Sistemas y Aplicaciones Informáticas

Tema 12. Organización Lógica de los Datos. Estructuras Dinámicas.

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Organización lógica de los datos.

2.1. Procesamiento de datos. Tipos de dato simples.

- Los datos que un ordenador debe procesar pueden ser captados directamente por el sistema o pueden ser introducidos por cualquier medio. Para ello debe realizarse lo siguiente:
  - Transformación de los datos a código binario.
  - Almacenamiento de los datos en la memoria.
  - Estructuración de los datos para su manejo adecuado.
- Los tipos de dato simples son aquellos que no pueden descomponerse en otros tipos de dato, y están incorporados en los lenguajes de alto nivel. Sin embargo, en numerosas ocasiones es necesario utilizar un conjunto de datos relacionados entre sí y tratarlos de forma unitaria.

2.2. Estructuras de datos internas. Estructuras dinámicas.

- Una estructura de datos es una manera de organizar un conjunto de datos simples con el objetivo de facilitar su manipulación como una sola unidad, definiendo la relación entre ellos y las operaciones que se pueden realizar sobre el conjunto. Si la estructura de datos reside en la memoria del ordenador se denomina estructura de datos interna.
- Una estructura dinámica es una estructura de datos interna formada por un número variable de elementos. A su vez pueden clasificarse en:
  - *Lineales*. Son aquellas en las que los datos están relacionados entre sí de forma secuencial. Las más importantes son las pilas, las colas y las listas enlazadas.
  - *No lineales*. Son aquellas en las que los datos pueden estar relacionados con cero o más datos. Las más importantes son los árboles y los grafos.

3. Estructuras dinámicas.

3.1. Lineales.

3.1.1. Pilas.

3.1.1.1. Concepto. Implementaciones.

- Son estructuras de datos que se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan solamente en un extremo de la estructura. El extremo donde se realizan estas operaciones se denomina habitualmente cima. Por esta razón, se dice que una pila es una estructura lineal de tipo LIFO (Last In First Out, el último que entra es el primero que sale).
- Existen dos posibles implementaciones en los lenguajes de alto nivel:
  - *Estática*. Es la mejor opción, por rapidez y sencillez, cuando se conoce el número máximo de datos que deben ser almacenados. De esta manera, la pila se representa mediante un registro con un campo formado por un vector que almacena los elementos de la pila, y un campo cima que almacena el índice del último elemento introducido en la pila. En C se expresa de la siguiente manera:

```
typedef int tPila;
typedef struct pila {
      tPila elem[MAX];
      int cima;
} Pila;
```

* *Dinámica*. Es la única opción cuando no se conoce el número máximo de datos que deben ser almacenados. De esta manera, la pila se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la pila y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existe un puntero externo que señala al primer nodo de la pila. En C se expresa de la siguiente manera:

```
typedef int tPila;
typedef struct pila {
      tPila elem;
      struct pila *sig;
} Pila;
```

3.1.1.2. Operaciones. Ejemplos en C.

– Existen una serie de operaciones necesarias para la manipulación de las pilas, que son:

* *Crear la pila*. La operación de creación inicia la pila como vacía.

```
void crearPilaEstatica(Pila *pPila) {

      pPila->cima=-1;
}

void crearPilaDinamica(Pila **pPila) {

      *pPila=NULL;
}
```

* *Añadir elementos*. Coloca un nuevo elemento en la cima de la pila.

```
void apilarEstatica(tPila *elem, Pila *pPila) {

      pPila->elem[++pPila->cima]=*elem;
}

void apilarDinamica(tPila *elem, Pila **pPila) {

      Pila *nodo=(Pila*)malloc(sizeof(Pila));
      nodo->elem=*elem;
      nodo->sig=*pPila;
      *pPila=nodo;
}
```

* *Eliminar elementos*. Devuelve el elemento situado en la cima y lo elimina de la pila.

```
tPila* desapilarEstatica(Pila *pPila) {

      tPila *elem=&(pPila->elem[pPila->cima]);
      pPila->cima--;
      return elem;
}

tPila* desapilarDinamica(Pila **pPila) {
```

```
      tPila *elem=&((*pPila)->elem);
      Pila *aux=*pPila;
      *pPila=(*pPila)->sig;
      free(aux);
      return elem;
}
```

* *Comprobar si la pila está vacía*. Para decidir si es posible eliminar elementos de la pila.

```
int pilaEstaticaVacia(Pila *pPila) {

      return pPila->cima==-1;
}

int pilaDinamicaVacia(Pila *pPila) {

      return *pPila==NULL;
}
```

* *Comprobar si la pila está llena*. Para decidir si es posible añadir elementos a la pila.

```
int pilaEstaticaLlena(Pila *pPila) {

      return pPila->elem==MAX-1;
}
```

