Pseudocódigos

1. **Pilas**

```
tipo pila = registro
   cima_de_pila  =  0..Tamaño_máximo_de_pila
   vector_pila  =  vector  [1..Tamaño_máximo_de_pila]  de  Tipo_de_Elemento
fin registro
```

```
procedimiento crearPila (P)                                ->    O(1)
   P.cima_de_pila := 0
fin procedimiento
```

```
función pilaVacía  (P)  : test                             ->    O(1)
   devolver P.cima_de_pila := 0
fin función
```

```
procedimiento apilar (x, P)                                ->    O(1)
   si  P.cima_de_pila  =  Tamaño_máximo_de_pila  entonces
         error Pila llena
   si no
         P.cima_de_pila := P.cima_de_pila + 1;
         P.  vector_pila[P.cima_de_pila]  := x
fin procedimiento
```

```
función cima  (P)  : Tipo_de_elemento                      ->    O(1)
   si pilaVacía (P) entonces error Pila Vacía
   si no devolver P.  vector_pila[P.cima_de_pila]
fin función
```

```
procedimiento Desapilar (P)                                ->    O(1)
   si pilaVacía (P) entonces error Pila Vacía
   si no P.cima_de_pila := P.cima_de_pila – 1
fin procedimiento
```

2. Colas

```
tipo cola = registro
   cabeza, final    : 1..Tamaño_maximo_cola
   tamaño           : 0.. Tamaño_maximo_cola
   vector_de_cola   : vector [1.. Tamaño_maximo_cola] de Tipo_de_Elemento
fin registro
```

```
procedimiento crearCola (C)                                  ->    O(1)
   C.tamaño := 0;
   C.cabeza := 1;
   C.final  := Tamaño_maximo_cola
fin procedimiento
```

```
función colaVacía (C) : test                                 ->    O(1)
   devolver P.tamaño := 0
fin función
```

```
procedimiento incrementar (x)       *privado*                ->    O(1)
   si x = Tamaño_maximo_cola entonces x:=1
   si no x := x + 1
fin procedimiento
```

```
procedimiento insertarEnCola (x, C)                          ->    O(1)
   si C.tamaño = Tamaño_maximo_cola entonces
        error Cola Llena
   si no
        C.tamaño = C.tamaño + 1;
        incrementar(C.final);
        C.vector_de_cola[C.final] := x;
   fin procedimiento
```

```
función quitarPrimero (C) : Tipo_de_elemento                 ->    O(1)
   si colaVacía (C) entonces
        error Cola Vacía
   si no
   C.tamaño = C.tamaño - 1;
   2.1   :=
   C.vector_de_cola[C.cabeza];
   incrementar(C.cabeza);

   devolver x
fin función
```

```
función quitarPrimero (C) : Tipo_de_elemento                 ->    O(1)
   si colaVacía (C) entonces
        error Cola Vacía
   si no devolver C.vector_de_cola[C.cabeza];
fin función
```

3. Listas con un nodo cabeceraR

```
tipo PNodo = puntero a Nodo
  Lista = PNodo
  Posición =
  PNodoNodo =
  registro
       Elemento : Tipo_de_elemento
       Siguiente : PNodo
  fin registro
```

```
procedimiento Crear Lista ( L )                         ->   O(1)
  nuevo ( tmp );
  si tmp = nil entonces error Memoria agotada
  sino
    tmp^.Elemento := { nodo cabecera };
    tmp^.Siguiente := nil;
    L := tmp
fin procedimiento
```

```
función Lista Vacía ( L ) : test                        ->  O(1)
  devolver L^.Siguiente = nil
fin función
```

```
función Buscar ( x, L ) : posición de la 1ª ocurrencia o nil  -> O(n)
  p := L^.Siguiente;
  mientras p <> nil y p^.Elemento <> x hacer
       p := p^.Siguiente;
  devolver p
fin función
```

