

(Versión de fecha 18 de diciembre de 2024)

La teoría de este documento sirve para entender la técnica de los *Arrays compactados* y los *Arrays con huecos*, que son la base de la práctica 4. Este documento es el mismo que el *P12\_Pizarras\_2024\_12\_03.pdf*

## 1 BASES DE DATOS

Una base de datos puede ser una lista de elementos de cierto tipo de datos. Por ejemplo, si tenemos una estructura que permite guardar los datos de una persona, podríamos utilizar un array para guardar una lista de personas. El siguiente código muestra cómo hacerlo:

```
typedef struct {  
    int id;  
    char nombre[20];  
    char apellidos[40];  
} Persona;  
  
Persona ListaDePersonas[20];
```

El código anterior define una pequeña base de datos que podría tener hasta 20 personas. Como ves, se define un tipo de datos de estructura llamado *Persona* que guarda los datos de una persona y luego se define un array, llamado *ListaDePersonas*, en el que cada elemento es una variable del tipo *Persona*; el array permite guardar hasta 20 personas.

En las bases de datos, se llama *registro* a cada uno de los elementos que guarda. Por ejemplo, en la base de datos de personas que se ha mostrado, un registro serían los datos de una persona, o sea, cada elemento *Persona* de la *ListaDePersonas* sería un registro.

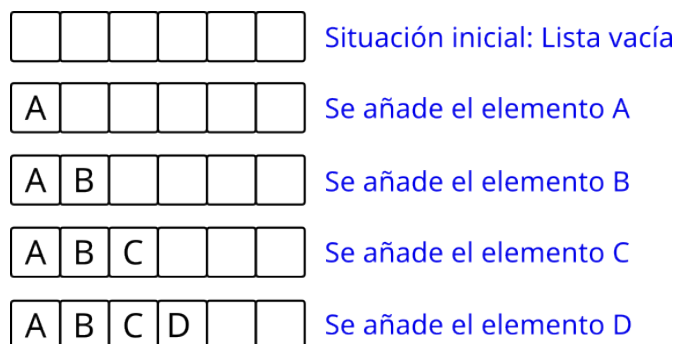
Las operaciones básicas de una base de datos se suelen denominar **CRUD**: *Create*, *Read*, *Update* y *Delete*:

- **Create**: consiste en crear un nuevo registro y añadirlo a la base de datos. En nuestro caso, sería añadir a la lista una variable del tipo *Persona*.
- **Read**: es acceder a los datos de los registros de la base de datos. En nuestro ejemplo, sería leer los datos de una o más personas.
- **Update**: se trata de modificar los datos de un registro existente en la base de datos. En la *ListaDePersonas* consistiría en modificar los datos de una persona de las que ya están en la lista.
- **Delete**: es borrar un registro de la base de datos. En la *ListaDePersonas* sería eliminar uno de los elementos de la lista.

En este documento vamos a explicar dos formas de gestionar una pequeña base de datos utilizando el lenguaje C: las **Listas compactadas** y las **Listas con huecos**. Cada una de ellas servirá para resolver una casuística concreta.

## 2 LISTAS COMPACTADAS

Supón que tenemos una lista con capacidad para 6 registros y que añadimos, de manera sucesiva, 4 registros: A, B, C y D. La situación se resume en la Figura siguiente:



En memoria tenemos reservado espacio para 6 registros. Tras añadir A, B, C y D, la lista sigue teniendo capacidad para añadir 2 elementos más. En este momento, la lista tiene 4 registros con datos válidos y otros 2 que solo contienen basura.

Si la lista está implementada con un array del lenguaje C, hay dos problemas que nuestro programa tendrá que resolver: los arrays no dan información acerca de su capacidad y tampoco dan información acerca de cuantos elementos están ocupados por datos válidos y cuántos están vacíos (ocupados por basura).

Por ejemplo, si quisiéramos añadir un elemento más a la lista del ejemplo anterior, tendríamos que saber que tenemos espacio para añadir el elemento y que el índice que le toca es el correspondiente al quinto elemento.

Una forma de gestionar esta situación es definir una estructura que tenga 2 campos: el array de los elementos y el contador de elementos que hay ya guardados en la lista. Además, el programa tiene que guardar en algún sitio la capacidad de la lista. La situación sería la de la siguiente figura:

Base de datos (struct)

Capacidad (int) 6

Lista (array)

A	B	C	D		
---	---	---	---	--	--

Contador (int)

4

La operación *CRUD Create* es sencilla de implementar: primero comprobamos si la lista está llena, comparando el campo *Contador* con la *capacidad* y, si hay sitio en la lista, añadimos el nuevo elemento en el índice que le toca.

