biblia 21/22.pdf

evichu_u

Algoritmos

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

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 PNodo = puntero a Nodo
     Lista = PNodo
     Posición = PNodo
     Nodo = 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
     tmpˆ.Elemento := { nodo cabecera };
     tmpˆ.Siguiente := nil;
     L := tmp
fin procedimiento

función Lista Vacía ( L ) : test O(1)
   devolver Lˆ.Siguiente = nil
fin función

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

función Último Elemento ( p ) : test { privada } O(1)
   devolver pˆ.Siguiente = nil
fin función
```

evichu_u

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_Primero ( 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 a nil { privada } O(n)
   p := L;
   mientras pˆ.Siguiente <> nil y
                    pˆ.Siguienteˆ.Elemento <> x hacer
     p := pˆ.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 := pˆ.Siguiente;
        pˆ.Siguiente := tmpˆ.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
     tmpˆ.Elemento := x;
     tmpˆ.Siguiente := pˆ.Siguiente:
     pˆ.Siguiente := tmp
fin procedimiento
```

ABB

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

procedimiento CrearABB(var A) O(1)
  A := nil
fin procedimiento

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

función BuscarMin(A): PNodo {c.medio:O(log n) c.peor:O(n)}
  si A = nil entonces devolver nil
  sino si Aˆ.Izquierdo = nil entonces devolver A
  sino devolver BuscarMin (Aˆ.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
      Aˆ.Elemento := x;
      Aˆ.Izquierdo := nil;
      Aˆ.Derecho := nil
  sino si x < Aˆ.Elemento entonces
          Insertar (x, Aˆ.Izquierdo)
  sino si x > Aˆ.Elemento entonces
          Insertar (x, Aˆ.Derecho)
  { si x = Aˆ.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 < Aˆ.Elemento entonces
          Eliminar (x, Aˆ.Izquierdo)
  sino si  x > Aˆ.Elemento entonces
          Eliminar (x, Aˆ.Derecho)
  sino    { x = Aˆ.Elemento }
    si Aˆ.Izquierdo = nil entonces
      tmp := A; A := Aˆ.Derecho; liberar (tmp)
    sino si Aˆ.Derecho = nil entonces
      tmp := A; A := Aˆ.Izquierdo; liberar (tmp)
    sino tmp := BuscarMin (Aˆ.Derecho);
         Aˆ.Elemento := tmpˆ.Elemento;
         Eliminar (Aˆ.Elemento, Aˆ.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 (Aˆ.Izquierdo);
    Escribir (Aˆ.Elemento);
    Visualizar (Aˆ.Derecho)
fin procedimiento
```

procedimiento Preorden (A)
si A <> nil entonces
Escribir (Aˆ.Elemento);
Preorden (Aˆ.Izquierdo);
Preorden (Aˆ.Derecho);

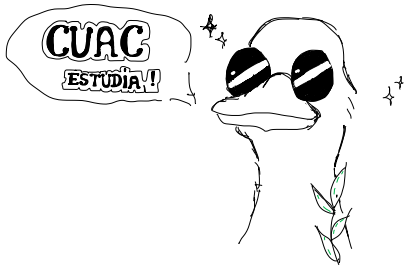
- Post-orden: Ambos subárboles primero. O(n)

```
función Altura (A) : número
  si A = nil entonces devolver -1
  sino devolver 1 + max (Altura (Aˆ.Izquierdo),
                         Altura (Aˆ.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);
    escribir(pˆ.Elemento); {Operación principal}
    si pˆ.Izq <> nil entonces InsertarEnCola(pˆ.Izq, C);
    si pˆ.Der <> nil entonces InsertarEnCola(pˆ.Der, C);
  fin mientras
fin procedimiento
```

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



evichu_u

WUOLAH

MONTÍCULOS

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

procedimiento Inicializar_Montículo ( M )
  M.Tamaño_monticulo := 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 div 2] < M.Vector_montículo[i]
  hacer intercambiar M.Vector_montículo[i div 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 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_montículo[HijoDer] > M.Vector_montículo[i]
    entonces i := HijoDer;
    si HijoIzq <= M.Tamaño_monticulo 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 Monticulo vacío
  sino
    x := M.Vector_montículo[1];
    M.Vector_montículo[1] :=
        M.Vector_montículo[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_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
```

- 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.

evichu_u

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);{No inserta repetidos}
  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
```

lo dan en el examen

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

lo dan en el examen

- 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 $O(m \cdot n)$.

evichu_u

unión por altura

```
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
```

- 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(n²)

peor : O(n²) → 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(n²)
medio: O(n²)
peor: O(n²)

```
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+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
```

mejor: O (nlogn)
medio: O (nlogn)
peor: O(n²)

```
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 ;
    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
```