MANUAL DE PRÁCTICAS
ESTRUCTURA DE DATOS

UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE LOS LAGOS
DIVISIÓN DE ESTUDIOS DE LA BIODIVERSIDAD E INNOVACIÓN TECNOLÓGICA
DEPARTAMENTO DE CIENCIAS EXACTAS Y TECNOLOGÍA

UNIVERSIDAD DE GUADALAJARA
Centro Universitario de los Lagos
División de Estudios de la Biodiversidad e Innovación Tecnológica
Departamento de Ciencias Exactas y Tecnología

MANUAL DE PRÁCTICAS

DATOS GENERALES

Materia:	Estructura de Archivos	Créditos:	7	Clave:	H0646
Carrera:	Ingeniería en Electrónica y Computación			Periodo:	2018 A
Maestro:	M. C. E. Candelario Agustín Cú Guerrero	Fechas en que se cursará:			16/enero/2018 al 26/mayo/2018

Tipo de curso: Tradicional () Taller ( ) Seminario ( ) Laboratorio ( ) Otro ( ) Curso-Taller (X)

Enrique Díaz de León No. 1144, Colonia Paseos de la Montaña C.P. 47460.
Lagos de Moreno, Jalisco, México Tels. [52] (474) 742 4314, 742 3678, 746 4563 Ext. 66511 Fax. Ext. 66527
www.lagos.udg.mx

Contenido

PRÁCTICA 1. Archivo descriptor 5
Introducción 5
Objetivo 5
Material y equipo 5
Desarrollo de la práctica 5
Resultados 6
Conclusiones 6
Bibliografía 6
PRÁCTICA 2. Apertura de un archivo de texto y archivo binario 7
Introducción 7
Objetivo 8
Material y equipo 8
Desarrollo de la práctica 8
Resultados 9
Conclusiones 9
Bibliografía 10
PRÁCTICA 3. Registros de longitud fija y de longitud variable 10
Introducción 10
Objetivo 10
Material y equipo 10
Desarrollo de la práctica 10
Resultados 12
Conclusiones 12
Bibliografía 12
PRÁCTICA 4. Acceso secuencial 13

Introducción 13
Objetivo 13
Material y equipo 13
Desarrollo de la práctica 13
Resultados 15
Conclusiones 15
Bibliografía 15
PRÁCTICA 5. Acceso directo 16
Introducción 16
Objetivo 16
Material y equipo 16
Desarrollo de la práctica 16
Resultados 21
Conclusiones 21
Bibliografía 21
PRÁCTICA 6. Búsqueda secuencial y directa 22
Introducción 22
Objetivo 22
Material y equipo 22
Desarrollo de la práctica 22
Resultados 24
Conclusiones 24
Bibliografía 24
PRÁCTICA 7. Ordenamiento Shell 25
Introducción 25
Objetivo 25

Material y equipo 25
Desarrollo de la práctica 25
Resultados 26
Conclusiones 26
Bibliografía 26
PRÁCTICA 8. Funciones Hash 27
Introducción 27
Objetivo 27
Material y equipo 27
Desarrollo de la práctica 27
Resultados 32
Conclusiones 32
Bibliografía 32
PRÁCTICA 9. Arboles binarios 33
Introducción 33
Objetivo 33
Material y equipo 33
Desarrollo de la práctica 33
Resultados 38
Conclusiones 38
Bibliografía 38

PRÁCTICA 1. Archivo descriptor

Introducción

Al utilizar las funciones de E/S continua de ILE C/400 definidas por el ANSI (American National Standards Institute) para ejecutar operaciones en un archivo, el archivo se identifica mediante la utilización de punteros. Al utilizar las funciones C del sistema de archivos integrado, el archivo se identifica especificando un descriptor de archivo. Un *descriptor de archivo* es un entero positivo que debe ser exclusivo en cada trabajo. El trabajo utiliza un descriptor de archivo para identificar un archivo abierto al realizar operaciones sobre el archivo. El descriptor de archivo se representa con la variable fildes en las funciones C que operan en el sistema de archivos integrado y con la variable descriptor en las funciones C que operan en sockets.
Cada descriptor de archivo hace referencia a una *descripción de archivo abierto*, que contiene información como, por ejemplo, desplazamiento de archivo, estado del archivo y modalidades de acceso para el archivo. Varios descriptores de archivo pueden hacer referencia a la misma descripción de archivo abierto, pero un descriptor de archivo solo puede hacer referencia a una descripción de archivo abierto (IBM, 2017).

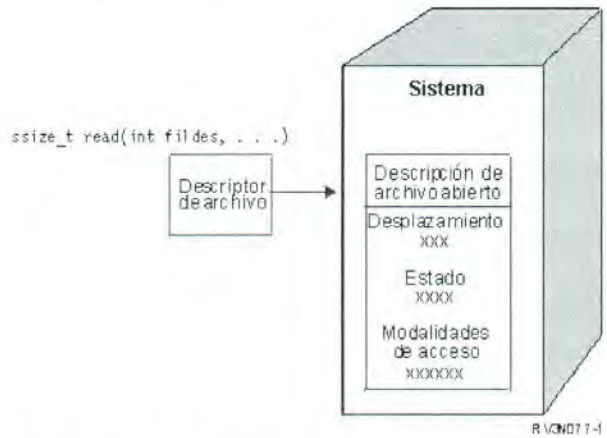

Figura 1 Descriptor de archivo y descripción de archivo abierto

Objetivo

El alumno desarrollará un programa para implementar el descriptor de archivos, usando la función fdopen. Esta función se comporta como la función fopen, pero en vez de abrir un archivo dado su nombre, usa el identificador de archivo para abrirlo.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C.

Desarrollo de la práctica

Capture el siguiente código, en el lenguaje C, revise que este bien ejecute el programa.

```
/*
 * fdopen example code
 * http://code-reference.com/c/stdio.h/fdopen
 */
#include <stdio.h>
#ifdef __linux__
```

```
  #include <unistd.h>
#else
  // windows
  #include <io.h>
#endif

int main(void) {
  FILE *fileptr, *fileptr2; // pointer
  int fdesc,fdesc2;              // descriptor
  char filename[42]="text.txt";

  fileptr=fopen(filename, "w+");

  /* error handling*/
  if(!fileptr)
    perror(NULL);
  fdesc = fileno(fileptr);

  printf("file descriptor %d for filename %s\n",fdesc ,filename);
  fdesc2=dup(fdesc);
  printf("copied file descriptor %d\n\n",fdesc2);

  printf("now we open a stream with fdopen\n");
  /**
  * fdopen
  **/
  fileptr2 = fdopen(fdesc2, "w");
   if(!fileptr2)
     perror(NULL);

  fprintf(fileptr,"Content of %s\n",filename);
  fprintf(fileptr2,"also content of %s\n",filename);
  fprintf(stdout,"content is written to %s\n",filename);
  return 0;
}
```

Investigue las instrucciones que no conozca, analice el código, y trate de interpretar lo que se está haciendo en el código.

Ahora de acuerdo con lo aprendido desarrolle un programa donde obtenga los descriptores de archivo de varios archivos que usted elija.

Resultados

Se espera que el alumno identifique que significa el número que está regresando el programa.
El alumno redactara los resultados que obtuvo.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

IBM. (10 de 12 de 2017). *IBM*. Obtenido de IBM: https://www.ibm.com/support/knowledgecenter/es/ssw_i5_54/ifs/rzaaxfildes.htm

El alumno complementara la bibliografía con el material extra que haya consultado.

PRÁCTICA 2. Apertura de un archivo de texto y archivo binario

Introducción

Un archivo es una colección de datos guardados en un dispositivo de almacenamiento permanente.

Hay dos tipos de archivos, archivos de texto y archivos binarios. Un archivo de texto es una secuencia de caracteres organizados en líneas terminadas por un carácter de nueva línea. En estos archivos se pueden almacenar canciones, fuentes de programas, base de datos simples, etc. Los archivos de texto se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho.

Un archivo binario es una secuencia de bytes que tienen una correspondencia uno a uno con un dispositivo externo. Así que no tendrá lugar ninguna traducción de caracteres. Además, el número de bytes escritos (leídos) será el mismo que los encontrados en el dispositivo externo. Ejemplos de estos archivos son Fotografías, imágenes, texto con formatos, archivos ejecutables (aplicaciones), etc.