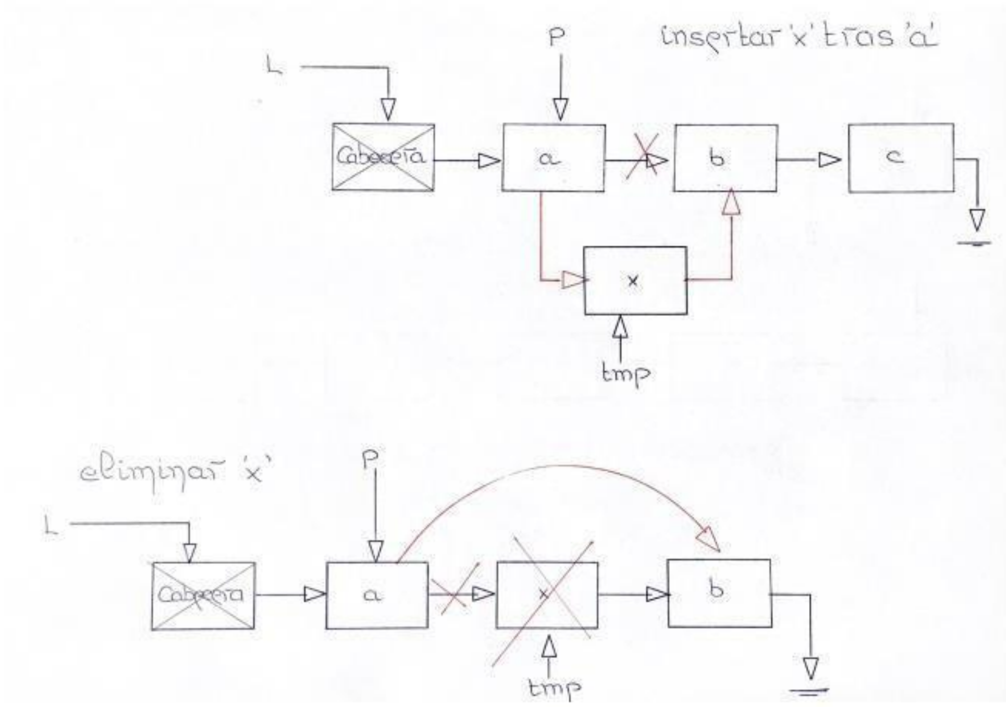
```
función Último Elemento ( p ) : test *privada*          ->  O(1)
  devolver p^.Siguiente = nil
fin función
```

```
función Buscar Anterior ( x, L ) : pos anterior a x o nil *privada*O(n)
  p := L;
  mientras p^.Siguiente <> nil y p^.Siguiente^.Elemento <> x hacer
       p := p^.Siguiente;

  devolver p
fin función
```

```
procedimiento Eliminar ( x, L )                    ->          O(n)
   p := Buscar Anterior ( x, L );

   si Último Elemento ( p ) entonces
         error No encontrado
   sino
         tmp := p^.Siguiente;
         p^.Siguiente := tmp^.Siguiente;
         liberar ( tmp )
fin procedimiento
```

```
procedimiento Insertar ( x, L, p )                 ->          O(1)
   nuevo ( tmp ); { Inserta después de la posición p }
   si tmp = nil entonces error Memoria agotada
   sino
      tmp^.Elemento := x;
      tmp^.Siguiente := p^.Siguiente;
      p^.Siguiente := tmp
fin procedimiento
```

- **Insertar y eliminar:**



4. Árboles Binarios de Búsqueda

```
tipo
   PNodo = ^Nodo
   Nodo = registro
         Elemento : TipoElemento
         Izquierdo, Derecho : PNodo
   fin registro

   ABB = PNodo
```

```
función Buscar(x, A): PNodo                    {c.medio:O(log n) c.peor:O(n)}
   si A = nil entonces
         devolver nil
   sino si x = A^.Elemento entonces
         devolver A
   sino si x < A^.Elemento entonces
         devolver Buscar (x, A^.Izquierdo)
   sino
         devolver Buscar (x, A^.Derecho)
fin función
```

```
función BuscarMin(A): PNodo                    {c.medio:O(log n) c.peor:O(n)}
   si A = nil entonces
         devolver nil
   sino si A^.Izquierdo = nil entonces
         devolver A
   sino
         devolver BuscarMin (A^.Izquierdo)
   fin función
```

```
procedimiento Insertar(x, var A)               {c.medio:O(log n) c.peor:O(n)}
   si A = nil entonces
         nuevo (A);

   si A = nil entonces error ''sin memoria''

   sino
         A^.Elemento := x;
         A^.Izquierdo := nil;
         A^.Derecho := nil

    sino si x < A^.Elemento entonces
          Insertar (x, A^.Izquierdo)

   sino si x > A^.Elemento entonces { si x = A^.Elemento : nada }
         Insertar (x, A^.Derecho)

   fin procedimiento
```

```
procedimiento Eliminar(x, var A)      {c.medio:O(log n)} {c.peor:O(n)}
    si A = nil entonces
        error ''no encontrado''

    sino si x < A^.Elemento entonces
        Eliminar (x, A^.Izquierdo)

    sino si x > A^.Elemento entonces
        Eliminar (x, A^.Derecho)

    sino { x = A^.Elemento }
        si A^.Izquierdo = nil entonces
            tmp := A;
            A := A^.Derecho;
            liberar (tmp)

        sino si A^.Derecho = nil entonces
            tmp := A;
            A := A^.Izquierdo;
            liberar (tmp)

        sino
            tmp := BuscarMin (A^.Derecho);
            A^.Elemento := tmp^.Elemento;
            Eliminar (A^.Elemento, A^.Derecho)
fin procedimiento
```

