Tema 7

Estructuras de datos no lineales

Tecnología de la Programación

Eficiencia

1. Introducción
2. Árboles
    1. Definiciones
    2. Árboles binarios
    3. Enriquecimiento de árboles binarios
    4. Árboles de búsqueda
    5. Montículos

Introducción

- ED lineal – cada elemento tiene como mucho un siguiente
- ED NO lineal – cada elemento puede tener varios siguientes
- ED no lineales
  - Árboles
  - Tablas
  - Grafos

Árboles

- Un **árbol** es un conjunto de elementos llamados nodos con una relación (ser antecesor o ser padre de; ser descendiente o ser hijo de) que impone una estructura jerárquica.
- Se denomina **raíz** al único nodo del árbol que no tiene antecesor.
- **Aplicaciones**: estructura adecuada para representar conjuntos entre cuyos elementos hay establecida una relación jerárquica (matemáticamente, un orden parcial):
  - Representación de fórmulas y expresiones matemáticas
  - Analizadores sintácticos
  - Transformación de programas recursivos en iterativos

Árboles

Definición

- **Base**
  - Un árbol puede tener 0 nodos, caso en el que se denomina árbol nulo.
  - Un nodo es un árbol (raíz).
- **Recursivo**

  Si d es un dato de tipo T y A_1, ..., A_k son árboles con raíces d_1, ..., d_k entonces se puede construir un árbol haciendo que d sea el nodo padre de d_1, ..., d_k (enraizar). En dicho árbol d es la raíz y A_1, ..., A_k los subárboles.

Términos

- **Nodo padre** de un nodo al primer ascendiente propio.
- **Nodo antecesor** de un nodo al que se encuentra en cualquier nivel superior.
- **Nodo descendiente** de un nodo al que se encuentra en cualquier nivel inferior.
- **Nodo hoja o nodo terminal** a un nodo que no tiene descendientes.
- **Nodo interior** a un nodo no terminal.
- **Camino** a una sucesión de nodos $d_1, ..., d_k$ tal que d_i es el padre de d_{i+1}, $1 \leq i < k$.

Términos

- **Nivel** al conjunto de nodos cuyos caminos desde la raíz tienen la misma longitud.
- **Profundidad o altura** de un árbol al máximo de los niveles de sus nodos. (Longitud del camino más largo.)
- **Grado de un nodo** al número de hijos del nodo.
- **Grado de un árbol** al máximo de los grados de sus nodos.
- Un árbol se dice **n-ario** si cada nodo es, como máximo, de grado n.
- Un árbol **binario** es un árbol de grado 2.

Árboles binarios

Un **arbol binario** es un árbol tal que:

- o es el conjunto vacío, en cuyo caso se denomina árbol vacío,
- o existe un elemento distinguido llamado raíz y el resto de elementos se distribuyen en dos subconjuntos disjuntos, cada uno de los cuales es un árbol binario, llamados subárboles izquierdo y derecho del árbol.

Árboles binarios

```
accion iniciarArbol( sal  A : arbolBin )
{ Inicia A como un árbol vacío }
accion enraizar( sal  A : arbolBin, E/S Aiz : arbolBin,
 E/S Ade : arbolBin, ent  d : telemento )
{Construye el árbol A en cuya raíz se sitúa el dato d
 del cual penden los árboles Aiz y Ade. El acceso a los
 subárboles de A mediante Aiz y Ade queda anulado}
funcion izquierdo( A : arbolBin ) devuelve arbolBin
{Devuelve el árbol izquierdo que pende del nodo raíz
  de A }
funcion derecho( A : arbolBin ) devuelve arbolBin
{ Devuelve el árbol derecho que pende del nodo raíz
  de A }
funcion raiz( A : arbolBin ) devuelve telemento
{ Devuelve el dato situado como nodo raíz del árbol A }
funcion arbolVacio( A : arbolBin ) devuelve booleano
{Si A está vacío devuelve VERDAD y FALSO en otro caso }
```

Enriquecimiento del TAD Árbol binario

Operación de recorrido en profundidad

```
preorden      raíz ---> izquierdo ---> derecho
inorden       izquierdo ---> raíz ---> derecho
postorden     izquierdo ---> derecho ---> raíz
```

Enriquecimiento del TAD Árbol binario

```
especificación ArbolBinarioEnriquecido
usa
  arbolBin
parámetros
  géneros
      telemento
  operaciones
      acción tratamiento(ent d:telemento)
      {Cualquier operación sobre el elemento d}
operaciones
  acción preOrden(ent A: arbolBin)
  {Recorre en preorden el árbol binario A }
  acción inOrden(ent A: arbolBin)
  {Recorre en inorden el árbol binario A }
  acción postOrden(ent A: arbolBin)
  {Recorre en postorden el árbol binario A }
```

Árboles de búsqueda

Los árboles de búsqueda son un tipo particular de árboles binarios, que pueden definirse cuando el tipo de los elementos del árbol posee una relación < de orden total. Tienen la propiedad de que todos los elementos almacenados en el subárbol izquierdo son menores estrictos que el elemento raíz, que a su vez es menor estricto que todos los elementos del subárbol derecho. Además ambos subárboles son árboles de búsqueda.

