Sistemas y Aplicaciones Informáticas

Tema 11. Organización Lógica de los Datos. Estructuras Estáticas.

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 estáticas.

- 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 estática es una estructura de datos interna formada por un número fijo de elementos. Las más importantes son los vectores y los registros.

3. Estructuras estáticas.

3.1. Vectores.

3.1.1. Concepto. Características. Definición. Ejemplo en C.

- Son estructuras que agrupan un número fijo y finito de datos del mismo tipo en posiciones contiguas de memoria bajo un nombre común.
- Características más importantes:
  - Son estructuras de una sola dimensión.
  - Cada uno de los elementos del vector se referencia mediante el nombre común y un índice, que es un número entero que determina su posición relativa en la memoria.
  - El número máximo de elementos de un vector se define en tiempo de compilación de los programas. No es posible modificarlo en tiempo de ejecución.
- La definición de un vector indica al compilador sus características, e incluye lo siguiente:
  - *Clase de almacenamiento*. Manera de almacenar el vector en la memoria.
  - *Tipo de dato*. Conjunto de valores que cada elemento del vector puede tomar, la cantidad de memoria que ocupa y las operaciones que pueden realizarse.
  - *Identificador del vector*. Nombre único que debe cumplir las normas de cada lenguaje.
  - *Número de elementos*. Cantidad de memoria que debe reservar el compilador para el vector.
- En C un vector se expresa de la siguiente manera: `int meses[12];`

3.1.2. Operaciones.

3.1.2.1. Recorrido.

- Es el proceso total o parcial de los elementos del vector. Esta operación se realiza utilizando sentencias repetitivas cuyas variables de control se utilizan como subíndices de control.
- En C se expresa de la siguiente manera:

```
for (i=0; i<elem; i++) {
      v[i]=v[i]+1;
}
```

3.1.2.2. Inserción.

- Supone la adición lógica, nunca física, de un elemento del vector. Si se produce al final del vector, debe comprobarse que el vector dispone de espacio de memoria suficiente.
- En caso de añadir elementos en el interior del vector, es necesario reorganizar el resto de elementos del vector, de tal manera que todos los elementos situados a la derecha del punto de inserción debe ser desplazados una posición hacia arriba antes de la inserción.

3.1.2.3. Borrado.

- Supone la eliminación lógica, nunca física, de un elemento del vector. Si se produce al final del vector, debe comprobarse que el vector dispone de al menos un elemento.
- En caso de eliminar elementos en el interior del vector, es necesario reorganizar el resto de elementos del vector, de tal manera que todos los elementos situados a la derecha del punto de eliminación debe ser desplazados una posición hacia abajo para evitar que queden huecos libres.

3.1.2.4. Búsqueda.

3.1.2.4.1. Definición. Algoritmos.

- Consiste en localizar la posición de un elemento con una determinada clave. La eficiencia de una búsqueda dependerá enormemente del grado de ordenación previa de la que dispongan los datos.
- Los algoritmos de búsqueda pueden ser:
  - *Búsqueda lineal*. Se aplica a datos ordenados y no ordenados. Consiste en recorrer el vector desde el primer elemento al último hasta encontrar un elemento cuya clave coincida con la buscada o hasta que se acabe el vector. En este último caso, debe indicarse la no existencia de dicho valor en el vector.
  - *Búsqueda binaria*. Se aplica a datos ordenados. Consiste en comparar en primer lugar con el componente central del vector y se decide la mitad en la que debe encontrarse el valor buscado. El proceso se repite hasta que se encuentre el valor buscado o hasta que el tamaño del intervalo de búsqueda quede anulado.

3.1.2.4.2. Ejemplos en C.

- Búsqueda lineal.

```
int busquedaLineal(int v[], int valorBuscado) {

      int i, elem=sizeof(v)/sizeof(int);

      for (i=0; v[i]!=valorBuscado && i<elem-1; i++) {
            if (v[i]==valorBuscado)
                  return i;
            else
```

```
                return -1;
        }
}
```

- Búsqueda binaria.

```
int busquedaBinaria(int v[], int valorBuscado) {

        int i, cen, izq=0, der=(sizeof(v)/sizeof(int))-1;
        cen=(izq+der)/2;

        while (v[cen]!=valorBuscado && izq<der) {
                if (v[cen]>valorBuscado)
                        der=cen-1;
                else
                        izq=cen+1;
                cen=(izq+der)/2;
        }

        if (v[cen]==valorBuscado)
                return cen;
        else
                return -1;
}
```