Se puede conseguir la entrada y la salida de datos a un archivo a través del uso de la biblioteca de funciones; C no tiene palabras claves que realicen las operaciones de E/S. La siguiente tabla da un breve resumen de las funciones que se pueden utilizar. Se debe incluir la librería STDIO.H. Observe que la mayoría de las funciones comienzan con la letra "F", esto es un vestigio del estándar C de Unix.

Nombre	Función
fopen()	Abre un archivo.
fclose()	Cierra un archivo.
fgets()	Lee una cadena de un archivo.
fputs()	Escribe una cadena en un archivo
fseek()	Busca un byte especifico de un archivo.
fprintf()	Escribe una salida con formato en el archivo.
fscanf()	Lee una entrada con formato desde el archivo.
feof()	Devuelve cierto si se llega al final del archivo.
ferror()	Devuelve cierto si se produce un error.
rewind()	Coloca el localizador de posición del archivo al principio del mismo.
remove()	Borra un archivo.
fflush()	Vacía un archivo.

El puntero a un archivo.

El puntero a un archivo es el hilo común que unifica el sistema de E/S con buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia identifica un archivo especifico y utiliza la secuencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Un puntero a un archivo es una variable de tipo puntero al tipo FILE que se define en **STDIO.H**. Un programa necesita utilizar punteros a archivos para leer o escribir en los mismos.

Para obtener una variable de este tipo se utiliza una secuencia como esta: FILE *F;

Apertura de un archivo.

La función fopen() abre una secuencia para que pueda ser utilizada y la asocia a un archivo. Su prototipo es: FILE *fopen(const char nombre_archivo, cost charmodo); Donde nombre_archivo es un puntero a una cadena de caracteres que representan un nombre valido del archivo y puede incluir una especificación del directorio. La cadena a la que apunta modo determina como se abre el archivo. La siguiente tabla muestra los valores permitidos para modo.

Modo	Significado
r	Abre un archivo de texto para lectura.
w	Crea un archivo de texto para escritura.
a	Abre un archivo de texto para añadir.
rb	Abre un archivo binario para lectura.
wb	Crea un archivo binario para escritura.
ab	Abre un archivo binario para añadir.
r+	Abre un archivo de texto para lectura / escritura.
w+	Crea un archivo de texto para lectura / escritura.
a+	Añade o crea un archivo de texto para lectura / escritura.
r+b	Abre un archivo binario para lectura / escritura.
w+b	Crea un archivo binario para lectura / escritura.
a+b	Añade o crea un archivo binario para lectura / escritura.

La función fopen() devuelve un puntero a archivo. Un programa nunca debe alterar el valor de ese puntero. Si se produce un error cuando se está intentando abrir un archivo, fopen() devuelve un puntero nulo.

Se puede abrir un archivo bien en modo texto o binario. En la mayoría de las implementaciones, en modo texto, las secuencias de retorno de carro / salto de línea se convierten a caracteres de salto de línea en lectura. En la escritura, ocurre lo contrario: los caracteres de salto de línea se convierten en salto de línea. Estas conversiones no ocurren en archivos binarios.

Cierre de un archivo.

La función fclose() cierra una secuencia que fue abierta mediante una llamada a fopen(). Escribe toda la información que todavía se encuentre en el buffer en el disco y realiza un cierre formal del archivo a nivel del sistema operativo. Un error en el cierre de una secuencia puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y posibles errores intermitentes en el programa. El prototipo de esta función es: **int fclose(FILE *F);**

Donde F es el puntero al archivo devuelto por la llamada a fopen(). Si se devuelve un valor cero significa que la operación de cierre ha tenido éxito. Generalmente, esta función solo falla cuando un disco se ha retirado antes de tiempo o cuando no queda espacio libre en el mismo.

Objetivo

El estudiante aplique los conocimientos para distinguir entre archivos de texto y binarios

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Capture el siguiente código, en el lenguaje de programación C y ejecútelo. (Nota: en el nombre del archivo sustitúyalo con el nombre del archivo de texto que usted desea abrir).

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        FILE *archivo;
        int caracter;
```

```
    archivo = fopen("prueba.txt","r");

    if (archivo == NULL)
  {
    printf("\nError de apertura del archivo. \n\n");
  }
  else
  {
    printf("\nEl contenido del archivo de prueba es \n\n");
    while((caracter = fgetc(archivo)) != EOF)
      {
          printf("%c",caracter);
      }
  }
  fclose(archivo);
    return 0;
}
```

Realice dos programas.
Diseñar un programa que, dado un archivo de texto, solicite al usuario una palabra y cuente el número de las mismas que hay en el archivo.

Escribe un programa C++ que lea el archivo de texto y escriba su contenido en un archivo binario alumnos.dat utilizando una estructura que contenga los datos del alumno.
Cada línea del archivo contiene los datos de un alumno, con el siguiente formato:

caracteres 01 a 20: Apellido 1
caracteres 21 a 40: Apellido 2
caracteres 41 a 60: Nombre
caracteres 61 a 70: Código del alumno

Ejemplo de contenido del archivo alumnos.txt:

Apellido 1	Apellido 2	Nombre	Código
González	Salcedo	Sancho	1023655890
López	Rodríguez	Ignacio	2258962122
Martínez	Pascual	Lucas	5893143333
Fernández	Aznar	Jesús	1458963250
García	García	Ana	8899204555

Resultados
Se espera que el alumno identifique los archivos de texto y binario, así como la forma de acceso.

El alumno redactara los resultados que obtuvo.

Resultados
Se espera que el alumno desarrolle un programa donde aplique el proceso que hay que seguir para la manipulación de archivos binarios y de texto.

Conclusiones
El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación*. México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java*. Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

PRÁCTICA 3. Registros de longitud fija y de longitud variable

Introducción

Un registro es un conjunto de campos que permanecen juntos cuando el archivo es visto en términos de organización de alto nivel.

En términos de programación lo que se busca es poder leer "registros" de archivos como un todo (buffer en memoria) para poder separar cada uno de sus campos.
Métodos para organizar registros en archivos

Requerir que los registros tengan una longitud fija (bytes)
Requerir que los registros tengan un número fijo de campos
Comenzar cada registro con un indicador de la longitud (suma de todos los bytes de cada campo en el registro)
Utilizar un segundo archivo para mantener una bitácora del byte de inicio donde comienza cada registro
Colocar un delimitador al final de cada registro para separarlo del siguiente

Objetivo

Que el alumno identifique las conveniencias de utilizar registros de longitud fija y variable.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Deseamos almacenar la siguiente información que tenemos en memoria en disco.

Mary Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater OK 74075	Ada OK 74820

Si consideramos el siguiente método para almacenar dichos datos.

```
ostream & operator << (ostream & outFile, Person & p)
{  //insert (write) fields into stream
    outputFile << p.lastName
       << p.FirstName
       << p.Address
       << p.City
       << p.State
       << p.ZipCode;
     return outputFile;
}
```

Los guardaremos en un archivo de manera consecutiva, tal como se fue especificando, pero ahora tenemos un gran problema. La información no se puede separar para distinguir los distintos campos que componen cada registro.

Existen distintas maneras de agregar estructuras de datos a archivos y mantener la identidad de los campos:

Forzar a los campos a tener una longitud fija
Comenzar cada campo con un indicador de la longitud del campo
Colocar un delimitador al final de cada campo para separarlo del siguiente
Usar una pareja "keyword=value" para identificar cada campo y su contenido

Forzar a los campos a tener una longitud fija
Los campos (nombre, dirección, estado) de nuestro ejemplo anterior tenían longitudes variables. Pero nosotros podemos pensar en establecer una medida fija para cada uno de los campos en el caso del ejemplo de Person cada campo tiene una longitud y el tamaño total de un registro siempre sera de 67 bytes (11+11+16+16+3+10)

```
class Person { public:
        char last[11];
        char first [11];
        char address [16];
        char city[16];
        char state[3];
        char zip[10];
};
```

De manera que el archivo quedaría de la siguiente manera (recordar cubrir los espacios vacíos):

Ames	Mary	123 Maple	Stillwater	OK	74075
Mason	Alan	90 Eastgate	Ada	OK	74820

Para recuperar la información se puede hacer matemáticamente leyendo el número de bytes/chars correspondientes a cada campo

Desventajas
El archivo tiende a hacerse demasiado grande
¿Qué sucede si algún campo en un registro necesita más espacio que lo acordado?, necesitamos arreglar la longitud de ese campo en todos los registros, desperdiciando espacio y tiempo.

Ventaja
Cuando de antemano sabemos que nuestros campos siempre tendrán la misma longitud es una buena opción

Comenzar cada campo con un indicador de la longitud del campo
Se coloca antes de cada campo su longitud, generalmente si los campos no son muy largos esta medida ocupará un solo byte (256 bytes de longitud)

```
04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820
```

Colocar un delimitador al final de cada campo para separarlo del siguiente
Escribir un caracter especial entre cada campo de manera que se pueda preservar la identidad La decisión de cual caracter "especial" utilizar es muy importante ya que no debe utilizarse en ninguno de los campos.
En algunos casos los espacios en blanco, el salto de línea o el tabulador pueden ser una buena opción.

