Árboles – Definiciones

- **Arbol**: Un arbol consiste en un nodo (**r**, llamado nodo **raiz**) y una lista o conjunto de subárboles ($\mathbf{A}_1$, $\mathbf{A}_2$, ... , $\mathbf{A}_k$). Si el orden de los subárboles importa, entonces se representan como una lista, y se denomina **árbol ordenado**. En caso contrario se representan como una colección y se denomina **arbol no ordenado**.

- Se definen como **nodos hijos** de **r** a los los nodos raices de los subárboles $\mathbf{A}_1$, $\mathbf{A}_2$, ... , $\mathbf{A}_k$

- Si **b** es un **nodo hijo** de **a** entonces **a** es el **nodo padre** de **b**.

- Un nodo puede tener cero o más hijos, y uno o ningún padre. Si no tiene nodo padre entonces es el **nodo raiz** del árbol.

- Un nodo sin hijos se denomina **nodo hoja**.

- Se define un **camino** en un arbol como cualquier secuencia de nodos del arbol, $\mathbf{n_1}$ ... $\mathbf{n_p}$, que cumpla que cada nodo es padre del siguiente en la secuencia (es decir, que $\mathbf{n_i}$ es el padre de $\mathbf{n_{i+1}}$). La **longitud** del camino se define como el número de nodos de la secuencia menos uno (**p-1**).

- Los **descendientes** de un nodo son aquellos nodos accesibles por un camino que comienze en el nodo. Los **ascendientes** de un nodo son los nodos del camino que va desde la raiz a él.

- La **altura de un nodo** en un arbol se define como la longitud del camino más largo que comienza en el nodo y termina en una hoja. La altura de un nodo hoja será de cero, y la altura de un nodo se puede calcular sumando uno a la mayor altura de sus hijos.

- La **altura de un árbol** se define como la altura de su raiz.

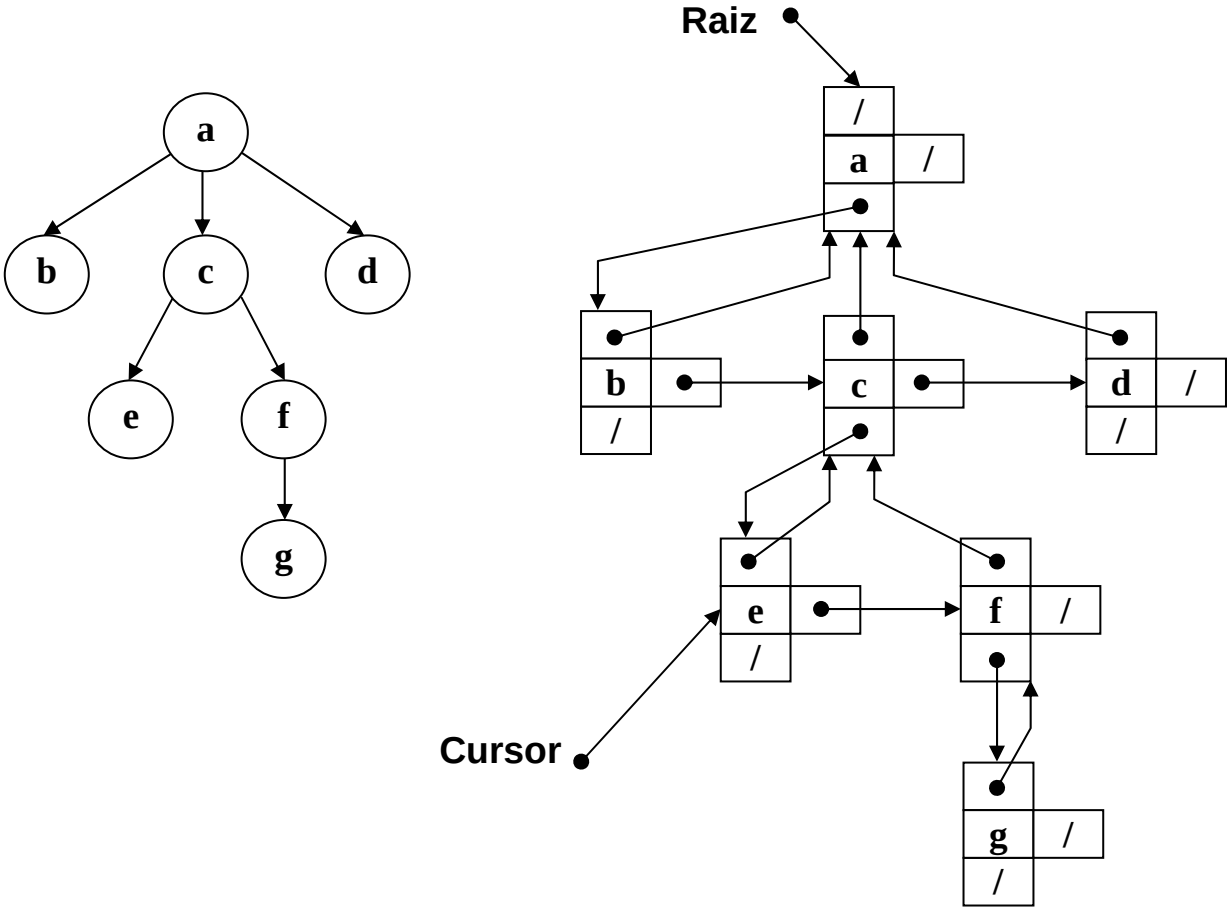
- La **profundidad de un nodo** se define como la longitud del camino (único) que comienza en la raiz y termina en el nodo. La profundidad de la raiz es cero, y la profundidad de un nodo se puede calcular como la profundidad de su padre mas uno. A la profundidad de un nodo también se la denomina **nivel** del nodo en el árbol.