3.1.2. Colas.

3.1.2.1. Concepto. Implementaciones.

- Son estructuras de datos que se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan en los extremos opuestos de la estructura. La inserción se produce en el extremo derecho o final de la estructura, mientras que la eliminación se realiza en el extremo izquierdo o inicio. Por esta razón, se dice que una cola es una estructura lineal de tipo FIFO (First In First Out, el primero que entra es el primero que sale).
- Existen dos posibles implementaciones en los lenguajes de alto nivel:
  * *Estática*. La representación más eficiente se realiza mediante una cola circular. De esta manera, la cola se representa mediante un registro con un campo formado por un vector que almacena los elementos de la cola, un campo inicio que almacena el índice anterior al primer elemento de la cola y un campo final que almacena el índice del último elemento. En C se expresa de la siguiente manera:

    ```
    typedef int tCola;
    typedef struct cola {
          tCola elem[MAX+1];
          int ini, fin;
    } Cola;
    ```

  * *Dinámica*. De esta manera, la cola se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la cola y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existen dos punteros externos: uno señala al nodo de inicio de la cola, y otro señala al nodo final. En C se expresa de la siguiente manera:

    ```
    typedef int tCola;
    typedef struct cola {
    ```

```
        tCola elem;
        struct cola *sig;
    } Cola;
```

3.1.2.2. Operaciones. Ejemplos en C.

- Existen una serie de operaciones necesarias para la manipulación de las colas, que son:
  - *Crear la cola*. La operación de creación inicia la cola como vacía.

```
void crearColaEstatica(Cola *pCola) {

      pCola->ini=pCola->fin=MAX+1;
}

void crearColaDinamica(Cola **pIniCola, Cola **pFinCola) {

      *pIniCola=*pFinCola=NULL;
}
```

  - *Añadir elementos*. Coloca un nuevo elemento en el final de la cola.

```
void encolarEstatica(tCola *elem, Cola *pCola) {

      if(pCola->fin==MAX+1)
            pCola->fin=0;
      else
            pCola->fin++;

      pCola->elem[pCola->fin]=*elem;
}

void encolarDinamica(tCola *elem, Cola **pFinCola) {

      Cola *nodo=(Cola*)malloc(sizeof(Cola));
      nodo->elem=*elem;
      nodo->sig=(*pFinCola)->sig;
      (*pFinCola)->sig=nodo;
      *pFinCola=nodo;
}
```

  - *Eliminar elementos*. Devuelve el elemento situado en el inicio y lo elimina de la cola.

```
tCola* desencolarEstatica(Cola *pCola) {

      if(pCola->ini==MAX+1)
            pCola->ini=0;
      else
            pCola->ini++;

      return &(pCola->elem[pCola->ini]);
}

tCola* desencolarDinamica(Cola **pIniCola) {

      tCola *elem=&((*pIniCola)->elem);
      Cola *aux=*pIniCola;
      *pIniCola =(*pIniCola)->sig;
      free(aux);
      return elem;
}
```

  - *Comprobar si la cola está vacía*. Para decidir si es posible eliminar elementos de la cola.

```
int colaEstaticaVacia(Cola *pCola) {

      return pCola->ini==pCola->fin;
}

int colaDinamicaVacia(Cola *pIniCola) {

      return pIniCola==NULL;
}
```

* *Comprobar si la cola está llena*. Para decidir si es posible añadir elementos a la cola.

```
int colaEstaticaLlena(Cola *pCola) {

      if(pCola->fin==MAX+1)
            return pCola->ini==0;
      else
            return pCola->ini==pCola->fin+1;
}
```

3.1.3. Listas enlazadas.

3.1.3.1. Concepto. Implementaciones. Tipos.

- Son estructuras de datos que se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan en cualquier punto según un determinado criterio de ordenación. Se trata de una generalización del concepto de pilas y colas.
- En pilas y colas cada elemento de la estructura de datos (estructura lógica) ocupa la posición de memoria (estructura física) contigua a la del elemento que le precede, lo cual favorece la implementación estática. En las listas enlazadas esto no es así; por tanto para evitar el movimiento de elementos cada vez que se realiza una operación de inserción o eliminación, la lista se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la lista y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existen dos punteros externos: uno señala al nodo de inicio de la lista, y otro señala a un punto de interés. En C se expresa de la siguiente manera:

```
typedef struct {
      int clave;
      int dato;
} tLista;

typedef struct lista {
      tLista elem;
      struct lista *sig;
} Lista;
```

- Otros tipos de listas enlazadas son las siguientes:
  * *Listas circulares*. Son listas enlazadas en las que el puntero del último nodo señala al nodo de inicio de la lista. Presentan la ventaja de que partiendo de cualquier elemento se puede hacer un recorrido completo de la lista.
  * *Listas doblemente enlazadas*. Son listas enlazadas en las que cada nodo apunta a su sucesor y a su predecesor. Presentan la ventaja de que se puede hacer un recorrido de la lista en ambos sentidos, pero implican una mayor reserva de memoria por cada nodo.