```
Ames|Mary|123 Maple|Stillwater|OK|74075
Mason|Alan|90 Eastgate|Ada|OK|74820
```

Usar una pareja "keyword=value" para identificar cada campo y su contenido

Conocidas como estructuras de auto-descripción esta manera de almacenar la información es muy útil en la organización de archivos

Puede combinarse con otro esquema de los antes mencionados, esto con la finalidad de mostrar la división entre las parejas

Desventaja:
El desperdicio de espacio en disco, casi un 50%.

```
last=Ames| first=Mary|address=123 Maple|city=Stillwater|state=OK|zip=74075
```

Crea un programa en base a la información anterior que sea funcional, y se puedan manejar los diferentes casos que se presentaron.

Resultados

Crear un programa que maneje registros de longitud fija y variable.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación*. México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java*. Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

UNIVERSIDAD DE GUADALAJARA
Centro Universitario de los Lagos
División de Estudios de la Biodiversidad e Innovación Tecnológica
Departamento de Ciencias Exactas y Tecnología

PRÁCTICA 4. Acceso secuencial

Introducción

El acceso secuencial es justamente este tipo de acceso, donde la lectura del archivo se da en forma de lectura tradicional occidental, es decir, tomando en cuenta que se consulta un archivo de tipo texto o binario, el cursor recorrerá las posiciones de arriba hacia abajo y de izquierda a derecha.

Se basa en el hecho de que, para acceder a una determinada posición de los mismos, hemos de recorrer desde el principio todas las posiciones hasta llegar a la deseada.

En los archivos secuenciales, los datos se almacenan de igual forma, lo cual quiere decir que los datos nuevos almacenados se insertaran al final del documento, así, el primer registro guardado se encontrara en el inicio del documento. Cada registro toma una posición, con la cual se accesara. En los archivos secuenciales no existen espacios entre registros, por lo cual, cada dato tomara su número consecutivo, primer registro, segundo registro etc.

Si bien la lectura del archivo es bastante fácil de seguir, no es óptimo cuando se trata de búsquedas o lecturas que no son secuenciales, es decir, si el dato que se busca se encuentra en una posición anterior a la ubicación del puntero, se deberá leer de nuevo el documento desde el inicio.

Objetivo

El alumno desarrolle programas implementando archivos. (Archivos Texto, Archivos de Registros, Acceso Secuencial)

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

La interfaz de la clase agenda es:

```
class Agenda : public fstream {  public:   Agenda(char * nombre_agenda);
  void agregar_contacto(Registro & reg)
  void buscar(string & cad, Registro * arr_reg, int & c) }
```

Observe que una Agenda ES un archivo, por lo tanto Usted puede invocar los métodos de la clase fstream sin necesidad de colocar ningún objeto. Para los puristas, se puede hacer con (*this).

```
(*this).open("nombre_archivo.dat");
```

El método "agregar_contacto" agrega, en cualquier parte del archivo, un registro de tipo Registro. Por comodidad podría hacerlo al final del archivo.

El método "buscar" se encarga de hacer una búsqueda sobre todos los campos de todos los registros del archivo. Para ello, observe que, se tiene una cadena y que podría ser subcadena de cualquiera de los campos en el archivo. Para entender cabalmente este método, remitase a la definición de subcadena.

Suponga que Usted ha declarado un tipo Registro con los siguientes campos:

```
struct RegAgenda {
```

```
string Nombre;
string Apellido;
int edad;
string Telefono;
string email; };
```

La búsqueda sobre los campos de tipo string puede hacerse con la función "find" de la clase "string" de la STL. Por ejemplo:

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
int main() {
string st("Aún hay camino por recorrer");
  // intentemos buscar la palabra camino en la
// frase anterior.
  int r = st.find("camino");
  if (r >= 0)
{
 cout << "Encontrado!!!" << endl;
}
  cout << "Valor del resultado: " << r << endl;
// Note que si la palabra no es encontrada dentro
// de la cadena, el valor de retorno es -1.
  int cedula_como_entero = 10248583;
  char cedula_como_cadena[12];
// La funcion sprintf se encarga de escribir en una
// cadena (char *) algun contenido segun se especifique
// en el formato. En el ejemplo, se convierte la cedula
// de tipo entero, a una cedula del tipo char *. Esta
// sera pasada luego a una variable tipo string.
  sprintf(cedula_como_cadena, "%i", cedula_como_entero);
  string st_ced(cedula_como_cadena);
// Supongamos que queremos buscar cedulas que tengan
// la cadena 102, por supuesto el ejemplo cumple con
// esta restriccion.
  r = st_ced.find("102");
if (r >= 0)
{
cout << "Encontrado!!!" << endl;
}
  return 0;
}
```

La función buscar devuelve un arreglo de registros que cumplen con la cadena solicitada. La cadena solicitada debe ser buscada exahustivamente en todo el registro.

Es posible que algunos campos dentro del registro no sean del tipo cadena. Entonces para resolver este inconveniente, se puede hacer una conversión del tipo numérico al tipo cadena tal y como se muestra en el ejemplo anterior.

Supongamos que se tienen los siguientes registros:
Andrés
Arcia
21
2402992

amoret@ula.ve
Camilo
García
1
2440303
gcamilo@gmail.com

Si suministráramos la cadena "arcia" a nuestra función de búsqueda, entonces debería devolver ambos registros, pues la subcadena se encuentra en cada uno de ellos. Asegúrese de entender bien esta restricción.

Realiza un programa en C donde mediante el uso de un archivo secuencial busques un dato específico sobre algún tema de tu elección, por ejemplo: un archivo de datos que contenga un listado de modelos de automóviles, donde se busque el modelo Jetta

Resultados

La función buscar devuelve un arreglo de registros que cumplen con la cadena solicitada. La cadena solicitada debe ser buscada exahustivamente en todo el registro.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

PRÁCTICA 5. Acceso directo

Introducción

Con el uso del acceso directo se puede acceder de forma directa a la posición que se requiera sin tener que pasar por las posiciones anteriores, como si fueran arreglos en la memoria. Lo cual permitirá tomar modifica e insertar información en la posición que se desee. Las operaciones de lectura y/o escritura pueden hacerse en cualquier punto del archivo.

Muchas veces se desea acceder un dato concreto de un archivo en una posición relativa conocida de antemano. Por ejemplo, queremos leer o escribir en el registro 713 de un archivo. El acceso directo está pensado para resolver fácilmente estas operaciones. Un archivo de acceso directo permite leer o escribir bloques de información en un punto cualquiera del archivo, que se pasa como parámetro en la llamada del sistema correspondiente. Hay total libertad en el orden en que se accede a los registros. Por ejemplo, el 15, luego 713, luego 9... este modo de acceso es muy útil en sistema de base de datos.