Representación de Árboles

- Salvo casos particulares, un árbol se almacena mediante nodos con referencias al nodo padre y a la lista de nodos hijos, o bien con referencias al nodo padre, al primer hijo y al nodo hermano. Se suele utilizar un acceso basado en cursor.

- Las operaciones principales son el acceso al nodo raiz, la inserción y borrado de subárboles hijos del nodo actual y el cambio del cursor (del nodo actual a su padre y del nodo actual a uno de sus hijos).

- Ejemplo (representación padre-primer hijo-hermano):



Variantes de Árboles

- **Árbol binario**: Árbol que consta de un nodo raiz y de dos subárboles, llamados **subárbol izquierdo** y **subárbol derecho**. Se permite que existan árboles vacíos (sin ningún nodo, ni siquiera el raiz). Los árboles vacíos tienen altura –1.

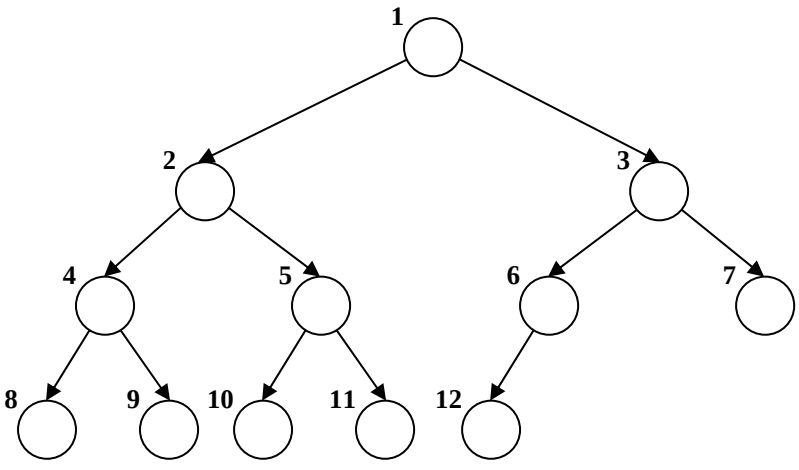
- Cada nodo de un árbol binario puede tener ningún hijo (subárbol izquierdo y derecho vacíos), un hijo (subarbol izquierdo **o** derecho vacío) o dos hijos. Dependiendo de si son la raiz del subarbol izquierdo o derecho se denominan **hijo izquierdo** e **hijo derecho**.