3.1.2.5. Ordenación interna.

3.1.2.5.1. Definición. Algoritmos.

- Consiste en organizar los elementos del vector por orden creciente o decreciente según una determinada clave. La eficiencia de un método de ordenación suele determinarse según el número de comparaciones realizadas y el número de movimientos de las mismas. Ambos son función del número de elementos que componen el vector a ordenar.
- Los algoritmos de ordenación pueden ser:
  - *Por inserción directa*. Consiste en tomar los elementos del vector desde el segundo hasta el último y con cada uno de ellos repetir el siguiente conjunto de operaciones:
    - Se saca del vector el elemento i-ésimo utilizando una variable auxiliar.
    - Desde el anterior al que estamos tratando y hasta el primero, desplazamos un lugar a la derecha todos los que sean mayores para buscar su hueco.
    - Encontrado el hueco del elemento, lo insertamos en él.
  - *Por selección directa*. Consiste en tomar los elementos del vector desde el primero hasta el penúltimo y con cada uno de ellos repetir el siguiente conjunto de operaciones:
    - Se guarda la posición del elemento i-ésimo utilizando una variable auxiliar.
    - Desde el posterior al que estamos tratando y hasta el último, guardamos en la variable auxiliar la posición del elemento menor.
    - Se intercambian los elementos i-ésimo y el elemento menor encontrado.
  - *Por intercambio*. Los elementos de la lista se intercambian hasta que estén ordenados:
    - **Intercambio directo**. Consiste en recorrer sucesivamente de izquierda a derecha el vector y realizar pasadas sucesivas desde el primer elemento hasta el penúltimo, comparando cada uno de ellos con el siguiente e intercambiándolos cuando estén descolocados. Existe una mejora de los métodos de intercambio directo en la que se

comprueba mediante un switch si el vector está totalmente ordenado después de cada pasada, terminando la ejecución en caso afirmativo.

- **Quick Sort**. Consiste en seleccionar un elemento especial llamado pivote, buscar un elemento situado a la izquierda del pivote mayor que él y un elemento situado a la derecha del pivote menor que él, e intercambiarlos. Debe repetirse el proceso hasta que las búsquedas por la izquierda y por la derecha se crucen. Después recursivamente se hace de nuevo la partición sobre cada una de las dos partes obtenidas siempre que tengan más de un elemento.

	Asignaciones			***Comparaciones***		
Método	mejor	medio	peor	mejor	medio	peor
Inserción	2 (n-1)	$\frac{1}{4}n^2$	$\frac{n}{2}(n+5)-3$	n-1	$\frac{1}{4}n^2$	$\frac{n}{2}(n+3)-2$
Selección	0	n	3n	$\frac{1}{2}(n^2-n)$		
Intercambio	0	$\frac{3}{4}n^2$	$\frac{3}{2}(n^2-n)$	$\frac{1}{2}(n^2-n)$		
Quicksort	$lg\, n$	$0.69\, n\, lg\, n$	$\frac{3}{8}n^2$	$n\, lg\, n$	$1.38\, n\, lg\, n$	$\frac{n^2}{2}$

3.1.2.5.2. Ejemplos en C.

– Ordenación por inserción directa.

```
void ordenaInsercion(int v[]) {

	int i, j, aux, elem=sizeof(v)/sizeof(int);

	for (i=1; i<elem; i++) {

		aux=v[i];
		j=i-1;

		while (v[j]>aux && j>=0) {
			v[j+1]=v[j];
			j--;
		}

		v[j+1]=aux;
	}
}
```

– Ordenación por selección directa.

```
void ordenaSeleccion(int v[]) {

	int i, j, aux, posMin, elem=sizeof(v)/sizeof(int);

	for (i=0; i<elem-1; i++) {

		posMin=i;
```

```
            for (j=i+1; j<elem; j++) {
                  if (v[j]<v[posMin]) posMin=j;
            }

            aux=v[i];
            v[i]=v[posMin];
            v[posMin]=aux;
      }
}
```

- Ordenación por intercambio directo.

```
void ordenaIntercambio(int v[]) {

      int i, j, aux, elem=sizeof(v)/sizeof(int);

      for (i=0; i<elem-1; i++) {
            for (j=elem-1; j>i; j--) {
                  if (v[j]<v[j-1]) {
                        aux=v[j];
                        v[j]=v[j-1];
                        v[j-1]=aux;
                  }
            }
      }
}
```

- Ordenación Quick Sort.

```
void ordenaQuickSort(int v[], int izq, int der) {

*/izq y der delimitan la zona de la partición/*
      int aux, i=izq, j=der;

*/Elección del pivote como el elemento central de la partición/*
      int piv=v[(izq+der)/2];

      while (i<=j) {
*/Búsqueda de elementos descolocados dentro de la partición/*
            while (v[i]<piv) i++;
            while (v[j]>piv) j--;

*/Intercambio de elementos mal ubicados/*
            if (i<j) {
                  aux=v[i];
                  v[i]=v[j];
                  v[j]=aux;
                  i++;
                  j--;
            } else
                  if (i==j) {
                        i++;
                        j--;
                  }
      }

*/Realiza la partición sólo si tienen más de un elemento/*
      if (izq<j) ordenaQuickSort(v, izq, j);
      if (i<der) ordenaQuickSort(v, i, der);
}
```