Objetivo

El alumno comprenda mejor el manejo de archivo de acceso directo.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Altas Directas

Aquí se presenta una rutina que inserta registros de productos en un archivo directo. Se tomará el mismo ejemplo del registro de producto que se usó en el archivo secuencial. En este caso se nombrará el archivo como "PRODUCTO.DIR" La primera ocasión que se intente insertar registros en un archivo, éste debe crearse; sin embargo, debe cuidarse no crear el archivo cada vez que se invoque esta rutina porque debe tenerse presente que si se crea un archivo existente, se pierde su contenido anterior.

```
void ALTA_DIRECTA(void)
{  int no_prod;  // Variable local para el numero de producto   clrscr();
   cout << "\n\rALTAS DE REGISTROS DE PRODUCTOS";
         alias=fopen("PRODUCTO.DIR","rb+");
         // Intenta abrir el archivo PRODUCTO.DIR
         // en modo de lectura/escritura
         if(alias==NULL)
          alias=fopen("PRODUCTO.DIR","wb");
         // Crea el archivo en caso de no existir
         cout << "\n\n\n\rNumero de producto: ";
         cin >> no_prod;
  dir_fisica=no_prod*sizeof(Registro);
         // Calculo de la dir. fisica
         fseek(alias,dir_fisica,SEEK_SET);
         //Posicionar el apuntador del archivo
  fread(&Registro,sizeof(Registro),1,alias);
         // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
         if(Registro.no_prod==no_prod)
         {
          cout << "\n\n\n\rRegistro duplicado !!!";
          fclose(alias);
          getch();
```

```
        return;
       }
  cout << "\n\rDescripcion: ";
       gets(Registro.descrip);
       cout << "\n\rCantidad   : ";
       cin >> Registro.cantidad;
       cout << "\n\rPrecio     : ";
       cin >> Registro.precio;
       do  {
                cout << "\n\rGarantia   : ";
                Registro.garantia=toupper(getche());
                }while(Registro.garantia!='S' && Registro.garantia!='N');
  Registro.no_prod=no_prod;
  fseek(alias,dir_fisica,SEEK_SET);
       //Posicionar el apuntador del archivo
       fwrite(&Registro,sizeof(Registro),1,alias);
       // Grabar el Registro completo
       fclose(alias);
       // Cierra el archivo
  cout << "\n\n\n\rProducto registrado !!!";
       cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
       getch();
  return;
 }
```

Consultas directas

Se analiza una rutina que busca un registro particular de un producto en un archivo directo. En este tipo de archivo no es necesario hacer el recorrido secuencial desde el primer registro almacenado, sino se calcula la dirección física del registro que se desea consultar y se posiciona el apuntador del archivo directamente.

```
void CONSULTA_DIRECTA(void)
{
 int no_prod;
 // Variable local para el numero de producto que desea consultar
 clrscr();
 cout << "\n\rCONSULTA DE REGISTROS DE PRODUCTOS";
 alias=fopen("PRODUCTO.DIR","rb");
 // Intenta abrir el archivo PRODUCTO.DIR
 // en modo de solo lectura
 if(alias==NULL)
 {
  cout << "\n\n\n\rNo existe el archivo !!!";
  cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
  getch();
  return;
 }
  cout << "\n\n\n\rNumero de producto: ";
  cin >> no_prod;
  dir_fisica=no_prod*sizeof(Registro);
  // Calculo de la dir. fisica
  fseek(alias,dir_fisica,SEEK_SET);
  //Posicionar el apuntador del archivo
  fread(&Registro,sizeof(Registro),1,alias);
  // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
  if(Registro.no_prod==no_prod)
  {
   cout << "\n\rNo Prod              Descripcion  Cantidad      Precio  Garantia";
```

```
        cout << "\n\r-------------------------------------------------------------------------";
        printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,Registro.cantidad,Registro.precio,Registro.garantia);
        }
        else
        {
         cout << "\n\n\n\rNo existe ese registro !!!";
         }
         fclose(alias);
         cout << "\n\n\n\n\r<<< Oprima cualquier tecla para continuar >>>";
         getch();
         return;
         }
```

Modificaciones directas

La forma de modificar el contenido de los campos de registros de un archivo depende mucho de la rutina de consulta, ya que es necesario localizar previamente el registro que se desea modificar, capturar los nuevos valores y posteriormente grabar el registro completo en la misma posición que se encontraba. Esto último es muy importante porque recuerde que cuando se termina de leer un registro del archivo, el apuntador se posiciona al inicio del siguiente registro, por lo que, antes de grabar el registro modificado, es necesario reposicionar el apuntador del archivo en la dirección correcta.

```
void MODIFICACION_DIRECTA(void)
{
 int no_prod; // Variable local para el numero de producto que desea modificar
 clrscr();
 cout << "\n\rMODIFICACION DE REGISTROS DE PRODUCTOS";
 alias=fopen("PRODUCTO.DIR","rb+");
 // Intenta abrir el archivo PRODUCTO.DIR
 // en modo de lectura/escritura
 if(alias==NULL)
 // Valida la existencia del archivo
 {
 cout << "\n\n\n\rNo existe el archivo !!!";
 cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
 getch();
 return;
 }
 cout << "\n\n\n\rNumero de producto: ";
 cin >> no_prod;
 dir_fisica=no_prod*sizeof(Registro);
 // Calculo de la dir. fisica
 fseek(alias,dir_fisica,SEEK_SET);
 //Posicionar el apuntador del archivo
 fread(&Registro,sizeof(Registro),1,alias);
 // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
 if(Registro.no_prod==no_prod)
 {
 cout << "\n\rNo Prod                Descripcion  Cantidad      Precio  Garantia";
 cout << "\n\r-------------------------------------------------------------------------";
 printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,Registro.cantidad,Registro.precio,Registro.garantia);
 cout << "\n\n\n\rAnote los nuevos datos ...";
 cout << "\n\rDescripcion: ";
 gets(Registro.descrip);
 cout << "\n\rCantidad   : ";
 cin >> Registro.cantidad;
 cout << "\n\rPrecio     : ";
```

```
cin >> Registro.precio;
do
{
 cout << "\n\rGarantia   : ";
 Registro.garantia=toupper(getche());
 }while(Registro.garantia!='S' && Registro.garantia!='N');
 // Es necesario reposicionar el apuntador del archivo al principio del
 // registro que desea modificar, ya que al leer un registro, el
// apuntador se posiciona en el registro siguiente
// La funcion ftell(alias) devuelve la posicion donde se encuentra el
// apuntador
fseek(alias,dir_fisica,SEEK_SET);
//Posicionar el apuntador del archivo
fwrite(&Registro,sizeof(Registro),1,alias);
// Graba el registro con
// los nuevos campos
fclose(alias);
// Cierra el archivo
cout << "\n\n\n\rRegistro modificado !!!";
cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
else
 {
  cout << "\n\n\n\rNo se encuentra ese registro !!!";
  }
fclose(alias);
// Cierra el archivo
cout << "\n\n\n\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
```

Bajas lógicas directas

Las bajas lógicas consisten en "marcar" los registros eliminados. En el ejemplo práctico que se muestra a continuación, el registro borrado se "limpia", dejando en blanco todos sus campos (colocando el valor cero en los campos numéricos y blancos en las cadenas o de tipo carácter). Para fines prácticos, la rutina de bajas lógicas se asemeja mucho a la rutina de modificaciones, sólo que en las bajas no se capturan los nuevos valores, sino se les asigna valores en blanco y se graba el registro completo en la misma posición en la que se encontraba. La diferencia entre las rutinas de bajas lógicas del archivo secuencial y el archivo directo radica en el mecanismo utilizado para localizar el registro que se desea eliminar.

```
void BAJA_LOGICA_DIRECTA(void)
{
 int no_prod; // Variable local para el numero de producto que desea eliminar
 char op;
 clrscr();
 cout << "\n\rBAJA LOGICA DE REGISTROS DE PRODUCTOS";
 alias=fopen("PRODUCTO.DIR","rb+");
 // Intenta abrir el archivo PRODUCTO.DIR
 // en modo de lectura/escritura
 if(alias==NULL)
 // Valida la existencia del archivo
 {
  cout << "\n\n\n\rNo existe el archivo !!!";
  cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
```

```
   getch();
   return;
   }
   cout << "\n\n\n\rNumero de producto: ";
   cin >> no_prod;
   dir_fisica=no_prod*sizeof(Registro);
   // Calculo de la dir. fisica
   fseek(alias,dir_fisica,SEEK_SET);
   //Posicionar el apuntador del archivo
   fread(&Registro,sizeof(Registro),1,alias);
   // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
   if(Registro.no_prod==no_prod)
   {
    cout << "\n\rNo Prod                Descripción  Cantidad      Precio  Garantía";
    cout << "\n\r-------------------------------------------------------------------------";
    printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c\n\n\n\n\r",Registro.no_prod,Regi
  stro.descrip,Registro.cantidad,Registro.precio,Registro.garantia);
    Registro.no_prod=0;
    strcpy(Registro.descrip,"");
    Registro.cantidad=0;
    Registro.precio=0.0;
    Registro.garantia=' ';
    // Es necesario reposicionar el apuntador del archivo al principio del
    // registro que desea modificar, ya que al leer un registro, el
    // apuntador se posiciona en el registro siguiente
    // La funcion ftell(alias) devuelve la posicion donde se encuentra el
    // apuntador
    do
    {
     cout << "\n\rEsta seguro que desea eliminar este registro? [S/N] --> ";
           op=toupper(getche());
           }while(op!='S' && op!='N');
           if(op=='S')
           {
            fseek(alias,dir_fisica,SEEK_SET);
            //Posicionar el apuntador del archivo
            fwrite(&Registro,sizeof(Registro),1,alias);
            // Graba el registro con
            // los nuevos campos
            cout << "\n\n\n\rRegistro eliminado logicamente !!!";
            }
            else
            {
             cout << "\n\n\n\rRegistro NO eliminado !!!";
             }
           fclose(alias);
           // Cierra el archivo
           cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
           getch();
           return;
   }
   else
   {
    cout << "\n\n\n\rNo se encuentra ese registro !!!";
  }
fclose(alias);
// Cierra el archivo
```

```
cout << "\n\n\n\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
```

