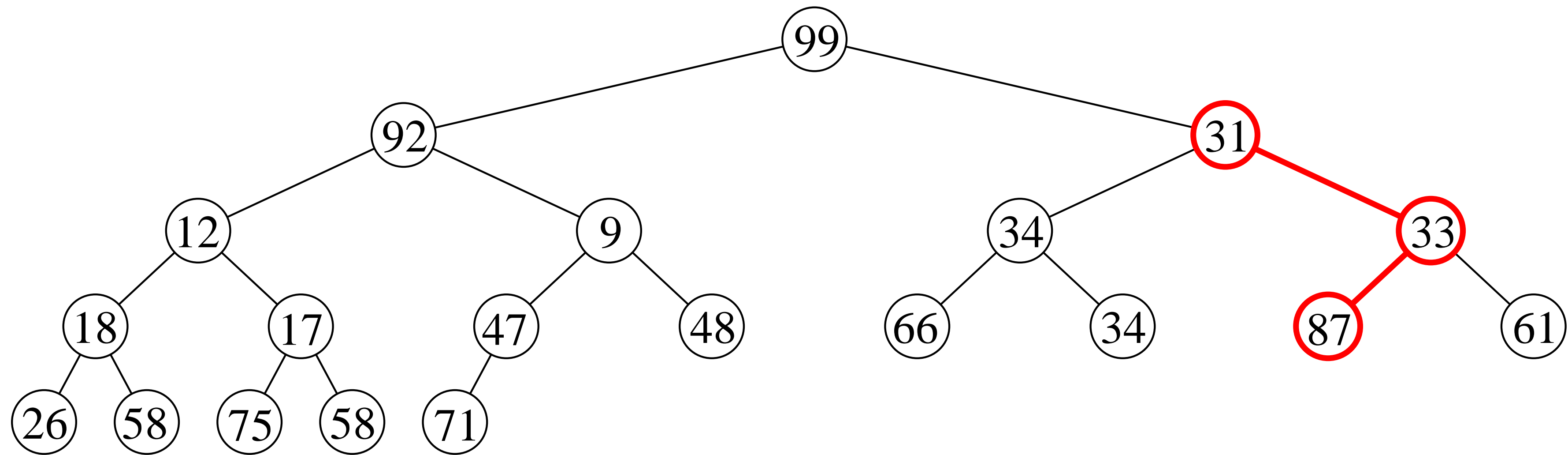


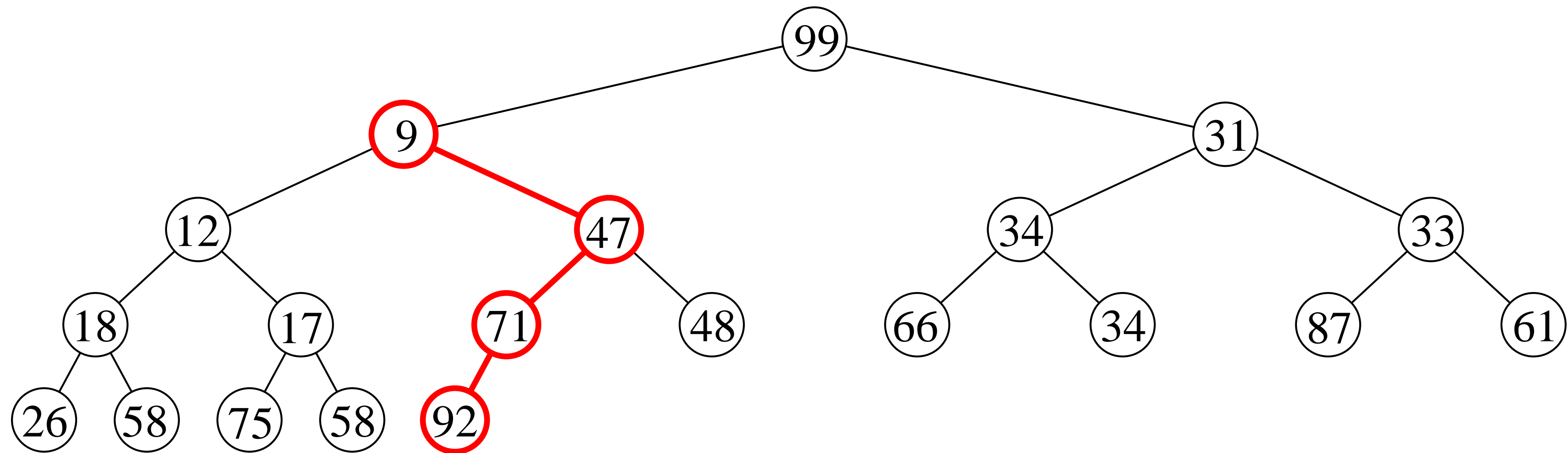
El montículo binario (heap)