3.1.3. Cadenas. Tratamiento en C.

- Son estructuras que agrupan un número fijo y finito de caracteres en posiciones contiguas de memoria bajo un nombre común, que suelen manejarse en conjunto.
- En algunos lenguajes existen como un tipo de dato más y en otros, como C, las cadenas son vectores de caracteres que terminan con un carácter nulo '\0'. La mayoría de lenguajes implementa una serie de funciones para la manipulación de cadenas como tales.
- El tratamiento que le da C a las cadenas es el siguiente:
  - *Cadenas constantes*. Cualquier conjunto de caracteres encerrados entre comillas dobles es una cadena constante. Los caracteres que estén encerrados entre las comillas, junto con un carácter '\0', se almacenan en posiciones adyacentes de memoria. No es necesario añadir de forma manual el carácter nulo al final de las constantes de cadena.
  - *Vectores de caracteres*. Para inicializar un vector de caracteres basta con asignarle una cadena constante. Si al declarar un vector de caracteres se especifica su tamaño, hay que tomar la precaución de que el número de elementos declarados sea como mínimo superior en uno (el correspondiente al carácter nulo) a la longitud de la cadena con que se inicializa. Si el tamaño es inferior a dicho mínimo se producirá un error; si es superior, los elementos sobrantes se inicializan a '\0'.
  - *Funciones de manipulación de cadenas*:
    - `char* gets(char *cadena)`. Captura cadenas desde el teclado; la entrada termina al pulsar INTRO.
    - `int puts(char* cadena)`. Muestra cadenas por pantalla, deberemos emplearla siempre que sea posible pues así se añade menos código a nuestro programa.
    - `char* strcat(char* cad1, const char* cad2)`. Concatena una cadena sobre otra y añade al final un carácter nulo.
    - `int strcmp(const char* cad1, const char* cad2)`. Compara según el orden establecido por la tabla ASCII dos cadenas que finalizan con un carácter nulo.
    - `char* strcpy(char* cad1, const char* cad2)`. Copia el contenido de una cadena sobre otra.
    - `char* strncat(char* cad1, const char* cad2, size_t cuenta)`. Concatena un determinado número de caracteres de una cadena sobre otra y añade al final un carácter nulo.
    - `char* strncpy(char* cad1, const char* cad2, size_t cuenta)`. Copia un determinado número de caracteres de una cadena sobre otra y añade al final un carácter nulo.
    - `char* strnset(char* cad, int c, size_t cuenta)`. Establece el valor de un determinado número de caracteres de la cadena.
    - `char* strstr(const char* cad1, const char* cad2)`. Indica coincidencias entre partes de cadenas.
    - `unsigned int strlen(char* cad)`. Devuelve la longitud de una cadena sin contar el carácter nulo.

3.1.4. Matrices. Concepto. Características. Definición. Ejemplo en C.

- Son estructuras que agrupan un número fijo y finito de vectores del mismo tamaño y tipo de datos en posiciones contiguas de memoria bajo un nombre común, como un vector de vectores.
- Características más importantes:
  - Son estructuras de dos dimensiones.
  - Cada uno de los elementos de la matriz se referencia mediante el nombre común, el índice del vector contenedor (fila) y el índice del elemento dentro del vector contenido (columna).
  - El número máximo de elementos de una matriz se define en tiempo de compilación de los programas. No es posible modificarlo en tiempo de ejecución.
  - Las matrices se almacenan en memoria ordenadamente por filas.
- La definición de una matriz es igual a la de un vector, indicando además el número de elementos de los vectores que forma parte del vector principal.
- En C una matriz se expresa de la siguiente manera: `float lluvia[5][12];`
- Extendiendo este concepto es posible realizar matrices de más de dos dimensiones, aunque no se suelen utilizar más de tres dimensiones debido a su complejidad de manejo.

3.2. Registros.

3.2.1. Concepto. Características. Definición. Ejemplo en C.

- Agrupan un número fijo y finito de datos de distinto tipo y tamaño bajo un nombre común.
- Características más importantes:
  - Cada uno de los elementos del registro se denomina campo y se representa mediante un identificador propio. Los campos suelen mantener una relación lógica entre sí.
  - La declaración de un registro crea un nuevo tipo de dato que puede asignarse a una variable. Cada uno de los campos de un registro se referencia mediante el nombre de la variable a la que se ha asignado el registro, un operador (punto) y el nombre del campo.
  - Los campos de un registro pueden ser de tipo simple o de tipo estructurado estático.
  - Los registros ocupan posiciones consecutivas de memoria. Cuando se declara una variable del tipo del registro, el compilador reserva automáticamente el espacio de memoria necesario para cada uno de sus campos, manteniendo una copia de cada campo de la misma.
  - La información contenida en un registro se puede asignar a otro del mismo tipo mediante una única instrucción de asignación. No es necesario asignar valor a valor cada campo.
- La definición de un registro indica al compilador sus características, e incluye el identificador del registro y los tipos de dato e identificadores de cada uno de los campos.
- En C un registro se denomina estructura, y se expresa de la siguiente manera:

```
struct agenda {
      char nombre[30];
      char calle[40];
      char ciudad[20];
      char provincia[15];
      char codigo[5];
      char telefono[9];
      int importe;
};
```