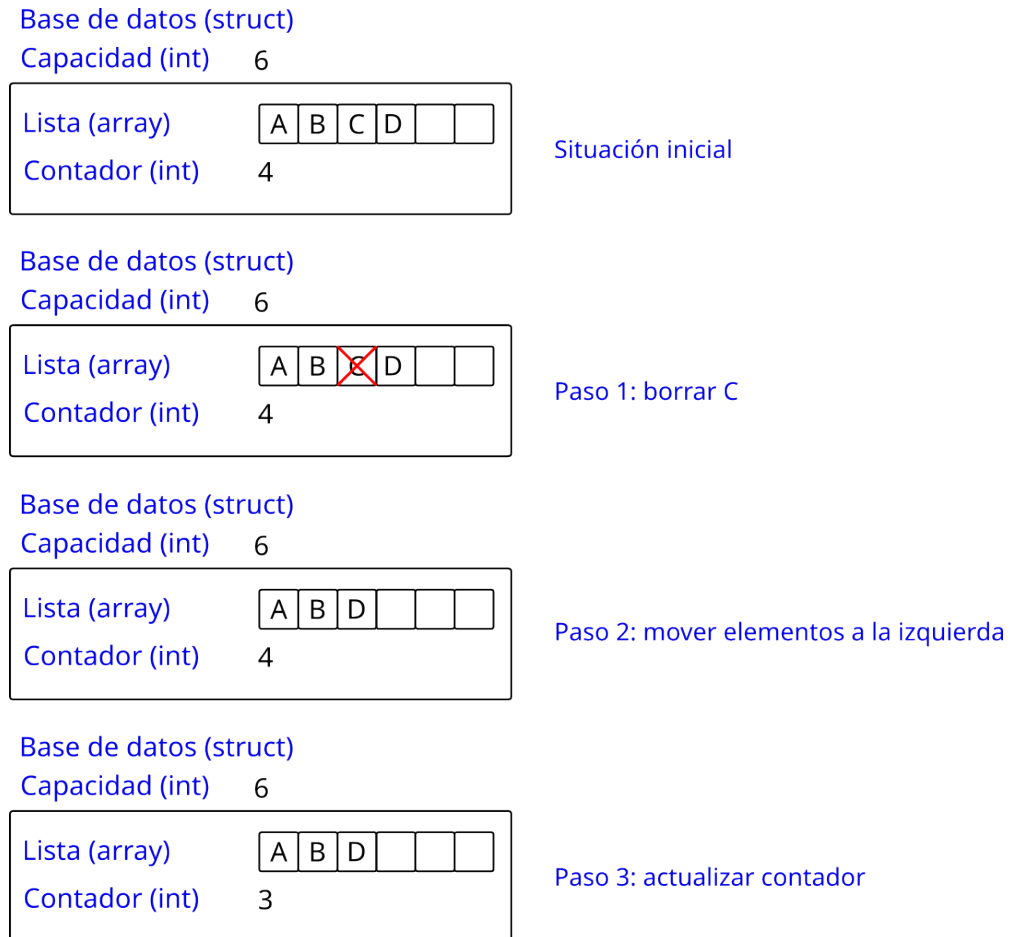
Las operaciones *Read* y *Update* necesitan recorrer los elementos válidos de la lista. Para ello, bastará con empezar a recorrer desde el primer elemento de la lista hasta el último elemento válido, que nos lo indica el campo *Contador*.

Quizás la operación más complicada es *Delete*, borrar un elemento de la lista. La forma de realizar esta operación permite comprender el sentido del adjetivo *Compactada* con el que se denominan estas listas.

Suponga que, en el ejemplo anterior, se quiere borrar de la lista el elemento C. Hay que proceder en varios pasos:

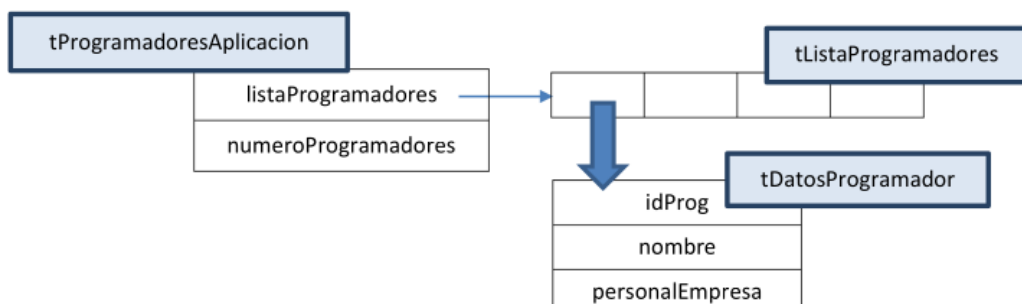
1. Borrar el tercer elemento de la lista, el C.
2. Mover los elementos posteriores al C una posición hacia la izquierda, para que todos los elementos de la lista estén en posiciones consecutivas desde el índice 0.
3. Actualizar el valor del contador, que ahora vale 3.