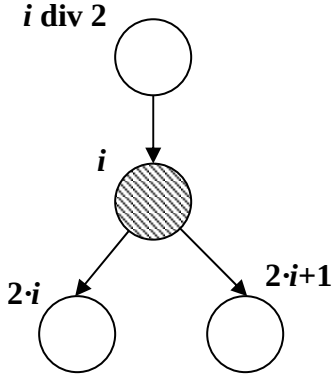
- **Árbol binario estricto**: No se permite que un subárbol esté vacío y el otro no lo esté. Por lo tanto cada nodo puede tener cero o dos hijos.

- **Árbol binario perfectamente equilibrado (árbol lleno)**: La altura del subarbol izquierdo es igual a la altura del subarbol derecho y además ambos subárboles también están perfectamente equilibrados. Un arbol perfectamente equilibrado tiene 2^{h+1}-1 nodos (*h* es la altura del árbol).

- **Árbol binario completo**: Un árbol perfectamente equilibrado hasta el penúltimo nivel, y en el último nivel los nodos se encuentran agrupados a la izquierda.

  En un árbol completo se pueden indexar los nodos mediante un recorrido por niveles, y a partir de ese índice es posible conocer el índice del nodo padre y los índices de los nodos hijos. Esta propiedad permite almacenar un árbol completo en un vector sin necesidad de información adicional (referencia al nodo padre y a los nodos hijos), simplemente almacenando cada nodo en la posición del vector que indica el recorrido por niveles.





- **Montículo**: Un árbol binario completo que almacena elementos con campo clave y donde los nodos cumplen la **propiedad de montículo**: Todo nodo del arbol almacena un elemento cuya clave es **menor** que las claves de sus **descendientes** en el arbol.

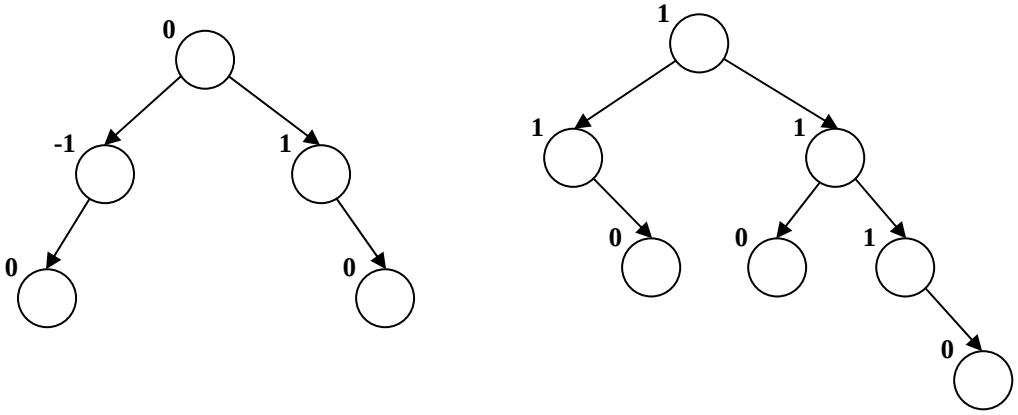
  La definición anterior es de un montículo cuya raiz es el elemento mínimo. Alternativamente, podemos definir un montículo cuya raiz sea el elemento máximo con sólo cambiar la palabra *menor* por *mayor*.

  Si n es el número de elementos del montículo y h su altura se cumple:

  $$n \in \{2^h ... 2^{h+1}-1\}, \quad h = \lfloor \lg n \rfloor, \quad \text{Nivel del nodo } i\text{-ésimo} = \lfloor \lg i \rfloor$$

- **Árbol binario de búsqueda**: Un árbol binario que almacena elementos con campo clave y donde los nodos cumplen la **propiedad de ordenación**: Todo nodo del arbol almacena un elemento cuya clave es **mayor** (o igual) que las claves de los nodos de su **subárbol izquierdo**, y **menor** (o igual) que las claves de los nodos de su **subárbol derecho**.

- **Arbol binario equilibrado**: Un arbol binario en el que la altura del subárbol izquierdo y la del subárbol derecho o son iguales o se diferencian en una unidad, y además ambos subárboles son equilibrados. Se define **factor de equilibrio** de cada nodo como el resultado de restar la altura del subarbol izquierdo a la altura del subarbol derecho. Sólo puede tomar los valores –1, 0 y +1 para un árbol binario equilibrado. Ejemplos:



- **Arbol AVL**: Un arbol binario de búsqueda equilibrado. Comparten las características de los árboles binarios de búsqueda pero el orden de las operaciones de acceso (búsqueda), inserción y borrado es estricto (no es un caso promedio).