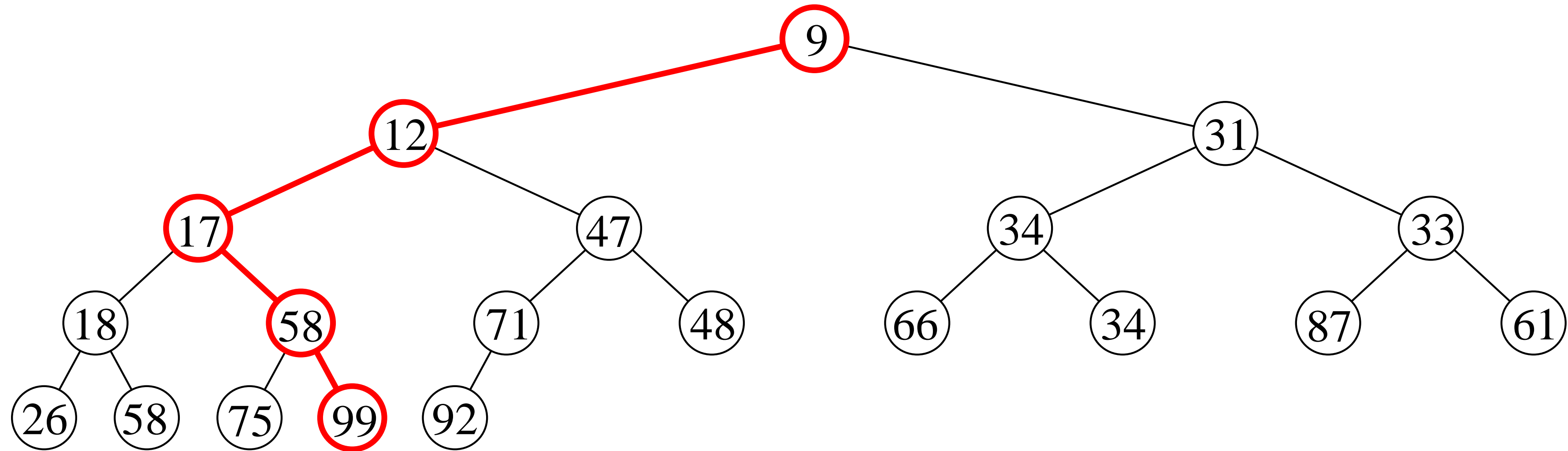


El montículo binario (heap)





El montículo binario (heap)




```
76 template <class TBH>
77 void
78 Heap<TBH>::FormarHeap(const vector<TBH> & items) {
79   if (maxheap < items.size()) Error("FormarHeap");
80   for (talla=0 ; talla < items.size() ; talla++)
81     elems[talla+1] = items[talla];
82   for (int i=talla/2 ; i > 0 ; i--)
83     Hundir(i);
84 }
```

$$t_{\texttt{FormarHeap}}(n) = \Theta(n)$$

El montículo binario (heap)

Breve análisis del coste de `FormarHeap`.