Árboles de búsqueda

```
especificación ArbolBusqueda;
usa
 arbolBin
parámetros
 géneros
     telemento
operaciones
 función esIgual(d1:telemento, d2:telemento) devuelve
                                                 booleano
 {Devuelve el valor verdad si d1=d2, en caso contrario
 devuelve el valor falso }
 función esMenor(d1:telemento, d2:telemento) devuelve
                                                 booleano
 {Devuelve el valor verdad si d1<d2, en caso contrario
 devuelve el valor falso }
```

Árboles de búsqueda

```
operaciones
  acción insertar(e/s A:arbolBin; ent d:telemento)
  {Inserta en el árbol de búsqueda A el elemento d. Si
  el elemento ya está en el árbol la operación no
  produce
  ningún efecto}
  acción borrar(e/s A: arbolBin; ent d:telemento)
  {Borra del árbol de búsqueda A el elemento d}
  función está?(A: arbolBin; d:telemento) devuelve
                                                  booleano
  {Devuelve el valor verdad si el elemento d está en el
  árbol de búsqueda A y falso en caso contrario}
```

Montículos

- **Un montículo es un árbol casi completo en el que cada nodo es menor o igual que sus nodos hijos (montículo de mínimos).**

- **Otra definición equivalente**
  - **a) El árbol binario ha de ser casi completo.**
  - **b) El elemento raíz ha de ser menor o igual que el resto de los elementos del árbol. Además, los *subárboles izquierdo* y *derecho* deben ser montículos.**

Montículos

```
especificación TAD Montículo;
parámetros
 géneros
     telemento
 operaciones
     función esMenor(d1:telemento,d2:telemento)
          devuelve booleano
     {Devuelve el valor verdad si d1<d2, en
     caso contrario devuelve el valor falso }
     acción permutar(e/s d1:telemento, e/s
          d2:telemento)
     {Intercambia los valores de d1 y d2 }
```

Montículos

```
géneros
  Montículo
operaciones
  acción iniciarMontículo( Sal M:Montículo )
  { Inicia M como un montículo vacío }
  acción insertar(e/s M:Monticulo; ent d:telemento)
  {Inserta en el montículo M el elemento d }
  función mínimo(M:Monticulo) devuelve telemento
  {Devuelve el elemento más pequeño del montículo M}
  acción eliminarMínimo(e/s M:Monticulo)
  {Elimina del montículo M el elemento mínimo }
  función montículoVacío(M:Monticulo) devuelve booleano
  {Devuelve verdad si M está vacío y falso en caso
  contrario}
  función alturaMontículo(M:Monticulo) devuelve entero
  {Devuelve la altura del montículo M }
```

Montículos

```
accion HeapSort1(e/s  v:vector[1..100] de entero, ent  n:entero)
{Ordena los datos del vector v de tamaño n en orden creciente}
variables
  i : entero
  M : Montículo
principio
  iniciarMontículo(M)
  para  i ← 1  hasta n hacer
       insertar(M,v[i])
  fpara
  para  i ← 1  hasta n hacer
       v[i] ← mínimo(M)
       eliminarMínimo(M)
  fpara
fin
```

Montículos

```
accion HeapSort2( e/s  v : vector[1..100] de entero, ent
 n : entero )
{ Ordena los datos del vector v de tamaño n en orden
 decreciente }
variables
 i : entero
principio
 { Crear montículo }
 para  i ← n div 2  descendiendo hasta 1 hacer
     hundir(v, n, i)
 fpara
 para  i ← n  descendiendo hasta 2 hacer
     permutar(v[1], v[i])
     hundir(v, i-1, 1)
 fpara
fin
```

Montículos

```
accion hundir( e/s  v : vector[1..100] de entero, ent  n : entero, ent
  i : entero )
{ Hunde el nodo i para restablecer la propiedad de montículo}
variables
  j : entero
principio
  repetir
        j ← i
        { Buscar el hijo menor del nodo j }
        Si  2*j<=n AND esMenor(v[2*j],v[i]) hacer
                i ← 2*j
        fsi
        Si  2*j<n AND esMenor(v[2*j+1],v[i]) hacer
                i ← 2*j+1
        fsi
        permutar(v[i],v[j])
  hasta que i=j
fin
```