El proceso se esquematiza en la siguiente figura:



En realidad, el paso 2 (borrar C) no se hace de manera explícita, pues al mover el elemento D hacia la izquierda ya queda sobrescrito el tercer elemento con los datos de D, pero es útil tenerlo en mente al realizar el proceso.

En la primera parte de la Práctica 4 se utiliza una estructura similar a la descrita, con dos campos: la lista de programadores y el número de programadores. La capacidad de la lista es una constante del programa llamada `MAX_PROGRAMADORES`, para la que se utiliza una cláusula `#define` y cuyo valor es 4.



Vamos a resolver a modo de ejemplo las operaciones CRUD para una pequeña base de datos similar a la de los ejemplos anteriores. Utilizaremos una estructura llamada *Persona* que sirve para guardar el nombre de una persona y una estructura llamada *BaseDatos* que permite gestionar una lista con capacidad para 6 personas.

El código necesario para declarar los tipos de datos que va a utilizar el programa y crear una instancia de *BaseDatos* podría ser el siguiente:

```
#define CAPACIDAD 6

typedef struct {
    char nombre[20];
    char apellidos[40];
} Persona;
typedef struct {
    Persona lista[CAPACIDAD];
    int contador;
} BaseDatos;

int main() {
    BaseDatos db;
}
```

Para poder añadir personas a la lista, lo primero que hay que hacer es comprobar que no está llena. Para ello, se puede implementar una función, que hemos llamado *is\_full()*:

```
// Devuelve true si la B.D está llena
// y false en caso contrario
bool is_full(BaseDatos db) {
    bool result = false;
    if(db.contador >= CAPACIDAD) {
        result = true;
    }
    return result;
}
```

La función *is\_full()* recibe como argumento una *BaseDatos* y compara su campo *contador* con el valor de la constante *CAPACIDAD*. Si la lista está llena, devuelve *true* y, en caso contrario, devuelve *false*.

Ahora ya podemos implementar la función que permita añadir personas a la base de datos. Hemos creado una función llamada *add\_persona()*:

```
// Añade una Persona a la B.D.
// Devuelve true si la añade y false en caso contrario
bool add_persona(BaseDatos* db, const char nombre[]) {
    bool result = false;
    if(is_full(*db) == false) {
        Persona p;
        strcpy(p.nombre, nombre);
        db->lista[db->contador] = p;
        db->contador++;
        result = true;
    }
    return result;
}
```

La función `add_persona()` recibe como argumentos un puntero a la *BaseDatos* y una cadena de caracteres con el nombre que se quiere añadir a la lista. Es necesario que la función reciba un puntero a la base de datos, para poder hacer modificaciones en la misma. La función devuelve *true*, si la persona se añade a la lista y *false*, si la lista está llena y no se pudo añadir.

Observa que, lo que se añade a la lista es una *Persona*, por lo que primero se crea una instancia de *Persona*, con el nombre que se recibe como argumento y luego se añade a la lista. Se podría haber copiado directamente el valor del campo *nombre* en la estructura ya existente en la lista:

```
strcpy(db->lista[db->contador].nombre, nombre);
```

Para completar esta primera parte de la base de datos de personas, nos hace falta una función que permita hacer un listado de todas las personas que hay en la lista. Hemos llamado `list_database()` a dicha función:

```
// Lista todos los registros de la B.D.
void list_database(BaseDatos db) {
    for(int i=0; i<db.contador; i++) {
        printf("%s ", db.lista[i].nombre);
    }
    printf("\n");
}
```

Ahora se puede hacer una primera prueba del programa antes de implementar nuevas funcionalidades. El código que se muestra a continuación tiene los prototipos de las funciones anteriores. Debes incluir el código de las mismas a continuación de la función `main()`.