Montículos: Operaciones auxiliares

Elevación de un nodo:

```
type
   { El montículo se representa por un vector que almacena sus elementos (registros con campo
     clave) en el orden de un recorrido por niveles. El vector tiene una capacidad máxima de Max
     elementos, y en un momento dado almacena únicamente Num elementos en los índices 1..N }
   TMonticulo = record
      Vec : array[1..MAX] of TElemento;
      Num : integer
   end
```

```
procedure Elevar(var M: TMonticulo; I: Integer);
{ Reorganiza un montículo en el que, al cambiar de valor el nodo I-ésimo, es posible que ya no
  cumpla la propiedad de monticulo para los ascendientes de ese nodo. El algoritmo consiste en
  intercambiar el  nodo con sus ascendientes hasta restablecer la propiedad. Eficiencia: O(lg n) }
var
   Padre, Hijo : integer;
   Seguir : boolean;
begin
   Hijo := I ; Seguir := true ;
   while (Hijo > 1) and Seguir do
   begin
      Padre := Hijo div 2 ;
      if M.Vec[Padre].Clave > M.Vec[Hijo].Clave then
      begin
         { No se cumple la propiedad de montículo: Se intercambia el nodo padre con el nodo hijo y se
           sigue comprobando los ascendientes }
         M.Vec[Padre] ⇔ M.Vec[Hijo] ;
         Hijo := Padre
      end else begin
         Seguir := false
      end { if }
   end { while }
end; { Elevar }
```