Eliminar: borra la clave x.

- Si x esta en una hoja, se elimina de inmediato.
- Si el nodo tiene un hijo, se ajusta un apuntador antes de eliminarlo.
- Si el nodo tiene dos hijos, se sustituye x por la clave más pequeña, w, del subárbol derecho.
  A continuación, se elimina en el subárbol derecho el nodo con w (que no tiene hijo izquierdo)

4.1 Recorridos en árboles

- **En orden**: Se procesa el subárbol izquierdo, el nodo actual y, por último,el subárbol derecho. O(n)

```
procedimiento Visualizar (A)
   si A <> nil entonces
      Visualizar (A^.Izquierdo);
      Escribir (A^.Elemento);
      Visualizar (A^.Derecho)
fin procedimiento
```

- **Post-orden**: Ambos subárboles primero. O(n)

```
función Altura (A) : número
   si A = nil entonces devolver -1
   sino devolver 1 + max (Altura (A^.Izquierdo), Altura (A^.Derecho))
fin función
```

- **Pre-orden**: El nodo se procesa antes. Ej: una funcion que ´ marcase cada nodo con su profundidad. O(n)

- **Orden de nivel**: Todos los nodos con profundidad p se procesan antes que cualquier nodo con profundidad p +1. Se usa una cola en vez de la pila implícita en la recursión. O(n)

```
procedimiento OrdenDeNivel (A)
   CrearCola(C);

   si A <> nil entonces
      InsertarEnCola(A, C);

      mientras no ColaVacía(C) hacer
         p:= QuitarPrimero(C);
         escribir(p^.Elemento); {Operación principal}

         si p^.Izq <> nil entonces InsertarEnCola(p^.Izq, C);
         si p^.Der <> nil entonces InsertarEnCola(p^.Der, C);

      fin mientras

fin procedimiento
```

5. **Montículos**

- Un **montículo** es un árbol binario completo : todos los niveles están llenos con la posible excepción del nivel más bajo, que se llena de izquierda a derecha.
- Un árbol binario completo de altura h tiene **entre 2h y 2h+1 –1 nodos.** Su **altura** es la **parte entera de log2 n.** Esta regularidad facilita su representación mediante un vector:
  Para cualquier elemento en la posición i del vector, el **hijo izquierdo está en la posición 2i, el hijo derecho en 2i +1, y el padre en i ÷ 2.**

```
tipo Montículo = registro
   Tamaño_montículo: 0..Tamaño máximo
   Vector_montículo: vector [1..Tamaño_máximo] de Tipo_elemento
fin registro
```

```
procedimiento Inicializar Montículo ( M )
   M.Tamaño_montículo := 0
fin procedimiento
```

```
función Montículo Vacío ( M ) : test
   return M.Tamaño_monticulo = 0
fin función
```

```
procedimiento Flotar ( M, i ) { privado }
   mientras i>1 y M.Vector_montículo[i/2]<M.Vector_montículo[i] hacer

         intercambiar M.Vector_montículo[i/ 2] y M.Vector_montículo[i];
         i := i div 2

   fin mientras
fin procedimiento
```

```
procedimiento Insertar ( x, M )
   si M.Tamaño_monticulo = Tamaño_máximo entonces error montículo lleno
   sino
         M.Tamaño_montículo := M.Tamaño_montículo + 1;
         M.Vector_montículo[M.Tamaño_montículo] := x;
         Flotar ( M, M.Tamaño_montículo )
fin procedimiento
```

```
procedimiento Hundir ( M, i ) { privado }
   repetir
      HijoIzq := 2*i;
      HijoDer := 2*i+1;
      j := i;

      si HijoDer <= M.Tamaño_montículo y M.Vector_montículo[HijoDer] >
      M.Vector_montículo[i] entonces
         i := HijoDer;

      si HijoIzq <= M.Tamaño_montículo y M.Vector_montículo[HijoIzq] >
      M.Vector_montículo[i] entonces
         i := HijoIzq;
         intercambiar M.Vector_montículo[j] y M.Vector_montículo[i];

   hasta j=i {Si j=i el nodo alcanzó su posición final}
fin procedimiento



función EliminarMax ( M ) : Tipo elemento
   si Montículo Vacío ( M ) entonces error montículo vacío

   sino
         x := M.Vector_montículo[1];
         M.Vector_montículo[1] := M.Vector montículo[M.Tamaño_montículo];
         M.Tamaño_montículo := M.Tamaño_montículo - 1;

         si M.Tamaño_montículo > 0 entonces
               Hundir ( M, 1);
         devolver x
fin función
```