### Ejemplo 1 Primera prueba del programa de base de datos

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define CAPACIDAD 6

// Tipos de datos
typedef struct {
    char nombre[20];
} Persona;

typedef struct {
    Persona lista[CAPACIDAD];
    int contador;
} BaseDatos;

// Prototipos de funciones
bool is_full(BaseDatos db);
bool add_persona(BaseDatos* db, const char nombre[]);
void list_database(BaseDatos db);

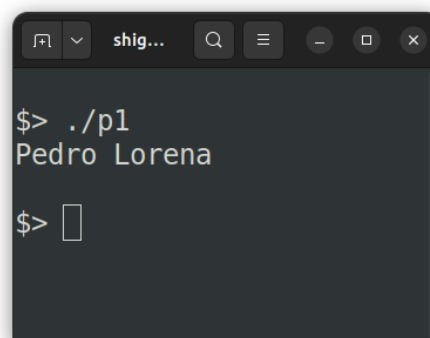
int main() {
    BaseDatos db;

    add_persona(&db, "Pedro");
    add_persona(&db, "Lorena");

    list_database(db);

    return 0;
}

// Incluir el código de las funciones
```



Vamos a implementar ahora el código para eliminar registros de la base de datos. Como hemos comentado, para borrar un registro y que la lista permanezca compactada, los registros posteriores al eliminado se deben mover una posición hacia el origen de la lista, de forma que no queden huecos.

Se muestra a continuación el código de una función llamada *delete\_index()*, que borra el registro que ocupa una posición determinada. Para ello, la función necesita un puntero a la base de datos, para poder hacer modificaciones en la misma y el índice del elemento que se quiere borrar. El código podría ser el siguiente:

```
// Elimina el registro de la B.D. cuyo índice
// se pasa como argumento. Devuelve true si
// se elimina y false si no se elimina
bool delete_index(BaseDatos* db, int index) {
    bool result = false;
    if(index >= 0 && index < db->contador) {
        for(int i=index+1; i<db->contador; i++) {
            db->lista[i-1] = db->lista[i];
        }
        db->contador--;
        result = true;
    }
    return result;
}
```

La función primero comprueba que el índice que se quiere eliminar está comprendido en el rango de datos válidos de la lista de nombres. Si es así, mueve los nombre posteriores de la lista una posición hacia el origen. La función devuelve *true* si se produce la eliminación y *false* en caso contrario.

Podemos probar el código anterior añadiendo la función *delete\_index()* al programa y utilizando la siguiente función *main()*:

## Ejemplo 2 Prueba de la función para borrar registros por índice

```
int main() {
    BaseDatos db;

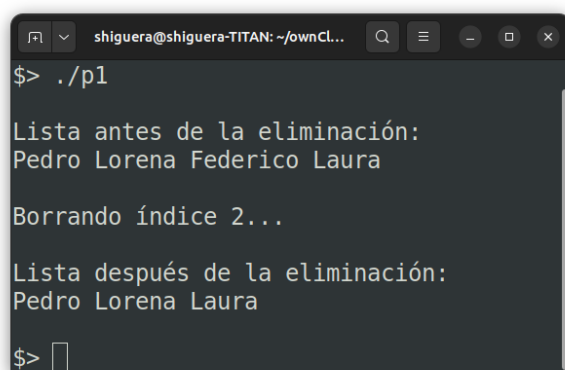
    add_persona(&db, "Pedro");
    add_persona(&db, "Lorena");
    add_persona(&db, "Federico");
    add_persona(&db, "Laura");

    printf("Lista antes de la eliminación:\n");
    list_database(db);

    printf("\nBorrando índice 2...\n\n");
    delete_index(&db, 2);

    printf("Lista después de la eliminación:\n");
    list_database(db);

    return 0;
}
```



```
shiguera@shiguera-TITAN: ~/ownCL...
$> ./p1

Lista antes de la eliminación:
Pedro Lorena Federico Laura

Borrando índice 2...

Lista después de la eliminación:
Pedro Lorena Laura

$> 
```

Otra posibilidad es borrar el registro correspondiente a un nombre concreto. Una forma de hacerlo es implementar un función, que hemos llamado *find\_nombre()*, que devuelva el índice que le corresponde a un nombre concreto y luego utilizar la función *delete\_index()* para borrar el registro.

El código de la función *find\_nombre()* podría ser el siguiente:

```
// Busca en la B.D. el primer registro cuyo
// nombre coincida con el del argumento
// Devuelve la posición del registro si lo
// encuentra y -1 si no lo encuentra
int find_nombre(BaseDatos db, const char nombre_buscado[]) {
    bool seguir = true;
    int resultado = -1;
    int posicion = 0;
    while(seguir == true && posicion<CAPACIDAD) {
        if(strcmp(db.lista[posicion].nombre, nombre_buscado) == 0) {
            resultado = posicion;
            seguir = false;
        } else {
            posicion++;
        }
    }
    return resultado;
}
```

La función devuelve un índice válido, si encuentra el nombre en la lista o el valor -1 si no lo encuentra.

Si se obtiene un índice válido, se puede usar la función *delete\_index()*, que se codificó anteriormente, para eliminar el registro. Podemos implementar una función específica llamada *delete\_nombre()* que utilice la función *find\_nombre()* para realizar la operación:

```
// Borra de la B.D. el primer registro cuyo
// nombre coincida con el del argumento
// Devuelve true si se borra un registro
// y false si no se borra ninguno
bool delete_nombre(BaseDatos* db, const char nombre[]) {
    bool result = false;
    int posicion = find_nombre(*db, nombre);
    if (posicion != -1) {
        delete_index(db, posicion);
        result = true;
    }
    return result;
}
```

La función *delete\_nombre()*, al igual que sucedía con la función *delete\_index()*, modifica la base de datos y necesita que el parámetro sea del tipo puntero, lo que en el curso llamamos «*parámetro de salida pasado por referencia*».

Podemos probar la nueva función añadiendo el código al programa y modificando la función *main()*:

### Ejemplo 3 Prueba de la función `delete_nombre()`

```
int main() {
    BaseDatos db;

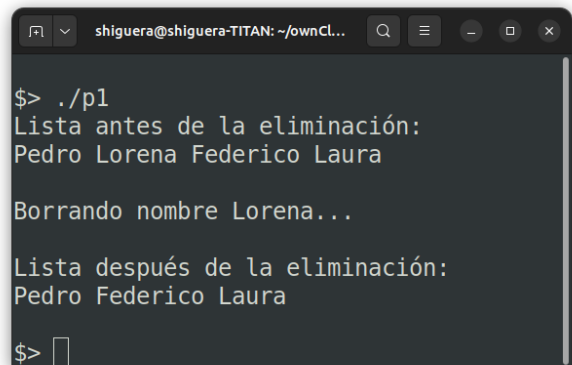
    add_persona(&db, "Pedro");
    add_persona(&db, "Lorena");
    add_persona(&db, "Federico");
    add_persona(&db, "Laura");

    printf("Lista antes de la eliminación:\n");
    list_database(db);

    printf("\nBorrando nombre Lorena...\n\n");
    delete_nombre(&db, "Lorena");

    printf("Lista después de la eliminación:\n");
    list_database(db);

    return 0;
}
```



```
shiguera@shiguera-TITAN: ~/ownCl...
$> ./p1
Lista antes de la eliminación:
Pedro Lorena Federico Laura

Borrando nombre Lorena...

Lista después de la eliminación:
Pedro Federico Laura
$> 
```

## 3 ARRAYS CON HUECOS

En este apartado vamos a explicar otra forma de gestionar una lista de datos en un array, la denominada *array con huecos* o *array con flag*.

En el apartado anterior hemos comentado algunos de los problemas al utilizar un array para gestionar una lista. En concreto, que el array no proporciona información acerca de su capacidad ni de cuantos elementos del array están ocupados con datos válidos.

En el sistema de *lista compactada*, el array se incluye como campo de una estructura. Los datos válidos siempre ocupan las primeras posiciones del array, sin dejar huecos. La estructura tiene otro campo que indica cuántos datos válidos hay en el array. Además, el programa necesita mantener en alguna variable o constante la capacidad total del array.

El sistema del *array con huecos* utiliza directamente un array como base de datos. Lo que se hace es crear, para cada elemento del array, una estructura con dos campos. A dicha estructura le vamos a llamar *Celda*, de forma genérica. El primer campo de esta estructura es el dato en sí; el segundo campo es un *bool* que indica si la *Celda* está ocupada con datos válidos o no. Los elementos del array que hace de base de datos son *Celdas*, que pueden tener datos válidos o no. El esquema es el de la siguiente figura:

Base de datos (array de Celdas)  
Capacidad (int) 6



Celda (struct)

Dato (cualquier tipo)  
ocupada (bool)



Aunque podríamos utilizar un ejemplo similar al del apartado anterior, con una base de datos de personas, vamos a utilizar otro ejemplo que permita apreciar por qué en algunos casos puede ser más conveniente utilizar esta técnica.

Suponga un pequeño hotel que tiene 4 habitaciones, numeradas del 100 al 103. Cada habitación tiene su precio diario de alquiler y puede estar ocupada o no. La aplicación tiene que gestionar las reservas de habitaciones. Podría iniciarse de la siguiente forma:

#### Ejemplo 4 Ejemplo de un array con huecos