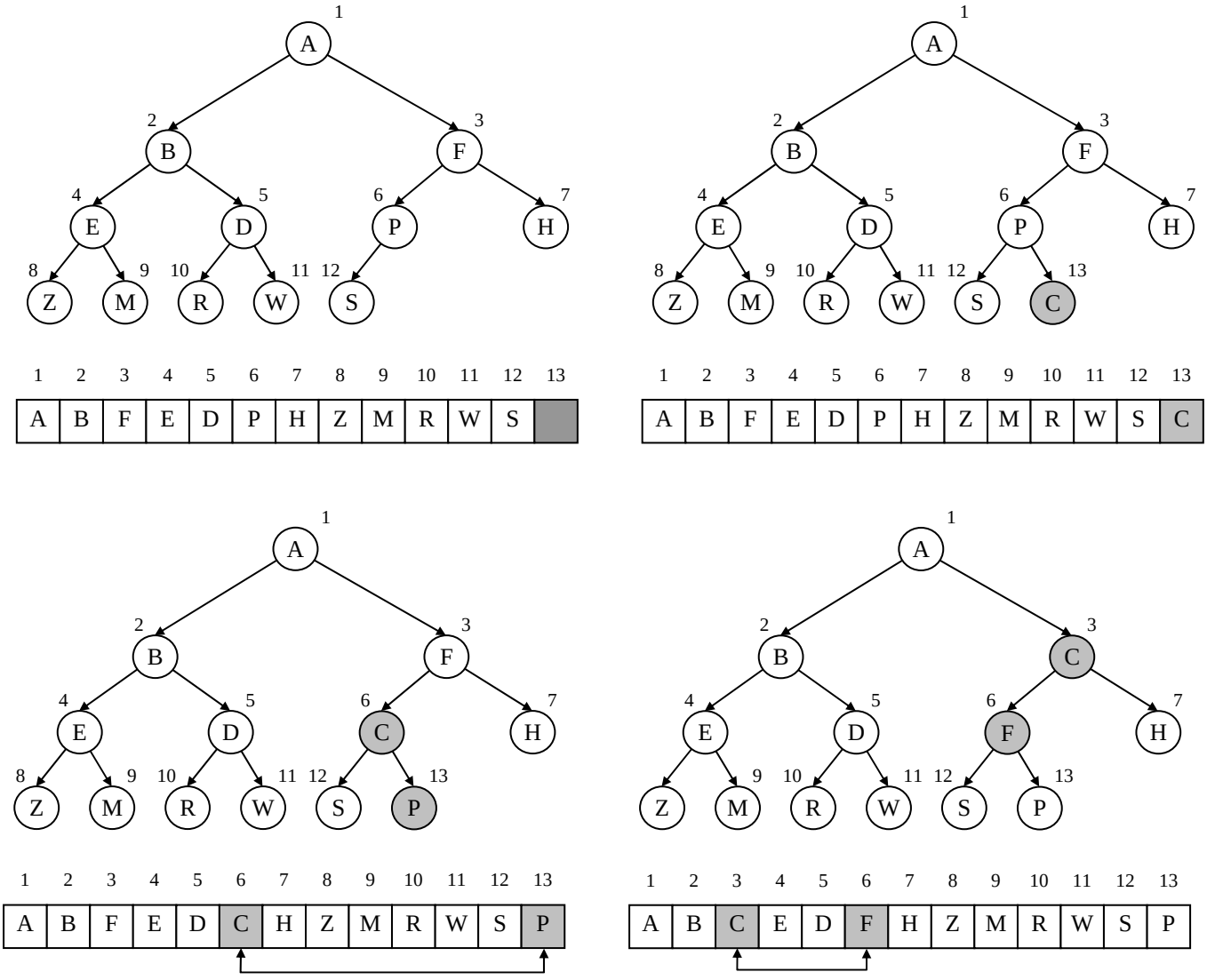
Reestructuración de un (sub)montículo:

```
procedure Reestructurar(var M: TMonticulo; I: integer) ;
{ Reorganiza un montículo en el que, al cambiar de valor el nodo I-ésimo, es posible que ya no se
  cumpla la propiedad de monticulo para los descendientes de ese nodo. Alternativamente, esta
  operación se puede contemplar como reorganizar un (sub)montículo cuya raiz es el nodo I, donde
  todos los nodos excepto la raiz cumplen la propiedad de montículo. El algoritmo consiste en
  intercambiar el nodo con sus descendientes hasta restablecer la propiedad. Eficiencia: O(lg n) }
var
   Padre, Hijo : integer;
   Seguir : boolean;
begin
   Padre := I;
   Hijo := 2*Padre ;   { hijo izquierdo }
   Seguir := Cierto
   while (Hijo ≤ M.Num) and Seguir do
   begin
      { Comprobar cual es el hijo con clave menor }
      if Hijo < M.Num then   { existe hijo derecho }
         if M.Vec[Hijo+1].Clave < M.Vec[Hijo].Clave then  { hijo derecho es el menor }
            Hijo := Hijo+1;
      { Comprobar si el padre tiene una clave mayor que la del hijo menor }
      if M.Vec[Padre].Clave > M.Vec[Hijo].Clave then
      begin
         { No se cumple la propiedad de montículo: Se Intercambia el nodo padre con el nodo hijo y
           se sigue comprobando los descendientes }
         M.Vec[Padre] ⇔ M.Vec[Hijo] ;
         Padre := Hijo ;
         Hijo := 2*Padre
      end else begin
         Seguir := false
      end  { if }
   end  { while  }
end;  { reestructurar }
```

Operaciones sobre montículos

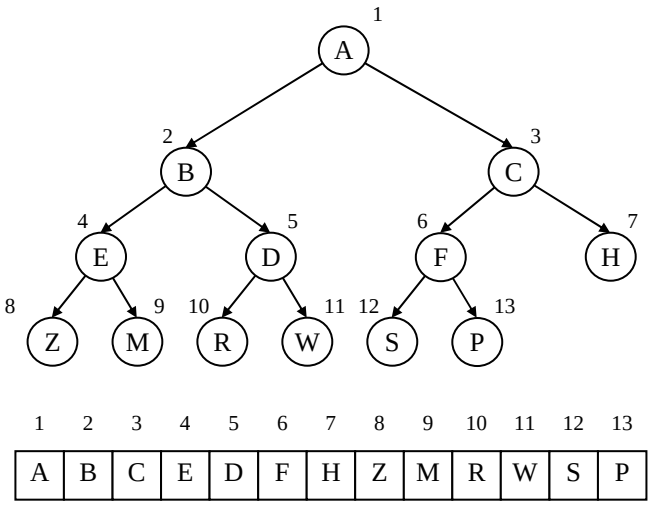
Inserción de un elemento:

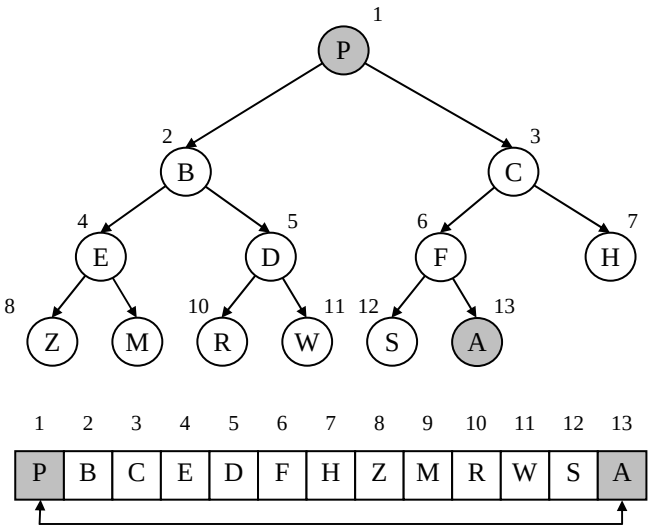
```
procedure Insertar(var M: TMonticulo ; E: TElemento) ;
{ Inserta el elemento E en el montículo M de manera que se siga manteniendo la estructura de
  montículo. Eficiencia: O(lg n) }
begin
   M.Num := M.Num+1 ;
   if M.Num > MAX then { ampliar capacidad de M.Vec } ;
   M.Vec[M.Num] := E;  { Inserción al final del vector, como el último nodo hoja }
   Elevar(M, M.Num)      { Reorganizar el montículo elevando lo necesario el nodo insertado }
end; { Insertar }
```

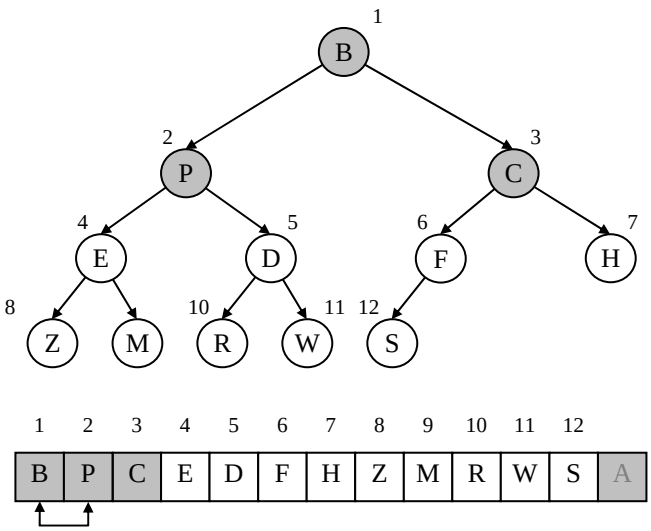


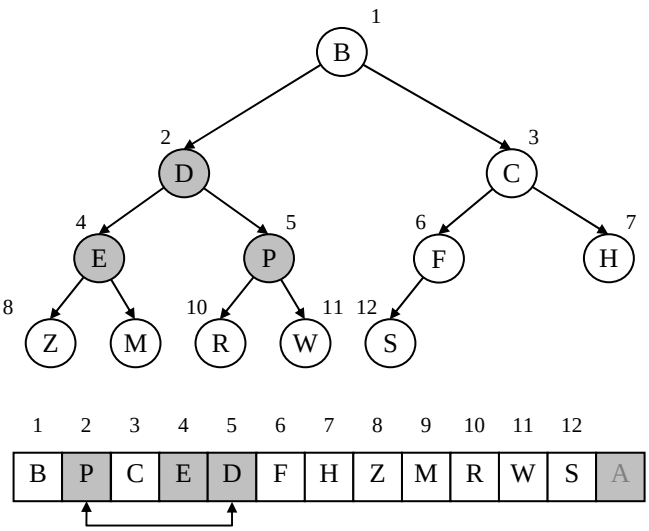
Borrado del nodo raiz (elemento con clave mínima):

```
procedure Borrar(var M: TMonticulo) ;
{ Precondición: M.Num > 1 }
begin
   M.Vec[1] ⇔ M.Vec[M.Num] ;   { Se intercambia el raiz con el último }
   M.Num := M.Num-1 ;          { Se borra el último elemento (el que era antes el raiz) }
   Reestructurar(M, 1)         { Se reestructura todo el montículo (se hace descender la raiz) }
end; { Borrar }
```