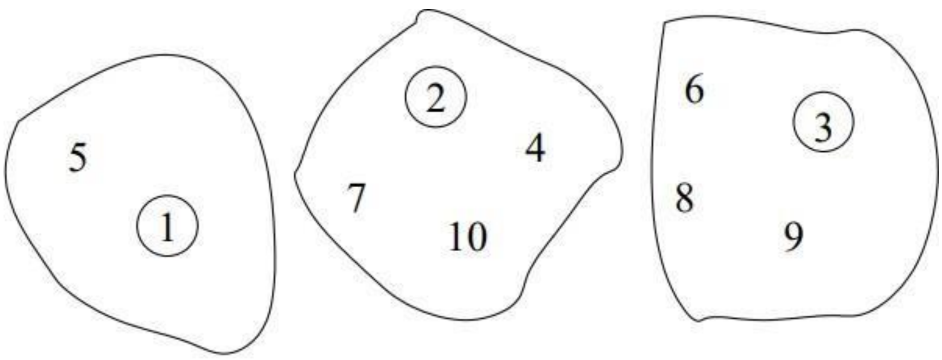
- **<u>Creación de montículos en tiempo lineal, O(n)</u>**:

```
   procedimiento Crear_Montículo ( V[1..n], M )
         Copiar V en M.Vector_montículo;
         M.Tamaño_montículo := n;
         para i := M.Tamaño_montículo div 2 hasta 1 paso -1
               Hundir(M, i);
         fin para
   fin procedimiento
```

6. <u>Conjuntos disjuntos</u>

6.1 Primer enfoque

- Todos los elementos se numeran de 1 a n.
- Cada subconjunto tomará su nombre de uno de sus elementos, su **representante**, p. ej. el valor más pequeño.
- Mantenemos en un vector el nombre del subconjunto disjunto de cada elemento.



	1	2	3	4	5	6	7	8	9	10
C	1	2	3	2	1	3	2	3	3	2

```
tipo
   Elemento = entero;
   Conj = entero;
   ConjDisj = vector [1..N] de entero
```

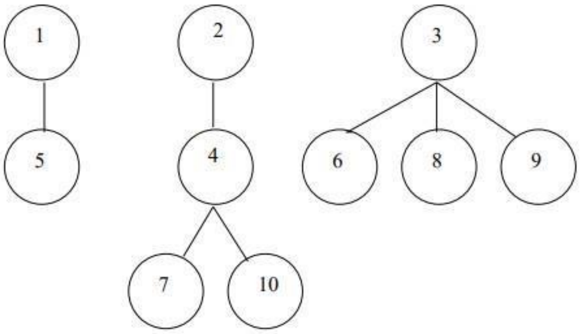
```
función Buscar1 (C, x) : Conj
   devolver C[x]
fin función
```

```
procedimiento Unir1 (C, a, b)
   i := min (C[a], C[b]);
   j := max (C[a], C[b]);
   para k := 1 hasta N hacer
         si C[k] = j entonces C[k] := i
   fin para
fin procedimiento
```

- **La combinación de m búsquedas y n-1 uniones toma $O(m + n^2)$**

6.2 Segundo enfoque

- o Se utiliza un árbol para caracterizar cada subconjunto.
- o La raíz nombra al subconjunto.
- o La representación de los árboles es fácil porque la única información necesaria es un apuntador al padre.
- o Cada entrada p[i] en el vector contiene el padre del elemento i.
- o Si i es una raíz, entonces p[i]=i.