```
#include <stdbool.h>

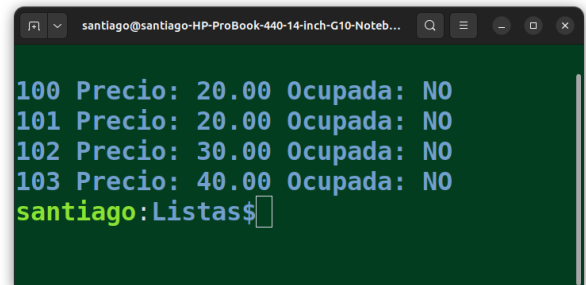
#define NUM_HABITACIONES 4

typedef struct {
    int numero;
    double precio;
} Habitacion;

typedef struct {
    Habitacion habitacion;
    bool ocupada;
} Celda;

void list_habitaciones(const Celda habitaciones[]) {
    for(int i=0; i<NUM_HABITACIONES; i++) {
        char ocup[] = "NO";
        if(habitaciones[i].ocupada == true) {
            strcpy(ocup, "SI");
        }
        printf("%d Precio: %.2lf Ocupada: %s\n",
            habitaciones[i].habitacion.numero,
            habitaciones[i].habitacion.precio,
            ocup);
    }
}

int main() {
    Celda hotel[NUM_HABITACIONES] = {
        {{100, 20.0}, false},
        {{101, 20.0}, false},
        {{102, 30.0}, false},
        {{103, 40.0}, false}
    };
}
```



```
santiago@santiago-HP-ProBook-440-14-Inch-G10-Noteb...
100 Precio: 20.00 Ocupada: NO
101 Precio: 20.00 Ocupada: NO
102 Precio: 30.00 Ocupada: NO
103 Precio: 40.00 Ocupada: NO
santiago:Listas$
```

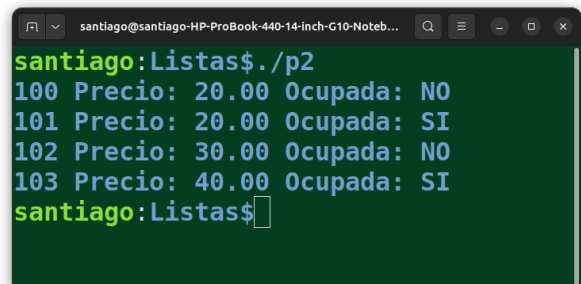
Los datos de cada habitación utilizan una estructura *Habitacion*. La base de datos de las habitaciones es un array de *Celdas* que hemos llamado *hotel*. Cada *Celda* tiene un campo que son los datos de una habitación y un *bool* que indica si está libre u ocupada. Observa cómo se ha inicializado el array *hotel*. Inicialmente, todas las habitaciones están libres.

Hemos añadido una función que haga un listado de las habitaciones y si están libres u ocupadas.

Cuando un cliente ocupa una habitación, puede ser cualquiera. Por ejemplo, podría suceder que el primer cliente ocupara la habitación 103 y el segundo cliente la habitación 101. El proceso de declarar ocupada una habitación consiste simplemente en poner a *true* el campo *ocupada* de la *Celda* correspondiente:

### Ejemplo 5 Ocupando un par de habitaciones

```
int main() {  
    Celda hotel[NUM_HABITACIONES] = {  
        {{100, 20.0}, false},  
        {{101, 20.0}, false},  
        {{102, 30.0}, false},  
        {{103, 40.0}, false}  
    };  
  