Implemente todas las funciones en un programa, donde tenga un menú de altas, bajas, modificaciones, consultas, salir.

Resultados

Obtener un programa con el acceso directo implementado.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

PRÁCTICA 6. Búsqueda secuencial y directa

Introducción

La búsqueda, se puede decir que es la acción de recuperar datos o información, siendo una de las actividades que más aplicaciones tiene en los sistemas de información, más formalmente se puede definir como "La operación de búsqueda sobre una estructura de datos es aquella que permite localizar un nodo en particular si es que éste existe.

Los métodos de comparación de llaves (secuencial y binaria) son los más sencillos de implementar, ya que como su nombre se indica, se dedican a comparar el dato a buscar con cada uno de los registros almacenados en el archivo o en la base de datos, siendo una de sus principales desventajas el tiempo de espera, ya que cuando los registros son pequeños funcionan de forma eficiente, pero van deteriorando su desempeño conforme se incrementa el número de registros a comparar.
La utilización de los métodos por comparación de llaves depende en gran medida del número de elementos, esto es, a mayor número de elementos son mayores las comparaciones.

Objetivo

El alumno solucione problemas de búsquedas en archivos mediante la implementación de búsquedas secuenciales y directas.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Búsqueda secuencial.
Este tipo de búsqueda consiste en examinar, a partir del primer elemento y de uno en uno, hasta encontrar el dato buscado o bien llegar al final de la lista que puede estar almacenada en archivo o arreglo.

En este tipo de listas los elementos pueden o no estar clasificados, ya que se empieza a comparar de uno en uno los elementos de la lista y no importa su orden para realizar la búsqueda, salvo para el tiempo de ejecución. Si el elemento que se está buscando, se encuentra al inicio de la lista, este tiempo, sería muy corto, pero si se encuentra al final, va a tardar más y si el elemento que se desea buscar, no se encuentra en la lista, se hizo necesario, recorrer toda la lista, para darse cuenta que no está en ella.

Y si se le aumenta a esto, que el número de elementos en la lista puede ser del orden de cientos o miles, va a hacer mucho más tardado su ejecución. Esta búsqueda tiene la ventaja de tener una fácil programación de su algoritmo.

El pseudocódigo del algoritmo es el siguiente:

```
inicio
 i=0
 bandera = 0
 mientras i < n
 inicio
   si k [ i] = x
     inicio
       desplegar "búsqueda exitosa y el dato es" x
       desplegar "y se encuentra en la posición" i
       i=n
       bandera = 1
     fin
   else
     i=i+1
```

```
  fin
si bandera = 0
  desplegar "no se encontró el dato en la lista"
fin
```

Para este pseudocódigo, se considera que se tienen n nodos (conjunto de uno o varios campos) en una lista como:

n0, n1 , n2 ,....n n _1

Cuyas llaves son: K0, K1, K2,....K n _ 1, donde n es ~ 0 y x es el valor del dato que se está buscando.

Elabora un programa en lenguaje C, tomando como base el pseudocódigo presentado en este tema para implementar una búsqueda secuencial.

La búsqueda Binaria o por Bisección no representa mucha dificultad para la programación de su algoritmo y además, es muy rápida su ejecución.

Este algoritmo requiere que los elementos de la lista, sobre la que va a actuar, estén clasificados, ya sea en forma ascendente o descendente, cada elemento de la lista puede tener varios campos. La lista se considera que empieza a almacenar sus elementos en la posición cero.

Va a utilizarse tres apuntadores, uno en la primera posición de la lista que se le denominara LI, para efectos de la explicación, otro en la última conocida como LS y el que apunte en la parte central, el cual se obtiene de la suma de LS mas LI entre dos (LI + LS/ 2) y tomando la parte entera, el cual se le llamará M.

A diferencia de la Búsqueda Secuencial, aquí el número de comparaciones no se comporta en forma lineal, ya que procede a realizar los siguientes pasos:

- Dividir la lista en dos partes, al determinar el elemento central de dicha lista, con lo que se iniciará el apuntador M.
- Comparar el valor del elemento buscado con el central.
- Si resultan ser iguales, la búsqueda termina con éxito, indicando en qué posición se encontró y cuáles son los datos que están en esa posición.
- En el caso de no ser iguales, se redefinen la posición de alguno de los apuntadores de los extremos (LI o LS), dependiendo del valor del elemento central, sea mayor o menor que el buscado. Por ejemplo, suponiendo que la lista está clasificada en forma ascendente, se pueden presentar los siguientes casos:

  1. Que el elemento buscado sea mayor que el elemento central (apuntado por M), entonces se desechará la primera mitad de la lista ya que por estar clasificada en forma ascendente se sabe que ahí no va a estar el elemento, con lo que se moverá el apuntador de inicio que es LI a la posición siguiente a donde está el elemento central que esta apuntado por M. Procediendo a considerar esta nueva lista, que es la segunda mitad, como la nueva lista.
  2. Que el elemento buscado ahora sea menor que el elemento apuntador por M, entonces, se desechará la segunda mitad de la lista y el apuntador que se actualizará será entonces LS, que es el que apuntaba al último elemento y ahora se moverá a una posición inmediata anterior a donde está apuntando M.

En ambos casos se aplicarán nuevamente los puntos anteriores del método, desde el primer paso, hasta que el elemento sea encontrado o bien que la lista que vaya quedando como resultado, quede vacía.

El pseudocódigo del algoritmo es el siguiente:

```
inicio
        LI = 0
        LS = n – 1
        mientras LI  < LS
        inicio
          M = Parte Entera (LI + LS) /2)
          si dato< lista[M]
             LS = M -1
         en caso contrario si dato> lista [M]
             LI = M +1
         en caso contrario
          inicio
            desplegar "el elemento buscado esta en la posición", M
             break
          fin
       fin
fin
```

Dato, es el valor que se va a buscar en la lista; lista, es donde están los elementos clasificados, LI es el límite inferior, LS es el límite superior, M es el apuntador de en medio y N el número de elementos de la lista.

Realiza un programa en el lenguaje de programación C para implementar la búsqueda binaria, partiendo del pseudo código visto.

Resultados

Tener un programa que implemente búsqueda secuencia y binaria.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

UNIVERSIDAD DE GUADALAJARA
Centro Universitario de los Lagos
División de Estudios de la Biodiversidad e Innovación Tecnológica
Departamento de Ciencias Exactas y Tecnología

PRÁCTICA 7. Ordenamiento Shell

Introducción

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

Objetivo

El alumno analice y diseñe un programa en cualquier lenguaje (C++ o java) donde se aplique la función Shell.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Ordena subgrupos de elementos separados K unidades (respecto de su posición en el arreglo) del arreglo original. El valor K es llamado incremento.