	1	2	3	4	5	6	7	8	9	10
C	1	2	3	2	1	3	4	3	3	4

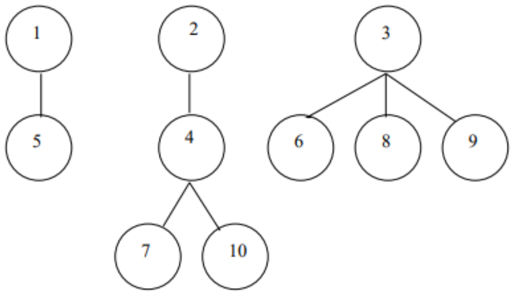
```
función Buscar2 (C, x) : Conjr := x;
     mientras C[r] <> r hacer
          r := C[r]
     fin mientras;
     devolver r
fin función
```

```
procedimiento Unir2 (C, raíz1, raíz2) {supone que raíz1 y raíz2 son raíces}
     si raíz1 < raíz2 entonces
          C[raíz2] := raíz1
     sino C[raíz1] := raiz2
fin procedimiento
```

- **La unión toma O(1). La combinación de m búsquedas y n-1 uniones toma O(m · n)**

6.3 Unión por alturas

- Las uniones anteriores se efectuaban de modo arbitrario. Una mejora sencilla es realizar las uniones haciendo del árbol menos profundo un subárbol del árbol más profundo. La altura se incrementa solo cuando se unen dos árboles de igual altura.



	1	2	3	4	5	6	7	8	9	10
C	1	2	3	2	1	3	4	3	3	4

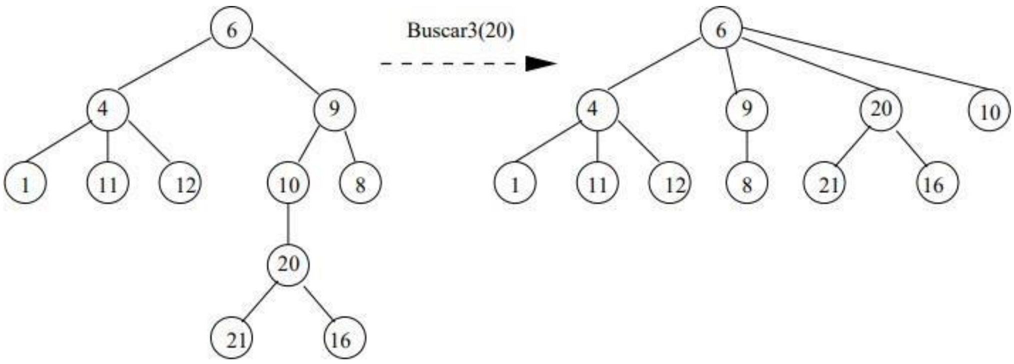
	1	2	3	4	5	6	7	8	9	10
A	1	2	1							

```
procedimiento Unir3 (C, A, raíz1, raíz2) {supone que raíz1 y raíz2
son raíces}
      si A[raíz1] = A[raíz2] entonces
            A[raíz1] := A[raíz1] + 1;
            C[raíz2] := raíz1
      sino si A[raíz1] > A[raíz2] entonces
            C[raíz2] := raíz1
      sino C[raíz1] := raíz2
fin procedimiento
```

- **El tiempo de ejecución de una búsqueda es O(log(n)). Combinando m búsquedas 6.4 n-1 uniones, $O(m \cdot \log(n) + n)$**

6.5 Compresión de caminos

- La comprensión de caminos se ejecuta durante búsqueda. Durante la búsqueda de un dato x, todo nodo en el camino de x a la raíz cambia su padre por la raíz. Es independiente del modo en que se efectúen las uniones.




```
función Buscar3 (C, x) : Conj
   r := x;
   mientras C[r] <> r hacer
      r := C[r]
   fin mientras;

   i := x;
   mientras i <> r hacer
         j := C[i];
         C[i]:= r;
         i := j
   fin mientras;
   devolver r
fin función
```

7. Tablas de dispersión

7.1 Dispersión abierta

- La solución consiste en tener una lista de todos los elementos que se dispersan en un mismo valor. Al buscar, usamos la función de dispersión para determinar que lista recorrer. Al insertar, recorremos la lista adecuada. Si el elemento resulta ser nuevo, se inserta al frente o al final de la lista. Además de listas, se podría usar cualquier otra estructura para resolver las colisiones, como un árbol binario de búsqueda.