    hotel[3].ocupada = true;  
    hotel[1].ocupada = true;  
    list_habitaciones(hotel);  
}
```



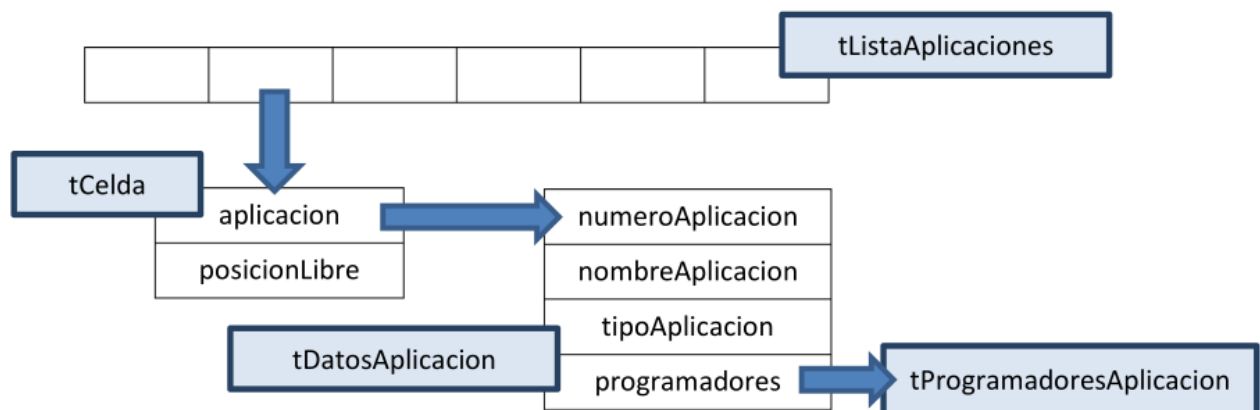
```
santiago:Listas$ ./p2  
100 Precio: 20.00 Ocupada: NO  
101 Precio: 20.00 Ocupada: SI  
102 Precio: 30.00 Ocupada: NO  
103 Precio: 40.00 Ocupada: SI  
santiago:Listas$
```

El proceso de vaciar una habitación consiste en poner a *false* el campo *ocupada* de la *Celda* correspondiente.

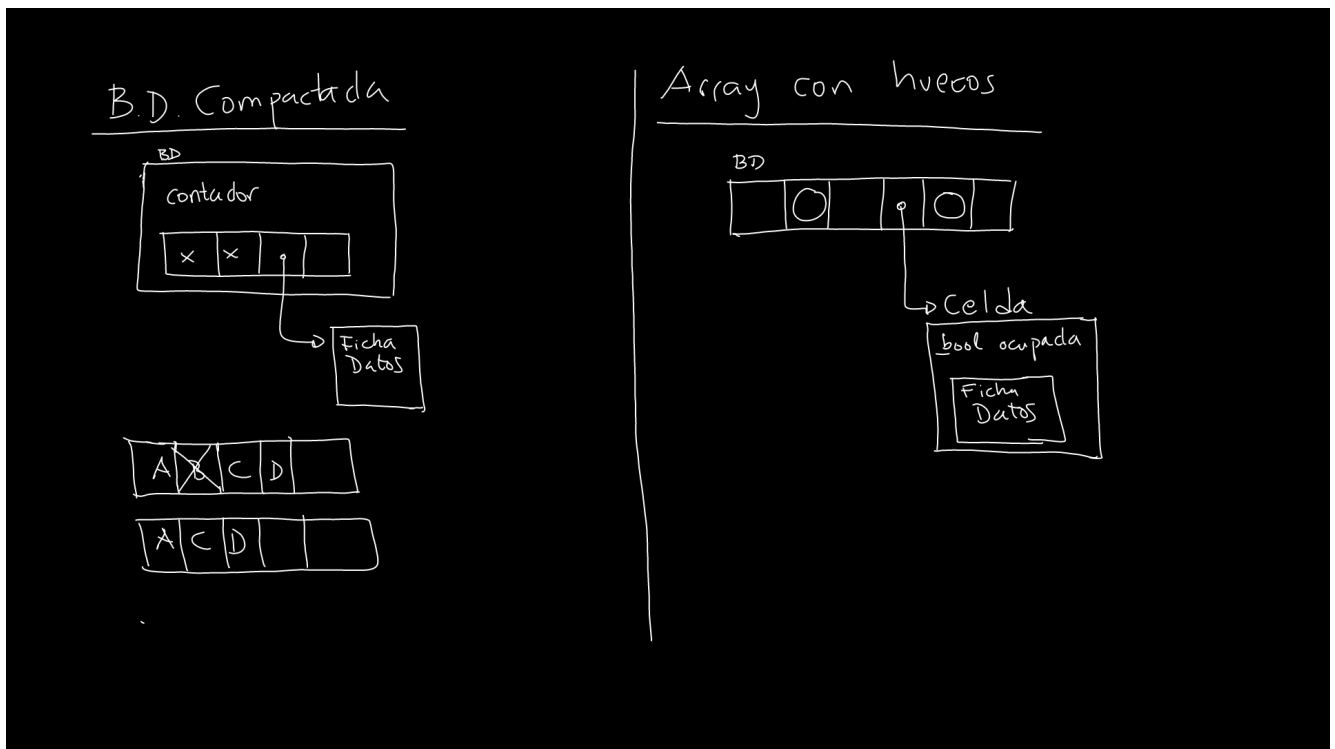
Dejamos a tu imaginación el tipo de funciones que se necesitarían para gestionar este tipo de base de datos, por ejemplo, buscar una habitación libre, comprobar si determinada habitación está libre u ocupada, sumar los importes de las habitaciones que están ocupadas, o cualquier otra.

Un ejemplo más completo podría tener un campo en la *Celda* de la habitación que fueran los datos del cliente que la ocupa, que a su vez sería algún tipo de estructura. Es fácil darse cuenta de por qué este método se llama *array con huecos*.

Puedes ver un ejemplo más complejo de utilización de esta técnica combinada con la del array compactado lo puedes ver en la segunda parte de la Práctica 4, que utiliza un array con huecos para gestionar las aplicaciones de una empresa de desarrollo de software y, en cada aplicación, los datos de la aplicación y los programadores que intervienen, siguiendo el siguiente esquema:



## 4 COMPARACIÓN ENTRE ARRAY COMPACTADO Y ARRAY CON HUECOS



Las técnicas de *arrays compactados* y *arrays con huecos* son dos estrategias diferentes para manejar estructuras datos utilizando arrays. Se diferencian especialmente en la forma de eliminar elementos y de gestionar espacios vacíos. Las principales operaciones que se pueden hacer se resumen en la Tabla 4.

Operaciones para manejar arrays compactados y arrays con huecos		
Operación	A. Compactado	A. con huecos
Base de datos	Estructura	Array de celdas
Contar registros con datos	Valor del campo contador	Hay que recorrer entero el array y contar celdas ocupadas
Añadir un registro	Se añade en la posición marcada por el campo contador	Hay que buscar una celda que no esté ocupada y utilizarla para añadir el nuevo registro
Listar registros con datos	Bucle de cero a contador	Bucle recorriendo todo el array y listar los elementos con la celda ocupada
Borrar un registro	Mover los registros siguientes una posición hacia atrás	Poner el campo ocupado a false

En un array compactado, todos los elementos están almacenados de manera contigua, sin espacios vacíos entre ellos. Cuando se elimina un elemento del array, los elementos restantes se desplazan para llenar el hueco dejado por el elemento eliminado. Esto asegura que no haya huecos en el array, pero puede ser costoso en términos de rendimiento debido a la necesidad de mover múltiples elementos.

Ventajas:

- Acceso rápido y eficiente a los elementos.
- Uso eficiente de la memoria.

Desventajas:

- Costoso en términos de tiempo cuando se eliminan elementos, ya que puede requerir mover muchos elementos.

#### **Ejemplo 6** Ejemplo sencillo de array compactado

