

PILAS

tipo Pila = registro

```
Cima_de_pila : 0..Tamaño_máximo_de_pila
Vector_de_pila : vector [1..Tamaño_máximo_de_pila]
de Tipo_de_elemento
```

fin registro

procedimiento Crear Pila (P) O(1)

```
P.Cima_de_pila := 0
fin procedimiento
```

función Pila Vacía (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
sino
  P.Cima_de_pila := P.Cima_de_pila + 1;
  P.Vector_de_pila[P.Cima_de_pila] := x
fin procedimiento
```

función Cima (P) : Tipo_de_elemento O(1)

```
si Pila Vacía (P) entonces error Pila vacía
sino devolver P.Vector_de_pila[P.Cima_de_pila]
fin función
```

procedimiento Desapilar (P) O(1)

```
si Pila Vacía (P) entonces error Pila vacía
sino P.Cima_de_pila := P.Cima_de_pila - 1
fin procedimiento
```

LISTAS

tipo PNode = puntero a Nodo

```
Lista = PNode
Posición = PNode
Nodo = registro
  Elemento : Tipo_de_elemento
  Siguiente : PNode
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(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
```

COLAS

tipo Cola = registro

```
Cabeza_de cola, Final_de cola: 1..Tamaño_máximo_de cola
Tamaño_de cola : 0..Tamaño_máximo_de cola
Vector_de cola : vector [1..Tamaño_máximo_de cola]
de Tipo_de_elemento
```

fin registro procedimiento Crear_Cola (C) O(1)

```
C.Tamaño_de cola := 0;
C.Cabeza_de cola := 1;
C.Final_de cola := Tamaño_máximo_de cola
```

fin procedimiento

función Cola_Vacía (C) : test O(1)

```
devolver C.Tamaño_de cola = 0
fin función
```

procedimiento incrementar (x) (* privado *) O(1)

```
si x = Tamaño_máximo_de cola entonces x := 1
sino x := x + 1
fin procedimiento
```

procedimiento Insertar_en_Cola (x, C) O(1)

```
si C.Tamaño_de_Cola = Tamaño_máximo_de cola entonces
  error Cola llena
sino
  C.Tamaño_de cola := C.Tamaño_de cola + 1;
  incrementar(C.Final_de cola);
  C.Vector_de cola[C.Final_de cola] := x;
fin procedimiento
```

función Quitar_Primer (C) : Tipo_de_elemento O(1)

```
si Cola_Vacía ( C ) entonces
  error Cola vacía
sino
  C.Tamaño_de cola := C.Tamaño_de cola - 1;
  x := C.Vector_de cola[C.Cabeza_de cola];
  incrementar(C.Cabeza_de cola);
  devolver x
fin función
```

función Primero (C) : Tipo_de_elemento O(1)

```
si Cola_Vacía ( C ) entonces
  error Cola vacía
sino
  devolver C.Vector_de cola[C.Cabeza_de cola]
fin función
```

función Buscar Anterior (x, L) : posición anterior a x 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
```

ABB

tipo

```
PNode = "Nodo"
Nodo = registro
  Elemento : TipoElemento
  Izquierdo, Derecho : PNode
fin registro
ABB = PNode
```

procedimiento CrearABB (var A) O(1)

```
A := nil
fin procedimiento
```

función Buscar(x, A): PNode {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): PNode {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
  Insertar (x, A^.Derecho)
  { si x = A^.Elemento : nada }
fin procedimiento
```

procedimiento Eliminar(x, var A) {c.medio:O(log n)}

```
si A = nil entonces error "no encontrado" {c.peor:O(n)}
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
```

- 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
```

- 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); {operación principal}
  si p^.Izq <> nil entonces InsertarEnCola(p^.Izq, C);
  si p^.Der <> nil entonces InsertarEnCola(p^.Der, C);
fin mientras
fin procedimiento
```

evichu_u



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

MONTÍCULOS

```
tipo Monticulo = registro
  Tamaño_monticulo : 0..Tamaño_máximo
  Vector_monticulo : vector [1..Tamaño_máximo]
  de Tipo.elemento

fin registro

procedimiento InicializarMonticulo ( M )
  M.Tamaño_monticulo := 0
fin procedimiento

función Monticulo.Vacio ( M ) : test
  return M.Tamaño_monticulo = 0
fin función

procedimiento Flotar ( M, i ) { privado }
  mientras i > 1 y M.Vector_monticulo[i div 2] < M.Vector_monticulo[i]
  hacer intercambiar M.Vector_monticulo[i div 2] y M.Vector_monticulo[i];
  i := i div 2
fin mientras
fin procedimiento