- El **factor de carga, λ**, de una tabla de dispersión es la relación entre el número de elementos en la tabla y su tamaño.

$$\lambda = \frac{n^{\underline{o}}\ de\ claves\ en\ la\ tabla}{N}$$

```
tipo
      Índice = 0..N-1
      Posición = ^Nodo
      Lista = Posición
      Nodo = registro
            Elemento : TipoElemento
            Siguiente : Posición
      fin registro

      TablaDispersión = vector [Índice] de Lista


procedimiento InicializarTabla (T)
      para i := 0 hasta N-1 hacer
            CrearLista(T[i])
      fin para
fin procedimiento



función Buscar (Elem, Tabla): Posición
      i := Dispersión(Elem);
      devolver BuscarLista(Elem, Tabla[i])
fin función



procedimiento Insertar (Elem, Tabla)
      pos := Buscar(Elem, Tabla); {No inserta repetidos}
      si pos = nil entonces
            i := Dispersión(Elem);
            InsertarLista(Elem, Tabla[i])
fin procedimiento
```

7.2 Dispersión cerrada

- En un sistema de dispersión cerrada, si ocurre una colisión, se buscan celdas alternativas hasta encontrar una vacía. Se busca en sucesión en las celdas: d0(x),d1(x),d2(x)... donde:

  $$\boxed{d_i(x) = (dispersion(x) + f(i)) \bmod N, \text{ con } f(0) = 0}$$

  La función f es la **estrategia de resolución de las colisiones**.

```
tipo
    ClaseDeEntrada = (legítima, vacía, eliminada)
    Índice = 0..N-1
    Posición = Índice
    Entrada = registro
        Elemento : TipoElemento
        Información : ClaseDeEntrada
    fin registro

    TablaDispersión = vector [Índice] de Entrada
```

```
procedimiento InicializarTabla (D)
    para i := 0 hasta N-1 hacer
        D[i].Información := vacía
    fin para
fin procedimiento
```

```
función Buscar (Elem, D): Posición
    i := 0;
    x = Dispersión(Elem);
    PosActual = x;

    Mientras D[PosActual].Información <> vacía y
    D[PosActual].Elemento <> Elem hacer
        i := i + 1;
        PosActual := (x + FunResoluciónColisión(x, i)) mod N
    fin mientras;
devolver PosActual
fin función
```

/*La búsqueda finaliza al caer en una celda vacía o al encontrar elelemento (legítimo o borrado)*/

```
procedimiento Insertar (Elem, D)
      pos = Buscar(Elem, D);

      si D[pos].Información <> legítima entonces {Bueno para insertar}
            D[pos].Elemento := Elem;
            D[pos].Informaci´on := legítima
fin procedimiento
```

```
procedimiento Eliminar (Elem, D)
      pos = Buscar(Elem, D);

      si D[pos].Información = legítima entonces
            D[pos].Información := eliminada
fin procedimiento
```

- **Dispersión cerrada con exploración lineal (agrupamiento primario):** $f(i) = i$
- **Dispersión cerrada con exploración cuadrática (agrupamiento secundario):** $f(i) = i^2$
- **Dispersión cerrada con exploración doble:** aplicamos una segunda función de dispersión, en general: $f(i) = i \cdot h_2(x)$

8. Algoritmos de ordenación

Ordenación por inserción

```
procedimiento Ordenación por Inserción (var T[1..n])
   para i:=2 hasta n hacer
      x:=T[i];
      j:=i-1;
      mientras j>0 y T[j]>x hacer
         T[j+1]:=T[j];
         j:=j-1
      fin mientras;
      T[j+1]:=x
   fin para
fin procedimiento
```

Ordenación de Shell

- **Secuencia de incrementos ≡** distancias para intercambios: Naturales, ordenados descendentemente: $h_t, ... h_k, h_{k-1}, ... h_1 = 1$

- **t iteraciones:** en la iteración k utiliza el incremento h_k

  Postcondición = $\{\forall i, T[i] \leq T[i + h_k]\}$

  ≡ los elementos separados por h_k posiciones están ordenados
  → vector h_k -ordenado