Después de que los primeros K subgrupos han sido ordenados (generalmente utilizando INSERCION DIRECTA), se escoge un nuevo valor de K más pequeño, y el arreglo es de nuevo partido entre el nuevo conjunto de subgrupos. Cada uno de los subgrupos mayores es ordenado y el proceso se repite de nuevo con un valor más pequeño de K.

Eventualmente el valor de K llega a ser 1, de tal manera que el subgrupo consiste de todo el arreglo ya casi ordenado.

Al principio del proceso se escoge la secuencia de decrecimiento de incrementos; el último valor debe ser 1.

"Es como hacer un ordenamiento de burbuja pero comparando e intercambiando elementos."

Cuando el incremento toma un valor de 1, todos los elementos pasan a formar parte del subgrupo y se aplica inserción directa.

El método se basa en tomar como salto N/2 (siendo N el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

Ejemplo:

Para el arreglo a = [6, 1, 5, 2, 3, 4, 0]

Tenemos el siguiente recorrido:

Recorrido	Salto	Lista Ordenada	Intercambio
1	3	2,1,4,0,3,5,6	(6,2),(5,4),(6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	Ninguno
4	1	0,1,2,3,4,5,6	(4,2),(4,3)
5	1	0,1,2,3,4,5,6	Ninguno

```
public static void shellSort( int a[ ]) {
```

```
for( int gap = a.length / 2; gap > 0; gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) ){
for( int i = gap; i < a.length; i++ ){
int tmp = a[ i ];
int j;
for(j = i; j >= gap && tmp < a[ j - gap ] ; j -= gap ){
a[ j ] = a[ j - gap ];
}
```

Realiza un programa en el lenguaje de su elección para implementar el ordenamiento Shell, se deben de generar 1000 números aleatoriamente almacenarlos en un archivo, ordenarlos y guardarlos en un archivo nuevo.

Resultados

Un programa que tenga implementado el ordenamiento Shell.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

PRÁCTICA 8. Funciones Hash

Introducción

Una función hash puede definirse como una transformación de clave a dirección, ya que al aplicar una función hash a una clave resulta el número de cubeta en la cual puede estar el registro con dicha clave. La función debe transformar las claves para que la dirección resultante sea un número comprendido entre los posibles valores de las cubetas. Por ejemplo, si se tienen 10 000 cubetas enumeradas del 0 al 9999, las direcciones producidas por la función deberán ser valores comprendidos entre el 0 y el 9999. Si las claves fueran alfabéticas o alfanuméricas, primero deberán convertirse a numéricas, tratando de no perder información, para luego ser transformadas a una dirección. Es importante que la función distribuya homogéneamente las claves entre los números de cubetas disponibles. Las funciones módulo, cuadrado, plegamiento y truncamiento presentadas para búsqueda interna son válidas también para búsqueda externa. Otra función que puede utilizarse para el cálculo de direcciones es la de conversión de bases, aunque no proporciona mayor homogeneidad en la distribución. De todas las citadas, la función módulo es de las que ofrece mayor uniformidad.

Cuando a dos ítems se les asigna la misma ranura, debemos contar con un método sistemático para colocar el segundo ítem en la tabla hash. Este proceso se denomina **solución de colisiones**. Como dijimos anteriormente, si la función hash es perfecta, nunca se producirán colisiones. Sin embargo, como esto no suele ser posible, la solución de colisiones se convierte en una parte muy importante de la transformación de claves.

Un método para solucionar las colisiones examina la tabla hash e intenta encontrar otra ranura disponible para contener el ítem que causó la colisión. Una manera sencilla de hacerlo es comenzar en la posición del valor hash original y luego moverse secuencialmente a lo largo de las ranuras hasta encontrar la primera que esté vacía.

Tenga en cuenta que es posible que necesite volver a la primera ranura (circularmente) para cubrir toda la tabla hash. Este proceso de solución de colisiones se conoce como **direccionamiento abierto** ya que intenta encontrar la siguiente ranura o dirección disponible ("abierta") en la tabla hash.

Objetivo

El alumno desarrolle un programa donde se implemente las diferentes funciones Hash y donde se maneje una tabla Hash.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Conceptos de hashing

a) Clave: La clave contiene el valor que permite ubicar, mediante la función Hash, la posicióno registro que contiene el resto de información asociada. Normalmente la clave es el campo que identifica en forma única la información. Por ejemplo:
Cédula de Identidad
Registro de Información Fiscal
Placa o Matrícula de Vehículo

b) Función Hash: Es un algoritmo o transformación que, aplicado a la clave, devuelve la posición del destino en donde podemos ubicar o recuperar el elemento que contiene dicha clave. Normalmente consta de tres partes:

Transformación: Si la clave no es numérica se convierte en un número. Con frecuenciase utiliza el valor ASCII de cada carácter y luego se aplican operaciones matemáticas para obtener un número entero.

Generación: El número generado se procesa con un algoritmo que trata deuniformizar la distribución de las claves en el rango de direcciones.
Compresión: Se comprime el número obtenido multiplicándolo por un factor para adecuarlo a la capacidad de almacenamiento disponible. La función Hash debe definirse al momento de diseñar el sistema y su selección tiene gran incidencia en rendimiento del sistema. Una buena función Hash debe tener las siguientes características:
Sencilla, de manera que sea fácil de codificar y minimice el tiempo de cálculo.

Distribución Uniforme de las direcciones tratando que la generación distribuya en forma aleatoria las claves y evite agrupamientos.

La idea es seleccionar una función que permita obtener una distribución con el mayor grado de uniformidad posible para evitar colisiones.

El código implementado consiste en crear un archivo llamado **dispersion.txt** donde se guardan los respetivos registros en su posición de acuerdo a la clave generada por la función hashing(); que manipula los caracteres de la clave mediante codigo ASCII y devuelve un entero que sería el número de registro. Si al devolver la función hashing se repite (hay una colisión) el número de registro, este registro pasa a ser insertado en el archivo **colisiones.txt**

Otro punto importante es que cada registro escrito en el archivo siempre tiene un puntero (llamado SR) que guarda el número de registro del siguiente creando así una lista. Allí en la imagen podemos apreciar la estructura de cómo se guarda los registros en el archivo.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
using namespace std;

struct registro{
   int  nr;
   char clave[8];
   char data[8];
   int sr;
}r, a, s;

struct encabezado{
   int nrs;
}ed, ec;    /// encabezado de dispersion y colisiones

FILE *fdd, *fdc;
int lr, le;
int fh( char *clave )
{
   int nr = 0;
   for(int i=0; i<strlen(clave); i++)
   {
      nr += int(clave[i]);
   }
   nr = nr%10 + 1;
   return nr;
}
```

```
/********************** Funcion Escribir ************************/
void escribir()
{
  int  pos, x;
  char rpt;
  bool band;
  fdd = fopen("dispersion.dat", "w+");
  if(fdd==NULL){
    cout << " No se pudo crear dispersion.dat" << endl;
    return;
  }

  fdc = fopen("colisiones.dat", "w+");
  if(fdc==NULL){
    cout << " No se pudo crear colisiones.dat" << endl;
    return;
  }
  cout <<" LR = "<<lr<<" | LE = "<<le<<endl<<endl;

  /// Inicializando variables

  ed.nrs = 0;
  ec.nrs = 0;
  fwrite(&ed, le, 1, fdd);
  fwrite(&ec, le, 1, fdc);

  /// Ciclo para insertar varios registros
  do
  {
    fflush(stdin);
    cout << " Clave: "; gets(r.clave);
    cout << " Data : "; gets(r.data);
    fflush(stdin);
    r.sr = -1;
    r.nr = fh(r.clave);
    cout << " Clave = "<< r.nr << endl;
    pos = (r.nr-1)*lr + le;
    fseek(fdd, pos, 0);
    fread(&s, lr, 1, fdd);
    if(strcmp(s.clave, "")==0)
    {
      fseek(fdd, pos, 0);
      fwrite(&r, lr, 1, fdd);
      ++ed.nrs;
    }
    else
    {
      band = false;
```