Modificación de un nodo:

```
procedure Modificar(var M: TMonticulo; I: integer; E: TElemento) ;
{ Cambia el valor del elemento I-ésimo del montículo por E }
begin
   if E.Clave < M.Vec[I].Clave then { Se cambia un elemento por otro menor }
      { Sólo puede afectar a los ascendientes del nodo que cambia }
      M.Vec[I] := E ;
      Elevar(M, I)                { Elevar el nodo lo necesario }
   end else begin                 { Se cambia un elemento por otro mayor }
      { Sólo puede afectar a los descendientes del nodo que cambia }
      M.Vec[I] := E ;
      Reestructurar(M, I)         { Descender el nodo lo necesario }
   end
end; { Modificar }
```

Creación de un montículo a partir de un vector desordenado:

```
procedure Crear(var M: TMonticulo) ;
{ En el array M.Vec[1..M.Num] se almacenan elementos desordenados. Este procedimiento
  reorganiza el array para que pase a tener estructura de montículo. Eficiencia: O(n) }
var I : integer;
begin
   { Realiza una secuencia de reestructuraciones desde los niveles inferiores (comenzando por el
     padre del último nodo) hasta la raiz. }
   for I := M.Num div 2 downto 1 do
      Reestructurar(M, I)
end; { Crear }
```

Implementación de Colas de prioridad

	Lista no ordenada*		Lista ordenada**	Montículo
Acceder al mínimo	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Insertar	$O(1)$	$O(1)$	$O(n)$	$O(\lg n)$
Borrar el mínimo	$O(n)$	$O(n)$	$O(1)$	$O(\lg n)$
Modificar	$O(1)$	$O(n)$	$O(n)$	$O(\lg n)$

*La columna derecha representa la variante de almacenar una referencia al elemento minimo y actualizarla adecuadamente en el resto de operaciones. Esto supone que el acceso es $O(1)$ y modificar pasa a ser $O(n)$.

**Se supone que está ordenada de mayor a menor.

Ordenación por montículos:

```
procedure OrdMonticulos(var M: TMonticulo);
{ Ordena el vector M.Vec[1..M.Num] de mayor a menor. Primero reorganiza el vector para dotarle
  de estructura de montículo, y despues extrae sus mínimos sucesivos (que se van depositando al
  final del vector). Cuando el montículo queda vacío, la zona del vector M.Vec[1..M.Num] contiene los
  elementos originales ordenados de mayor a menor. Si se desea el orden habitual de menor a mayor
  se debe trabajar con un montículo de máximos en lugar de mínimos (basta con cambiar la
  comparación entre padres e hijos de menor a mayor en los subprogramas Elevar y Reestructurar }
var I, N : integer;
begin
   N := M.Num;   { Tamaño original }
   Crear(M);     { Reorganiza el vector como montículo }
   { Extracción de los elementos máximos, que se depositan en orden al final del vector, fuera de la
     parte que representa el montículo. El último elemento no es necesario extraerlo. }
   for I := 1 to N-1 do  Borrar(M);
   M.Num := N    { Se restablece el tamaño original }
end; { OrdMonticulos }
```

Comparación de algoritmos avanzados de ordenación:

	Eficiencia		**Ctes. de proporcionalidad**	
	Tiempo	**Espacio**	**Movimientos**	**Comparaciones**
Fusión	$O(n \log n)$	$O(n)$	2.00	0.92
Rápida	$\Omega(n \log n)^*$	$\Omega(\log n)^*$	0.75	1.35
Montículos	$O(n \log n)$	$O(1)$	1.33	1.80

* Se dan las cotas y constantes del caso promedio, el peor caso sería tiempo $O(n^2)$ y espacio $O(n)$