- **Trabajo de la iteración k:** h_k ordenaciones por Inserción

- **Propiedad:** un vector h_k-ordenado que se h_{k-1}-ordenado sigue estando h_k-ordenado

```
procedimiento Ordenación de Shell (var T[1..n])
      incremento := n;
      repetir
            incremento := incremento div 2;

            para i := incremento+1 hasta n hacer
                  tmp := T[i];
                  j := i;
                  seguir := cierto;

            mientras j-incremento > 0 y seguir hacer
                  si tmp < T[j-incremento] entonces
                        T[j] := T[j-incremento];
                        j := j-incremento
                  sino seguir := falso ;
                  T[j] := tmp
      hasta incremento = 1
fin procedimiento
```

Ordenación por montículos

```
procedimiento Ordenación por Montículos (var T[1..n])
      Crear montículo (T, M);
      para i := 1 hasta n hacer
            T[n-i+1] := Obtener mayor valor (M);
            Eliminar mayor valor(M)
      fin  para
fin procedimiento
```

- **Para crear un montículo a partir de un vector:**

```
procedimiento Crear montículo (V[1..n], var M)
{V[1..n]: entrada: vector con cuyos datos se construirá el montículo
M: entrada/salida: montículo a crear}

      Copiar V[1..n] en M[1..n];

      para i := n div 2 hasta 1 paso -1 hacer
            hundir(M,i)
      fin  para
fin procedimiento
```

Ordenación por fusión

```
procedimiento Fusión ( var T[Izda..Dcha], Centro:Izda..Dcha )

{fusiona los subvectores ordenados T[Izda..Centro] y T[Centro+1..Dcha] en
T[Izda..Dcha], en T[Izda..Dcha], utilizando un vector auxiliar Aux[Izda..Dcha]}

     i := Izda ;
     j := Centro+1 ;
     k := Izda ;

     {i, j y k recorren T[Izda..Centro], T[Centro+1..Dcha] y Aux[Izda..Dcha]
     respectivamente}

     mientras i <= Centro y j <= Dcha hacer
          si T[i] <= T[j] entonces
               Aux[k] := T[i] ;
               i := i+1
          sino Aux[k] := T[j] ;
          j := j+1 ;
          k := k+1 ;

     {copia elementos restantes del subvector sin recorrer}

     mientras i <= Centro hacer
          Aux[k] := T[i] ;
          i := i+1 ;
          k := k+1 ;

     mientras j <= Dcha hacer
          Aux[k] := T[j] ;
          j := j+1 ;
          k := k+1 ;

     para k := Izda hasta Dcha hacer
          T[k] := Aux[k]
fin procedimiento
```

```
procedimiento Ordenación por Fusión Recursivo ( var T[Izda..Dcha] )
     si Izda+UMBRAL < Dcha entonces
          Centro := ( Izda+Dcha ) div 2 ;
          Ordenación por Fusión Recursivo ( T[Izda..Centro] ) ;
          Ordenación por Fusión Recursivo ( T[Centro+1..Dcha] ) ;

          Fusión ( T[Izda..Dcha], Centro )
     sino Ordenación por Inserción ( T[Izda..Dcha] )
fin procedimiento
```

```
procedimiento Ordenación por Fusión ( var T[1..n] )
     Ordenación por Fusión Recursivo ( T[1..n] );
fin procedimiento
```

Ordenación Rápida

```
procedimiento Mediana 3 ( var T[i..j] )
      centro := ( i+j ) div 2 ;

      si T[i] > T[centro] entonces intercambiar (T[i], T[centro]);
      si T[i] > T[j] entonces intercambiar (T[i], T[j]);
      si T[centro] > T[j] entonces intercambiar (T[centro], T[j]);

      intercambiar (T[centro], T[j-1])
fin procedimiento


procedimiento Qsort ( var T[i..j] )
      si i+UMBRAL <= j entonces Mediana 3 ( T[i..j] ) ;

      pivote := T[j-1] ;
      k := i ;
      m := j-1 ; {sólo con Mediana 3}

      repetir
            repetir k := k+1 hasta T[k] >= pivote ;
            repetir m := m-1 hasta T[m] <= pivote ;
                  intercambiar ( T[k], T[m] )
      hasta m <= k ;

      intercambiar ( T[k], T[m] ) ; {deshace el último intercambio}
      intercambiar ( T[k], T[j-1] ) ; {pivote en posición k}

      Qsort ( T[i..k-1] ) ;
      Qsort ( T[k+1..j] )
fin procedimiento



procedimiento Quicksort ( var T[1..n] )
      Qsort ( T[1..n] ) ;
      Ordenación por Inserción ( T[1..n] )
fin procedimiento
```