Mejor caso. Desde $\lfloor n/2 \rfloor$ hasta 1, el elemento de cada nodo considerado es el menor de su subárbol $\Rightarrow$ el número de asignaciones (desplazamientos de elementos) realizadas por `Hundir` es $2 \cdot \lfloor n/2 \rfloor$. Por tanto, es $\Omega(n)$.

Peor caso. Desde $\lfloor n/2 \rfloor$ hasta 1, el elemento de cada nodo considerado debe hundirse hasta una de las hojas más profundas de su subárbol.

El número de asignaciones (desplazamientos de elementos) realizadas por `Hundir` es $2 \cdot \lfloor n/2 \rfloor + S$, siendo S la suma de las alturas de los nodos y $S < n$. Por tanto, es $\mathcal{O}(n)$.

Por ejemplo, denominando h_i a la altura del nodo i, en el heap anterior:

$$S = \sum_{i=1}^{10} h_i = 4 + 3 + 2 + 2 + 2 + 1 + 1 + 1 + 1 + 1 = 18 < 20 = n$$

$$h_i = 0, \ 11 \leq i \leq 20$$

El montículo binario (heap)

Aplicación: Códigos de Huffman

Una secuencia de símbolos se puede codificar en binario estableciendo a priori el conjunto de símbolos (alfabeto) y asignándoles **códigos binarios de longitud fija**.

Por ejemplo, el estándar ASCII utiliza 7 bits para codificar 128 dígitos decimales, letras mayúsculas y minúsculas, símbolos de puntuación y otros caracteres.

La secuencia `CD54` se codifica en ASCII como 1000011100010001101010110100.

Si el número de símbolos es n, los códigos deben tener una longitud fija de $\lceil \log_2 n \rceil$ bits para disponer de suficientes códigos.

Generalmente, esta codificación produce secuencias de bits (ficheros) de mayor tamaño que otras, desaprovechando espacio de almacenamiento o tiempo de transmisión.

Asignando **códigos binarios de longitud variable** a los símbolos, de manera que los más frecuentes tengan códigos más cortos y los más infrecuentes más largos, se puede reducir el tamaño de las secuencias codificadas en binario (ficheros).

El montículo binario (heap)

Como ejemplo, consideremos un fichero compuesto sólo por los siguientes símbolos, que aparecen el número de veces indicado.

símbolos	frecuencia	código 1 long. fija	número de bits	código 2 long. variable	número de bits
4	8	000	24	000	24
5	22	001	66	001	66
C	25	010	75	010	75
D	23	011	69	011	69
E	15	100	45	10	30
F	7	101	21	11	14
número total de bits del fichero:			300		278

Sin aprovechar la información de las frecuencias, ya se obtiene ganancia al reducir a dos bits el código de dos símbolos.

El montículo binario (heap)

Un código binario puede representarse mediante un **árbol binario de codificación**.

▷ Los símbolos se sitúan en los nodos.

▷ En cada nodo no terminal, la rama al hijo izquierdo representa un `0` y la rama al hijo derecho un `1`.

▷ El código de cada símbolo se corresponde con el camino desde la raíz al nodo que lo contiene.

código 1

código 2

El montículo binario (heap)

Un árbol de codificación que tenga nodos sin símbolo y con un sólo hijo produce secuencias codificadas más largas que las que produce otro en el que estos nodos se fusionan con sus únicos hijos. Así, todos los nodos tienen dos hijos o ninguno.

Si todos los símbolos están en las hojas, ningún código es prefijo de otro.

▷ Una codificación con esta propiedad es un **código libre de prefijos** ("prefix code").

Así, una secuencia de bits se **descodifica de forma no ambigua e instantánea**.

▷ La secuencia `101001001100000101011` es no ambigua para los códigos 1 y 2.

Descodificación con código 1: `F 5 5 E 4 F D`

`1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 1 0 1 0 1 1`

Descodificación con código 2: `E E C D 4 5 C F`

▷ Código ambiguo: (`4`, `000`), (`5`, `001`), (`C`, `010`), (`D`, `011`), (`E`, `00`), (`F`, `01`).