```
#include <stdio.h>

#define SIZE 5

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void removeElement(int arr[], int *size, int index) {
    if (index < 0 || index >= *size) {
        printf("Índice fuera de rango\n");
        return;
    }
    // Mover los elementos para llenar el hueco
    for (int i = index; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--; // Reducir el tamaño del array
}

int main() {
    int array[SIZE] = {1, 2, 3, 4, 5};
    int size = SIZE;

    printf("Array original: ");
    printArray(array, size);

    // Eliminar el elemento en la posición 2 (valor 3)
    removeElement(array, &size, 2);

    printf("Array después de eliminar el elemento en la posición 2: ");
    printArray(array, size);

    return 0;
}
```

En un array con huecos, los elementos no se desplazan cuando se elimina un elemento. En su lugar, el espacio dejado por el elemento eliminado se marca como un hueco. Esto puede hacer que el array sea menos eficiente en términos de uso de memoria, pero puede ser más rápido para operaciones de eliminación.

Ventajas:

- Eliminación de elementos rápida y eficiente, ya que no es necesario mover otros elementos.
- Menos operaciones de movimiento de datos, lo que puede mejorar el rendimiento.

Desventajas:

- Acceso menos eficiente debido a la presencia de huecos.
- Uso de memoria menos eficiente, ya que el array puede tener muchas posiciones vacías.

### Ejemplo 7 Ejemplo sencillo de array con huecos

```
#include <stdio.h>

#define SIZE 5
#define FLAG -1

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] != FLAG) {
            printf("%d ", arr[i]);
        } else {
            printf("_ "); // Representa un hueco
        }
    }
    printf("\n");
}

int main() {
    int array[SIZE] = {1, 2, 3, 4, 5};

    // Eliminar el elemento en la posición 2 (valor 3)
    array[2] = FLAG;

    // Imprimir el array
    printArray(array, SIZE);

    return 0;
}
```

En resumen, la elección entre arrays compactados y arrays con huecos depende de las necesidades específicas de la aplicación, como la frecuencia de eliminación de elementos y la importancia del acceso rápido a los mismos.