```
        x = s.sr;
        /// entramos al archivo de colisiones
        while(x != -1)
        {
           band = true;
           pos = (x-1)*lr + le;
           fseek(fdc, pos, 0);
           fread(&s, lr, 1, fdc);
           a = s;
           x = s.sr;
        }
        /// Listas de en el archivo de colision
        if(band==true)
        {
           r.nr = ++ec.nrs;
           a.sr = r.nr;
           fseek(fdc, pos, 0);
           fwrite(&a, lr, 1, fdc);
           fseek(fdc, 0, 2);
           fwrite(&r, lr, 1, fdc);
        }
        if(band==false)
        {
           r.nr = ++ec.nrs;
           s.sr = r.nr;
           fseek(fdd, pos, 0);
           fwrite(&s, lr, 1, fdd);
           fseek(fdc, 0, 2);
           fwrite(&r, lr, 1, fdc);
        }
     }
     cout << " Mas registros [s/n] : "; cin >> rpt; cout << endl;
   }while(rpt!='n');
   fseek(fdd, 0, 0);
   fwrite(&ed, le, 1, fdd);  /// actualizamos la cabezera del archivo DISPERSION
   fseek(fdc, 0, 0);
   fwrite(&ec, le, 1, fdc);  /// actualizamos la cabezera del archivo COLISIONES
   fclose(fdd);
   fclose(fdc);
}

/********************** Funcion Leer **********************/
void leer()
{
   int  pos, x, nr;
   char v_clave[10];
   bool band;
   fdd = fopen("dispersion.dat", "r");
   if(fdd==NULL){
```

```
    cout << " No se pudo abrir disperson.dat" << endl; return;
  }
  fdc = fopen("colisiones.dat", "r");
  if(fdc==NULL){
    cout << " No se pudo abrir colisiones.dat " << endl; return;
  }
  fflush(stdin);
  cout << " Ingrese clave : "; gets(v_clave);
  nr = fh(v_clave);
  pos = (nr-1)*lr + le;
  fseek(fdd, pos, 0);
  fread(&r, lr, 1, fdd);
  if(strcmp(v_clave, r.clave)==0)
  {
    cout << " Su data es: " << r.data << endl;
    return;
  }
  else
  {
    band = false;
    x = r.sr;
    while( x != -1 )
    {
      pos = (x-1)*lr + le;
      fseek(fdc, pos, 0);
      fread(&r, lr, 1, fdc);
      if(strcmp(v_clave, r.clave)==0)
      {
        band = true;
        cout << " Su data es : " << r.data << endl;
        break;
      }
    }
    if(band == false)
    {
      cout << " No esta registrado..!" << endl;
    }
  }
  fclose(fdd);
  fclose(fdc);
}

void menu()
{
  cout << "\t\t METODO DE DISPERSION - HASHING \n\n";
  cout << "\t 1. Escribir          \n";
  cout << "\t 2. Leer              \n";
  cout << "\t 3. Salir             \n";
  cout << "\t >> Ingrese opcion:  ";
```

```
}

/********************** Funcion Principal **********************/
int main()
{
  int op;
  lr = sizeof(struct registro);
  le = sizeof( struct encabezado);
  do
  {
    menu(); cin >> op;  cout << endl;
    switch(op)
    {
      case 1:
        escribir(); break;
      case 2:
        leer(); break;
      case 3:
        exit(0);
    }
    cout <<"\n\n ";
    system("pause"); system("cls");
  }while(op>0);
 return 0;
}
```

Haga los ajustes necesarios para implementar las diferentes funciones Hash.

Resultados

Obtener un programa donde se implementen las diferentes funciones Hash.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados

PRÁCTICA 9. Arboles binarios

Introducción

Un árbol AVL es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como mucho 1.

La denominación de árbol AVL viene dada por los creadores de tal estructura (Adelson-Velskii y Landis).

Recordamos que un árbol binario de búsqueda es un árbol binario en el cual cada nodo cumple con que todos los nodos de su subárbol izquierdo son menores que la raíz y todos los nodos del subárbol derecho son mayores que la raíz.

Recordamos también que el tiempo de las operaciones sobre un árbol binario de búsqueda son O(log n) promedio, pero el peor caso es O(n), donde n es el número de elementos.

La propiedad de equilibrio que debe cumplir un árbol para ser AVL asegura que la profundidad del árbol sea O(log(n)), por lo que las operaciones sobre estas estructuras no deberán recorrer mucho para hallar el elemento deseado. Como se verá, el tiempo de ejecución de las operaciones sobre estos árboles es, a lo sumo O(log(n)) en el peor caso, donde n es la cantidad de elementos del árbol.

Sin embargo, y como era de esperarse, esta misma propiedad de equilibrio de los árboles AVL implica una dificultad a la hora de insertar o eliminar elementos: estas operaciones pueden no conservar dicha propiedad.

Objetivo

El alumno aplique los conocimientos de los arboles AVL y sus operaciones.

Material y equipo

- ✓ Computadora.
- ✓ Compilador C o el compilador que el estudiante elija.

Desarrollo de la práctica

Declaración del tipo de dato

Iremos ya declarando el tipo de dato que representará un árbol AVL. Esto nos ayudará a formalizar las cosas y nos permitirá en el correr de este documento ir definiendo las operaciones sobre el tipo de dato abstracto.

Algunas consideraciones sobre la implementación del tipo de dato abstracto

- Las declaraciones que se listarán a continuación no tienen porqué tomarse al pie de la letra. Cada programador tendrá su estilo y su forma de resolver sus problemas.
- Las declaraciones que se listarán a continuación no tienen porqué tomarse al pie de la letra. Cada programador tendrá su estilo y su forma de resolver sus problemas.
- Como se podrá ver en el siguiente listado, la única diferencia de los nodos de un árbol AVL con los de un árbol binario común es la variable altura en la estructura nodo.
- Los nodos de un árbol pueden almacenar cualquier tipo de dato, arbitrariamente complejo. En este documento, por razones de simplicidad se usará el tipo de dato más simple que soporte
- comparaciones, o sea los enteros (tipo int de Ansi C). En el caso de que los datos almacenados en cada nodo sean más complicados (por ejemplo estructuras) o sean dinámicamente almacenados en memoria, algunas funciones deberán adaptarse para manejarlos. Por ejemplo, se deberá pasar como parámetros funciones de comparación, equivalencia, y de liberación de memoria.

A continuación, se lista la declaración del tipo abstracto de dato Árbol AVL:

```
typedef struct AVLNode AVLTree;

struct AVLNode
{
 int dato;
 AVLTree izq;
 AVLTree der;
 int altura;
};
```

Por cuestiones de simplicidad, la información almacenada en cada nodo del árbol será un entero.

Cada nodo tendrá almacenada su propia altura con respecto a la raíz absoluta del árbol con el que estamos trabajando.

A continuación declaramos las operaciones básicas sobre árboles binarios y con las cuales trabajaremos para acceder al tipo abstracto de dato Árbol AVL de aquí en más.

Si se usa algún lenguaje orientado a objetos como C++ o java y ya se tienen clases como árboles binarios o árboles binarios de búsqueda, conviene declarar los árboles AVL como una subclase de alguna de estas. Luego, las operaciones declaradas a continuación se heredarán de estos tipos.

```
/* Constructores */
AVLTree *vacio (void);
/* devuelve un árbol AVL vacío */
AVLTree *hacer (int x, AVLTree * izq, AVLTree * der);
/* devuelve un nuevo árbol formado por una raíz con valor x, subárbol izquierdo el árbol izq y subárbol derecho el
árbol der. */
/* Predicados */
bool es_vacio (AVLTree * t);
/* devuelve true sii. t es un árbol vacío. */
/* Selectores */
AVLTree *izquierdo (AVLTree * t);
/* devuelve el subárbol izquierdo de t. */
AVLTree *derecho (AVLTree * t);
/* devuelve el subárbol derecho de t. */
int raiz (AVLTree * t);
/* devuelve el valor de la raíz del árbol t. Precondición: !es_vacio(t) */
int altura (AVLTree * t);
/* devuelve la altura del nodo t en el árbol */
/* Destructures */
void destruir (AVLTree * t, void (*free_dato) (int));
/* libera la memoria ocupada por los nodos del árbol. Si los datos almacenados en cada nodo están almacenados
dinámicamente y se los quiere liberar también, debe pasarse como segundo parámetro una función de tipo void
func(int t) que libere la memoria de objetos int. Si los datos no están almacenados dinámicamente o simplemente
no se los quiere destruir (liberar de memoria), pásese como segundo parámetro NULL. Nota: Función Recursiva! */
```

Como se ha podido apreciar en el segmento de código anterior, se ha tratado de usar, en lo posible, el lenguaje español tanto para los comentarios como para los identificadores de variables y funciones. Sin embargo, esto se hace sólo con motivo de ser coherentes con el documento y el autor recomienda a los lectores programadores que en sus programas utilicen el lenguaje inglés para nombrar los identificadores.

Implementación de la rotación simple:

```
void rotar_s (AVLTree ** t, bool izq);
```

```
/* realiza una rotación simple del árbol t el cual se pasa por referencia. La rotación será izquierda sii. (izq==true) o
será derecha sii. (izq==false).
Nota: las alturas de t y sus subárboles serán actualizadas dentro de esta función
Precondición: si (izq==true) ==> !es_vacio(izquierdo(t)) si (izq==false) ==> !es_vacio(derecho(t))
*/

void rotar_s (AVLTree ** t, bool izq)
{
 AVLTree *t1;
 if (izq) /* rotación izquierda */
        {
         t1 = izquierdo (*t);
        (*t)->izq = derecho (t1); t1->der = *t;
        }
 else /* rotación derecha */
 {
        t1 = derecho (*t);
        (*t)->der = izquierdo (t1);
        t1->izq = *t;
 }
/* actualizamos las alturas de ambos nodos modificados */
actualizar_altura (*t);
actualizar_altura (t1);
/* asignamos nueva raíz */
*t = t1;
}
```

Declaración de la función rotar_s(). Esta declaración y el siguiente comentario deberían ir en el archivo de cabecera, interfaz del tipo abstracto árbol AVL con el usuario.
Un breve comentario que explica lo que hace la función, los parámetros que acepta y las precondiciones que éstos deben cumplir para que la función se ejecute correctamente.
Como el programador de C más experimentado puede ver, el paso a la función es el paso por referencia clásico en C. Lo que se pasa no es un puntero a la raíz del árbol sino la dirección de dicho puntero. De esta manera, dentro de la función podremos cambiar la misma raíz si es necesario (lo que justamente hacemos en las rotaciones).
También se acepta como segundo parámetro un valor booleano que determina si la rotación simple a efectuar sobre el árbol es izquierda o derecha.

Rotación doble derecha

```
void rotar_d (AVLTree ** t, bool izq);
/* realiza una rotación doble. Funciona análogamente a rotar_s(). */
void rotar_d (AVLTree ** t, bool izq)
{
 if (izq) /* rotación izquierda */
{
 rotar_s (&(*t)->izq, false);
 rotar_s (t, true);
}
else /* rotación derecha */
{
 rotar_s (&(*t)->der, true);
 rotar_s (t, false);
}
```

```
/* la actualización de las alturas se realiza en las rotaciones simples */
}
```

Declaración de la función rotar_d(); debería ir en un archivo de cabecera.
Los parámetros de rotar_d() son análogos a los de rotar_s()

Balance del árbol

```
void balancear (AVLTree ** t);
/* Detecta y corrige por medio de un número finito de rotaciones un desequilibrio en el árbol *t. Dicho
desequilibrio no debe tener una diferencia de alturas de más de 2. */
void balancear (AVLTree ** t) { if(!es_vacio(*t))
{
 if (altura (izquierdo (*t)) - altura (derecho (*t)) == 2)
 { /* desequilibrio hacia la izquierda! */
 if (altura ((*t)->izq->izq) >= altura ((*t)->izq->der))
  /* desequilibrio simple hacia la izquierda */
  rotar_s (t, true);
 else
  /* desequilibrio doble hacia la izquierda */
  rotar_d (t, true);
 }
 else if (altura (derecho (*t)) - altura (izquierdo (*t)) == 2)
 {
 /* desequilibrio hacia la derecha! */
 if (altura ((*t)->der->der) >= altura ((*t)->der->izq))
 /* desequilibrio simple hacia la izquierda */
  rotar_s (t, false);
 else
  /* desequilibrio doble hacia la izquierda */
  rotar_d (t, false);
 }
}
}
```

Declaración de la función balancear(). Esta declaración junto con el comentario que le sigue deberían estar en un archivo de cabecera usado para la interfaz del tipo abstracto de dato árbol avl con el usuario-programador.
Como dice en el comentario de la función, sólo se contemplarán aquellos desequilibrios cuya diferencia entre alturas es hasta 2.
Sabiendo que en el nodo al que apunta *t hay un desequilibrio hacia la izquierda (de diferencia de alturas 2), debemos averiguar qué clase de rotación aplicar. En Figure 12 se explica gráficamente a dónde apuntan las variables de la función en un árbol genérico.
Como puede verse en el código, nos decidimos por una rotación simple izquierda si el subárbol más pesado de (*t)->izq es el izquierdo o por una rotación doble izquierda si el subárbol más pesado de (*t)->izq es el derecho.
Si detectamos un desequilibrio hacia la derecha, la toma de deciciones son análogas a las de un desequilibrio hacia la izquierda, las cuales ya explicamos.

Inserción

```
void insertar (AVLTree ** t, int x);
/* inserta x en el árbol en un tiempo O(log(n)) peor caso. */
```

```
void insertar (AVLTree ** t, int x)
{
 if (es_vacio (*t))
        *t = hacer (x, vacio (), vacio ());
 /* altura actualizada automáticamente */
 else
 {
  if (x < raiz (*t))
    insertar (&(*t)->izq, x);
  else
    insertar (&(*t)->der, x);
  balancear (t);
  actualizar_altura (*t);
 }
}
```

Eliminación

```
void eliminar (AVLTree ** t, int x);
/* elimina x del árbol en un tiempo O(log(n)) peor caso. Precondición: existe un nodo con valor x en el árbol t. */
int eliminar_min (AVLTree ** t);
/* Función auxiliar a eliminar(). Elimina el menor nodo del árbol *t devolviendo su contenido (el cual no se libera de
memoria). Se actualizan las alturas de los nodos. Precondición: !es_vacio(*t) */

void eliminar (AVLTree ** t, int x)
 {
  AVLTree *aux;
  if (x < raiz (*t))
    eliminar (&(*t)->izq, x);
  else if (x > raiz (*t))
    eliminar (&(*t)->der, x);
  else /* coincidencia */
   {
    if (es_vacio (izquierdo (*t)) && es_vacio (derecho (*t)))
    {/* es una hoja */
          free (*t);
          (*t) = vacio();
    } else if (es_vacio (izquierdo (*t)))
          {/* subárbol izquierdo vacio */
            aux = (*t);
            (*t) = (*t)->der;
            free (aux);
          }
     else if (es_vacio (derecho (*t)))
          {/* subárbol derecho vacio */
            aux = (*t);
            (*t) = (*t)->izq;
            free (aux); }
          else /* caso más complicado */
           {
            (*t)->dato = eliminar_min (&(*t)->der);
       }
      }
    balancear (t); actualizar_altura (*t);
   }

 int eliminar_min (AVLTree ** t)
 {
```

```
 if (es_vacio (*t))
  {
   fprintf (stderr, "No se respeta precondición de eliminar_min()\n");
   exit(0);
  }
 else
  {
   if (!es_vacio (izquierdo (*t)))
    {
     int x = eliminar_min (&(*t)->izq);
     balancear (t);
     actualizar_altura (*t);
     return x;
    }
   else
    {
     AVLTree *aux = (*t);
     int x = raiz (aux);
     *t = derecho (*t);
     free (aux);
     balancear (t);
     actualizar_altura (*t);
     return x;
    }
  }
}
```

Con la información anterior integrarlas en un programa o desarrollarlo en el lenguaje de su elección donde se maneje la inserción, balanceo, eliminación y consulta en un árbol binario.

Resultados

Un programa con la implementación de un árbol binario.

Conclusiones

El alumno redactará la conclusión de esta práctica, lo que aprendió, que se le dificulto.

Bibliografía

Joyanes Aguilar, L. (2013). *Fundamentos generales de programación.* México: Mc Graw Hill.
Sznajdleder, P. A. (2014). *Algoritmos a fondo con implementaciones en C y Java.* Argentina: Alfaomega.
El alumno complementara la bibliografía con los recursos consultados