¿Cómo se descodifica `010000`? ¿`C4` = `010-000` ó `FEE` = `01-00-00`?

El montículo binario (heap)

Llamando $p(a)$ a la profundidad de la hoja en la que está el símbolo a del alfabeto A y $f(a)$ a su frecuencia, el **coste de un árbol binario de codificación** para A es:

$$\sum_{a \in A} p(a) \cdot f(a)$$

Este coste coincide con el número total de bits calculado anteriormente.

Está claro que podemos definir distintos códigos binarios de longitud variable, para un conjunto de símbolos, que tendrán distinto coste.

Dado un conjunto de símbolos A y sus frecuencias en un fichero, ¿podemos encontrar un **árbol binario de codificación óptimo** para A, es decir, cuyo coste sea mínimo?

De esta manera, obtendríamos un código binario óptimo que necesitaría el menor número total de bits para representar el fichero.

Este código es óptimo para una codificación del tipo "símbolo - código de longitud variable (libre de prefijos)", dados un alfabeto y las frecuencias de sus símbolos. Otros tipos de codificación pueden reducir más el número de bits empleados.

El montículo binario (heap)

Algoritmo de Huffman

Trabaja con un **bosque**, que es una colección de árboles.

Inicialmente, cada par (símbolo,frecuencia) forma un árbol de un sólo nodo en el bosque.

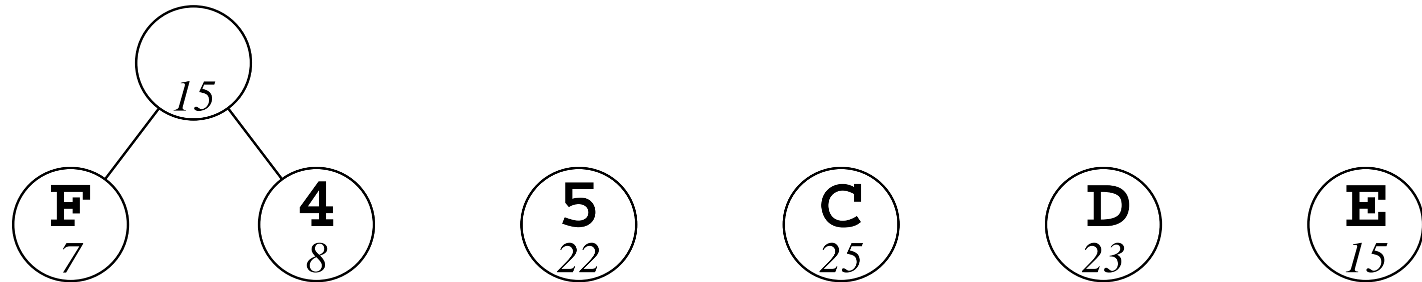
Iterativamente, hasta que el bosque tenga un sólo árbol:

- ▷ Se seleccionan los dos árboles con menor coste (da lo mismo cómo se resuelvan los empates).
- ▷ Se forma un nuevo árbol binario, creando un nuevo nodo raíz que contendrá:
  - ◇ como coste la suma de los costes de los dos árboles seleccionados, y
  - ◇ como hijos a dichos árboles (da lo mismo cuál sea el izquierdo y cuál el derecho).