3.1.3.2. Operaciones. Ejemplos en C.

- Existen una serie de operaciones necesarias para la manipulación de las listas enlazadas, que son:

  * *Crear la lista*. La operación de creación inicia la lista como vacía.

```
void crearLista(Lista **pLista) {

      *pLista=NULL;
}
```

  * *Insertar elementos*. Coloca un nuevo elemento dentro de una lista ordenada.

```
void insertarLista(tLista *elem, Lista **pLista) {

      Lista *ant=*pLista, *pos=*pLista;
      Lista *nodo=(Lista*)malloc(sizeof(Lista));
      nodo->elem=*elem;

      if(*pLista==NULL) {
            nodo->sig=*pLista;
            *pLista=nodo;
      }
      else {
            while(pos!=NULL && (pos->elem).clave < elem->clave) {
                  ant=pos;
                  pos=pos->sig;
            }

            nodo->sig=pos;
            ant->sig=nodo;
      }
}
```

  * *Eliminar elementos*. Busca un elemento situado en cualquier posición y lo elimina.

```
tLista *eliminarLista(tLista *elem, Lista **pLista) {

      Lista *ant=*pLista, *pos=*pLista;
      tLista *aux;

      while(pos!=NULL && (pos->elem).clave!=elem->clave) {
            ant=pos;
            pos=pos->sig;
      }

      if(pos==NULL)
            return NULL;
      else {
            ant->sig=pos->sig;
            aux=&(pos->elem);
            free(pos);
            return aux;
      }
}
```

  * *Comprobar si la lista está vacía*. Para decidir si es posible eliminar elementos de la lista.

```
int listaVacia(Lista *pLista) {

      return *pLista==NULL;
}
```

3.2. No lineales.

3.2.1. Árboles.

3.2.1.1. Concepto. Tipos. Implementaciones.

- Un árbol es un conjunto finito de cero o más nodos, tal que existe un nodo especial, llamado nodo raíz, y donde los restantes nodos están separados en $n \geq 0$ conjuntos disjuntos, cada uno de los cuales es a su vez un árbol llamado subárbol del nodo raíz. Esto implica que cada nodo del árbol es raíz de algún subárbol contenido en el árbol principal.
- Algunos tipos de árboles son los siguientes:
  - *Árboles binarios*. Constituyen un tipo particular de árboles de gran aplicación, y se caracterizan porque cada nodo tendrá como máximo dos subárboles, llamados subárbol izquierdo y subárbol derecho. Cualquier árbol puede transformarse en un árbol binario.
  - *Árboles binarios enlazados*. Son árboles binarios en los que los enlaces nulos situados en el subárbol derecho de un nodo apuntan al sucesor de ese nodo en un determinado recorrido del árbol, mientras que los enlaces nulos en el subárbol izquierdo apuntan al predecesor del nodo en el mismo tipo de recorrido.
  - *Árboles binarios de búsqueda (ABB)*. Son árboles binarios utilizados para encontrar un determinado nodo sin ser necesario recorrer todos los nodos que le preceden:
    - Todos los nodos están identificados por una clave única.
    - Las claves de los nodos del subárbol izquierdo son menores que la clave del nodo raíz.
    - Las claves de los nodos del subárbol derecho son mayores que la clave del nodo raíz.
    - Los subárboles izquierdo y derecho son también árboles binarios de búsqueda.
  - *Montículos*. Un montículo de máximos (mínimos) es un árbol binario completo tal que, el valor de la clave de cada nodo es mayor (menor) o igual que las claves de sus nodos hijos, si los tiene. Se utilizan para el mantenimiento de las colas de prioridad.
- Un árbol binario se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con tres campos, uno de ellos almacena uno de los elementos del árbol y los otros dos son punteros que señalan al primer nodo del subárbol izquierdo y al primer nodo del subárbol derecho respectivamente. Los punteros de los nodos sin subárbol izquierdo o sin subárbol izquierdo señalan un valor nulo. En C se expresa de la siguiente manera:

```
typedef struct {
      int clave;
      int dato;
} tArbol;

typedef struct arbol {
      tArbol elem;
      struct arbol *izq, *der;
} Arbol;
```

3.2.1.2. Operaciones. Ejemplos en C.

- Existen una serie de operaciones necesarias para la manipulación de árboles binarios, que son:
  - *Recorrer el árbol*. Es el proceso que permite acceder una sola vez a cada uno de los nodos del árbol. El recorrido completo de un árbol produce un orden lineal en la información del

árbol. Adoptando el convenio de que siempre se recorre el subárbol izquierdo antes que el derecho, puede hacerse en amplitud (recorriendo todos los nodos de cada nivel empezando por el raíz) o en profundidad, de la siguiente manera:

- **Pre-orden**. Nodo, subárbol izquierdo y subárbol derecho.

```
void recorrerPreorden(Arbol *pArbol) {

      if(pArbol!=NULL) {
            procesar(&(pArbol->elem));
            recorrerPreorden(pArbol->izq);
            recorrerPreorden(pArbol->der);
      }
}
```

- **In-orden**. Subárbol izquierdo, nodo y subárbol derecho.

```
void recorrerInorden(Arbol *pArbol) {

      if(pArbol!=NULL) {
            recorrerInorden(pArbol->izq);
            procesar(&(pArbol->elem));
            recorrerInorden(pArbol->der);
      }
}
```

- **Post-orden**. Subárbol izquierdo, subárbol derecho y nodo.

```
void recorrerPostorden(Arbol *pArbol) {

      if(pArbol!=NULL) {
            recorrerPostorden(pArbol->izq);
            recorrerPostorden(pArbol->der);
            procesar(&(pArbol->elem));
      }
}
```

* *Buscar un nodo en un ABB*. Es una búsqueda binaria de manera recursiva o iterativa. En un ABB los nodos están ordenados por su clave recorriendo el árbol según el criterio in-orden.

```
tArbol *buscarNodoABB(tArbol *elem, Arbol *pArbol) {

      if(pArbol==NULL)
         return NULL;
      else
         if(elem->clave==(pArbol->elem).clave)
               return &(pArbol->elem);
         else
               if(elem->clave < (pArbol->elem).clave)
                     return buscarNodoABB(elem, pArbol->izq);
               else
                     return buscarNodoABB(elem, pArbol->der);
}
```

* *Insertar un nodo en ABB*. Siempre será un nodo hoja, y la posición se determina mediante una búsqueda binaria en función de la clave del nodo a insertar.
* *Eliminar un nodo en un ABB*. Existen tres casos posibles:
  - **Nodo sin hijos**. El nodo se elimina directamente.
  - **Nodo con un hijo**. El nodo se elimina y se sustituye por su hijo.

- **Nodo con dos hijos**. El nodo se elimina y se sustituye por el nodo hoja que tenga la clave menor de todos los nodos del subárbol derecho, o bien por el nodo hoja que tenga la clave mayor de todos los nodos del subárbol izquierdo.

3.2.2. Grafos.

3.2.2.1. Concepto. Definiciones. Clasificación.

- Un grafo es un conjunto finito de nodos unidos entre sí por arcos. Un nodo puede tener cero o más arcos, pero todo arco debe unir exactamente dos nodos. Los grafos representan conjuntos de objetos que no tienen restricción jerárquica entre ellos, son un superconjunto de los árboles.
- Existen una serie de definiciones en torno a los grafos:
  - Un camino es una secuencia de uno o más arcos que conectan dos nodos. Dos nodos son adyacentes si hay un arco que los une, es decir, si la longitud del camino es uno.
  - El grado de un nodo es el número de aristas de las que es extremo.
- Los grafos pueden clasificarse de la siguiente manera:
  - Un grafo es dirigido si los arcos que lo forman tienen una sola dirección.
  - Un grafo es ponderado si los arcos que lo forman tienen un peso.
  - Un grafo es regular si todos sus nodos tienen el mismo grado.
  - Un grafo es completo si existe un arco entre cada par de nodos.
  - Un grafo es conexo si cualquier nodo es alcanzable por algún otro.
  - Un grafo es euleriano si contiene al menos un camino que comprenda todos los arcos.

3.2.2.2. Representación. Recorrido.

- Los grafos pueden representarse mediante:
  - *Matrices de adyacencia*. Consiste en una matriz cuadrada en la que el número de filas y columnas corresponde al número de nodos en el grafo, de manera que los arcos entre los nodos se ven como relaciones entre los índices de la matriz. Un par de índices contiene el peso del arco si los nodos correspondientes son adyacentes, y cero en caso contrario.
  - *Listas de adyacencia*. Consiste en almacenar por cada nodo una lista dinámica con los nodos a los que se puede acceder desde él. No consume tanta memoria como las matrices de adyacencia, pero presenta la dificultad de obtener las relaciones inversas.
- Los grafos pueden recorrerse de dos maneras:
  - *En anchura*. Supone recorrer el grafo a partir de un nodo dado en niveles, es decir, primero los que están a una distancia de un arco del nodo de salida, después los que están a dos arcos de distancia, y así sucesivamente hasta alcanzar todos los nodos a los que se pudiese llegar desde el nodo de salida.
  - *En profundidad*. Trata de buscar los caminos que parten desde el nodo de salida hasta que ya no es posible avanzar más. Cuando ya no puede avanzarse más sobre el camino elegido, se vuelve atrás en busca de caminos alternativos, que no se estudiaron previamente.