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

procedimiento Hundir ( M, i ) { privado }
  repetir
    HijoIzq := 2*i;
    HijoDer := 2*i+1;
    j := i;
    si HijoDer <= M.Tamaño_monticulo y
      M.Vector_monticulo[HijoDer] > M.Vector_monticulo[i]
    entonces i := HijoDer;
    si HijoIzq <= M.Tamaño_monticulo y
      M.Vector_monticulo[HijoIzq] > M.Vector_monticulo[i]
    entonces i := HijoIzq;
    intercambiar M.Vector_monticulo[j] y M.Vector_monticulo[i];
  hasta j=i {si j=i el nodo alcanzó su posición final}
fin procedimiento

función EliminarMax ( M ) : Tipo.elemento
  si Monticulo.Vacio ( M ) entonces
    error Monticulo vacío
  sino
    x := M.Vector_monticulo[1];
    M.Vector_monticulo[1] := M.Vector_monticulo[M.Tamaño_monticulo];
    M.Tamaño_monticulo := M.Tamaño_monticulo - 1;
    si M.Tamaño_monticulo > 0 entonces
      Hundir ( M, 1 );
    devolver x
fin función
```

- Creación de montículos en tiempo lineal, $O(n)$:
procedimiento Crear.Monticulo (V[1..n], M)
Copiar V en M.Vector_monticulo;
M.Tamaño_monticulo := n;
para i := M.Tamaño_monticulo div 2 hasta 1 paso -1
Hundir (M, i);
fin para
fin procedimiento
- El número de intercambios está acotado por la suma de las alturas de los nodos.
- Se demuestra mediante un argumento de marcado del árbol.
 - Para cada nodo con altura h, marcamos h aristas:
 - bajamos por la arista izquierda y después sólo por aristas derechas.
 - Así una arista nunca se marca 2 veces.

TABLA DE DISPERSIÓN ABIERTA

```
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);
  si pos = nil entonces
    i := Dispersión(Elem);
    InsertarLista(Elem, Tabla[i])
  fin procedimiento
```

TABLA DE DISPERSIÓN CERRADA

```
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 el elemento (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].Información := legítima
  fin procedimiento

procedimiento Eliminar (Elem, D)
  pos := Buscar(Elem, D);
  si D[pos].Información = legítima
  entonces
    D[pos].Información := eliminada
  fin procedimiento
```

CONJUNTOS DISJUNTOS

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

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

- La búsqueda es una simple consulta $O(1)$.
 - El nombre del conjunto devuelto por búsqueda es arbitrario.
 - Todo lo que importa es que $búsqueda(x)=búsqueda(y)$ si y sólo si x e y están en el mismo conjunto.

```
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 unión toma $O(n)$. No importa, en lo que concierne a corrección, qué conjunto retiene su nombre.
- Una secuencia de $n-1$ uniones (la máxima, ya que entonces todo estaría en un conjunto) tomaría $O(n^2)$.
- La combinación de m búsquedas y $n-1$ uniones toma $O(m+n^2)$.

segundo enfoque

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

- Una búsqueda sobre el elemento x se efectúa devolviendo la raíz del árbol que contiene x.
- La búsqueda de un elemento x es proporcional a la profundidad del nodo con x.
 - En el peor caso es $O(n)$

```
procedimiento Unir2 (C, raíz1, raíz2)
  si raíz1 < raíz2 entonces C[raíz2] := raíz1
  sino C[raíz1] := raíz2
fin procedimiento
```

- La unión de dos conjuntos se efectúa combinando ambos árboles: apuntamos la raíz de un árbol a la del otro.
- La unión toma $O(1)$.
- La combinación de m búsquedas y $n-1$ uniones toma $O(m+n)$.

unión por altura

```
procedimiento Unir3 (C, A, raíz1, raíz2)
  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
```

- La profundidad de cualquier nodo nunca es mayor que $\log_2(n)$.
 - Todo nodo está inicialmente a la profundidad 0.
 - Cuando su profundidad se incrementa como resultado de una unión, se coloca en un árbol al menos el doble de grande.
 - Así, su profundidad se puede incrementar a lo más $\log_2(n)$ veces.
- El tiempo de ejecución de una búsqueda es $O(\log(n))$.
- Combinando m búsquedas y $n-1$ uniones, $O(m \cdot \log(n) + n)$.
- La compresión 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
```

ALGORITMOS de ORDENACIÓN

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

```

mejor: $O(n)$

↳ No insertar nunca, ya que el vector está ordenado

medio: $O(n^2)$ peor: $O(n^2)$

↳ insertar siempre en la 1ra posición

procedimiento Ordenación por Selección (var T[1..n])

```

para i:=1 hasta n-1 hacer
  minj:=i;
  minx:=T[i];
  para j:=i+1 hasta n hacer
    si T[j]<minx entonces
      minj:=j;
      minx:=T[j]
  fin si
  fin para;
  T[minj]:=T[i];
  T[i]:=minx
fin para
fin procedimiento

```

mejor: $O(n^2)$ medio: $O(n^2)$ peor: $O(n^2)$

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

```

mejor: O medio: O peor: O

ORDENACIÓN RÁPIDA

procedimiento Qsort (var T[i..j])

```

si i+UMBRALE <= 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

```

mejor: $O(n \log n)$ medio: $O(n \log n)$ peor: $O(n^2)$

```

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

```

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

```

i := Izda;
j := Centro;
k := Izda;
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];
    i := i+1;
    k := k+1;
  fin si
  k := k+1;
  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

```