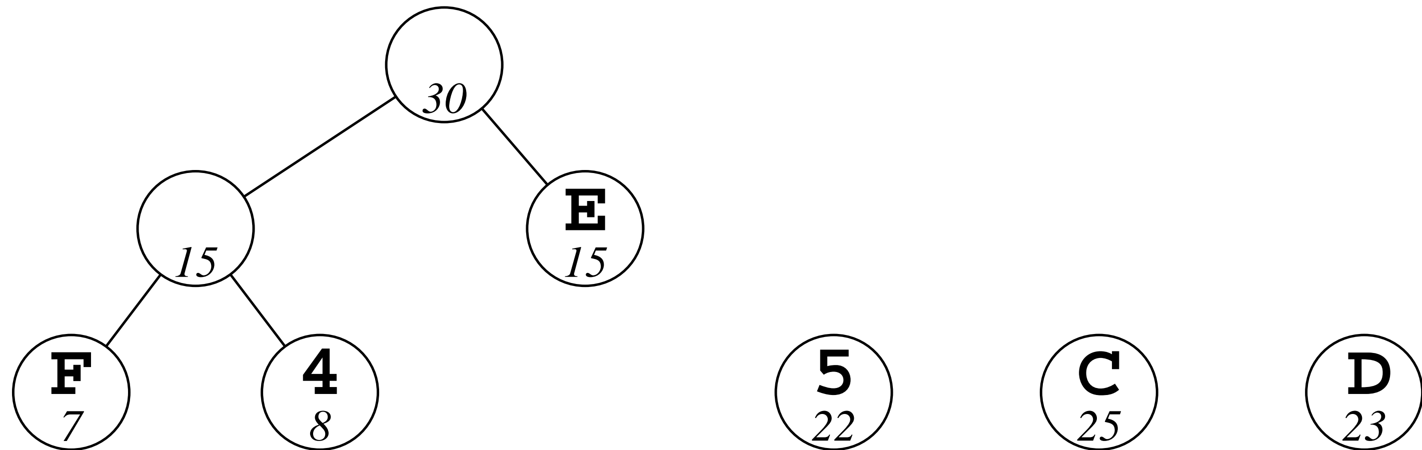
El montículo binario (heap)

Inicio:

Iteración 1:

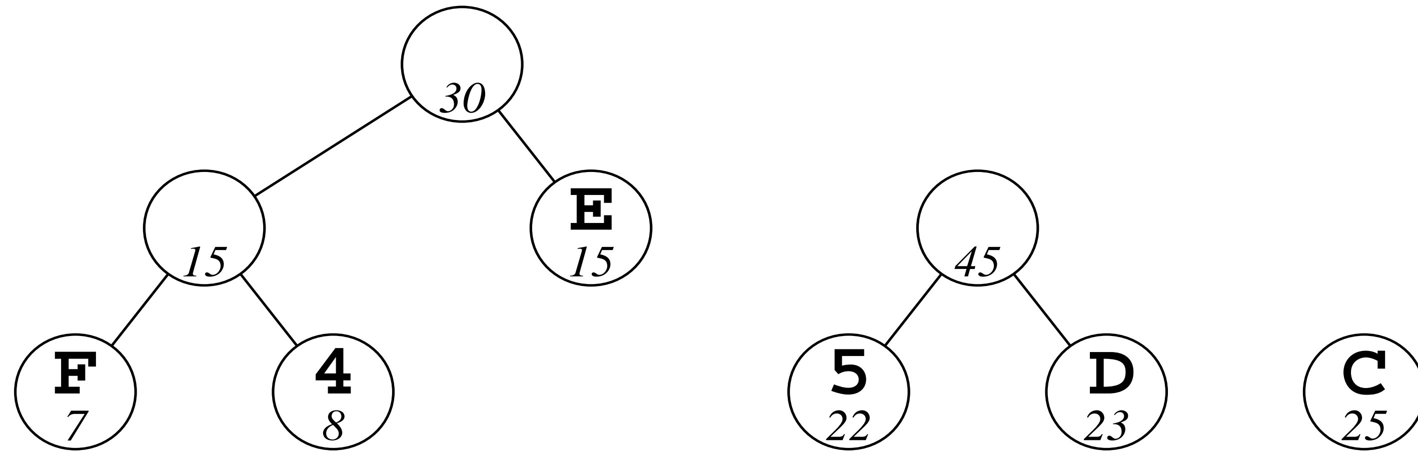


Iteración 2:

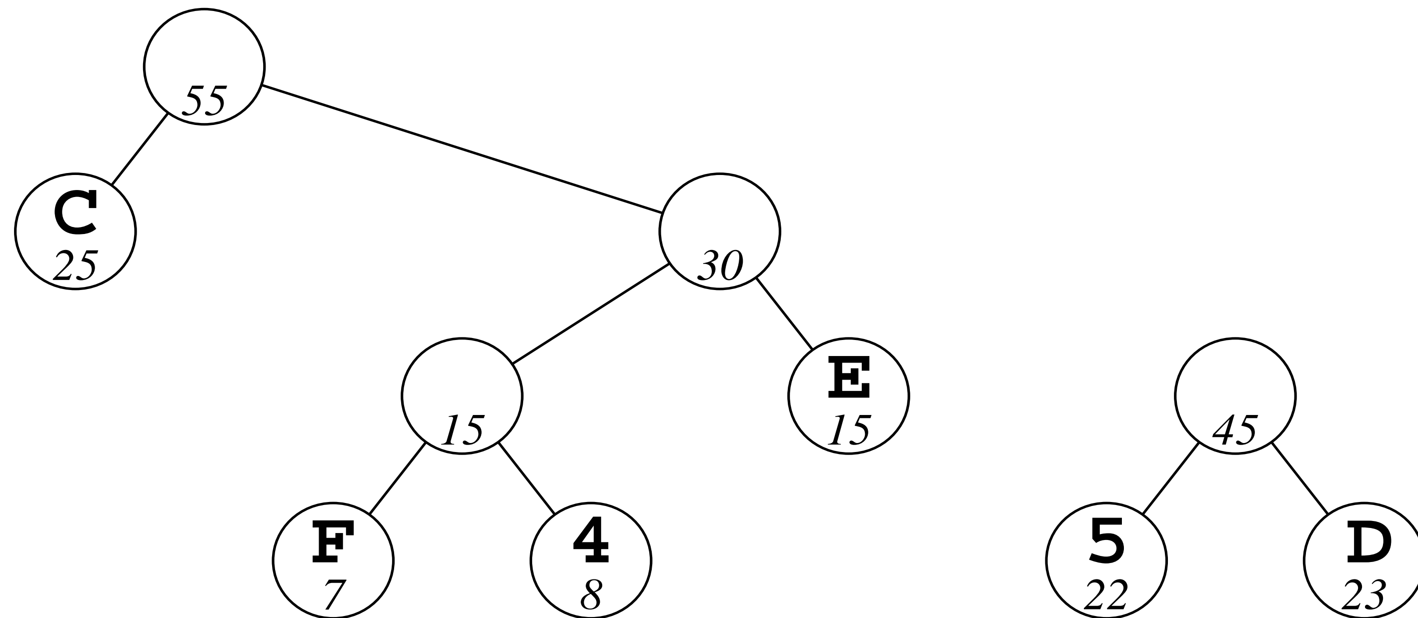


El montículo binario (heap)

Iteración 3:

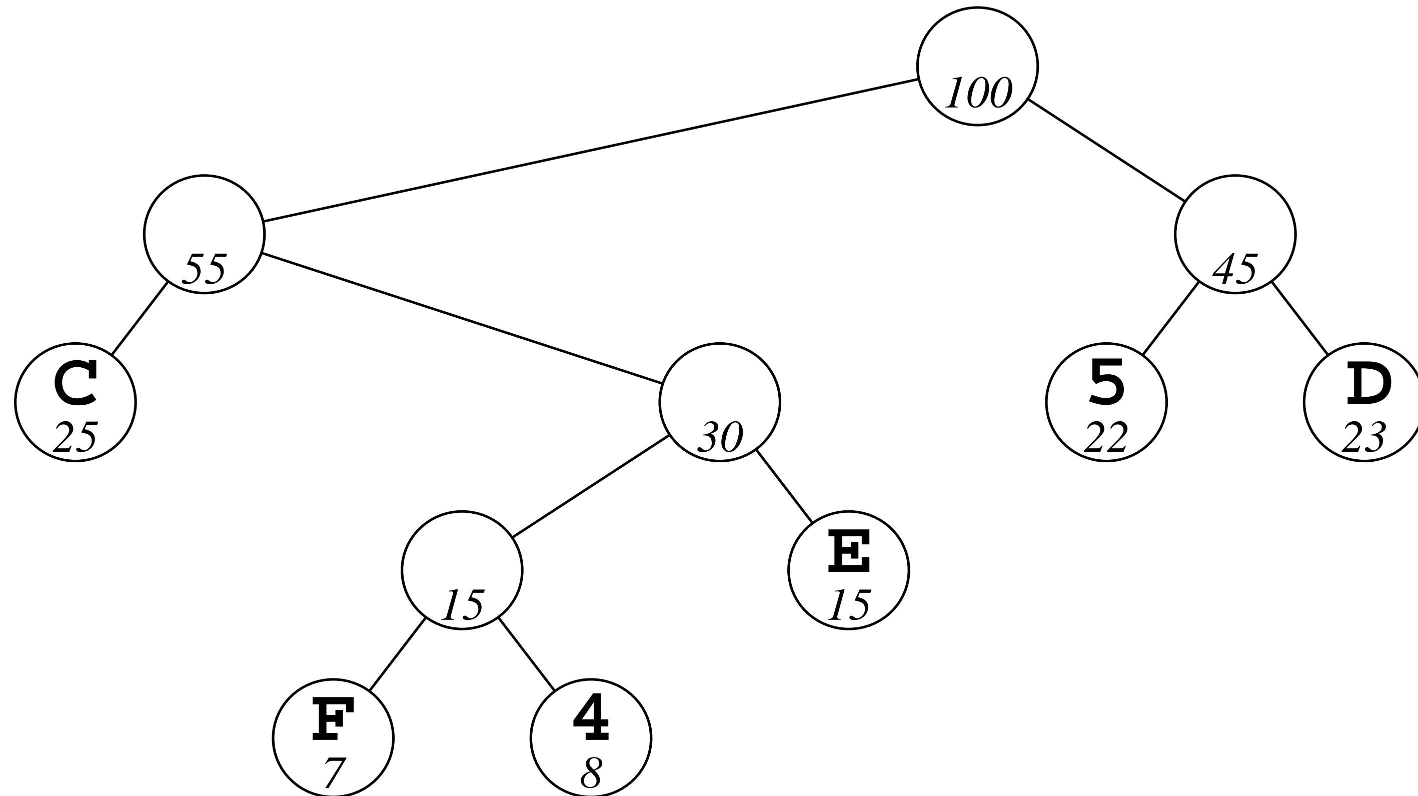


Iteración 4:



El montículo binario (heap)

Iteración 5:



Coste del **árbol de codificación de Huffman** obtenido:

$$2 \cdot 25 + 4 \cdot 7 + 4 \cdot 8 + 3 \cdot 15 + 2 \cdot 22 + 2 \cdot 23 = 245$$

El montículo binario (heap)

El algoritmo de Huffman es un algoritmo voraz que, partiendo de los símbolos y sus frecuencias, va creando local y sucesivamente subárboles de mínimo coste, y finaliza obteniendo el árbol binario de codificación de mínimo coste (codificación óptima de longitud variable y libre de prefijos).

▷ La selección de los árboles del bosque con menor coste para formar otro puede hacerse con un heap.

El árbol de codificación de Huffman puede utilizarse para implementar dos programas de compresión y descompresión de ficheros.

▷ Para ello, el árbol de codificación debe incluirse en el fichero comprimido (con una representación adecuada), ya que sino será imposible descomprimir.

▷ Esta manera de comprimir y descomprimir tiene el inconveniente de que la compresión necesita dos recorridos del fichero original: uno para calcular las frecuencias y otro para codificar.