Análisis de Algoritmos 2016/2017

Práctica 3

Litzy Tatiana Rochas Ávila y Lucía Colmenarejo Pérez, 1201.

Código	Gráficas	Memoria	Total

1. Introducción y objetivos:

En esta tercera y última práctica hemos visto cómo diseñar y analizar el algoritmo de búsqueda sobre diccionarios. También ,o que hemos hecho ha sido determinar experimentalmente el tiempo medio de búsqueda de elementos en un diccionario que emplea como tipo de datos una tabla.

Lo primero que hemos hecho de todo ha sido implementar el TAD diccionario que está definido en busqueda.h. Para ello nos hemos servido de una estructura DICC utilizada para implementar in diccionario empleando una tabla, ya sea ordenada o desordenada. Esta estructura está compuesta por un entero *tamanio* (indica el tamaño de nuestra tabla), un entero *n_datos* (que es el número de datos de nuestra tabla), un tipo char llamado *orden* (nos indicará si nuestra tabla está ordenada o desordenada) y , por último, un array de enteros llamado *tabla* (será nuestra tabla de datos). Hemos de señalar que otra parte muy importante que hemos tenido que utilizar ha sido pfunc_busqueda que es un puntero a la función de búsqueda que corresponde con las declaraciones de búsqueda de bbin (es búsqueda binaria) , blin (es búsqueda lineal) y blin_auto (es búsqueda lineal auto-organizada).

En el primer ejercicio el objetivo a alcanzar era que teníamos que comprobar el correcto funcionamiento tanto de la búsqueda binaria como de búsqueda lineal sobre una tabla ordenada. Antes de llegar a esta comprobación, hemos tenido que implementar ocho funciones. La primera que implementamos fue ini_diccionario, la cual crea un diccionario vacío, sin datos, del tipo indicado por sus argumentos (tamanio indica el tamaño inicial que deseamos darle al diccionario y orden indica el tamaño usando las constantes ORDENADO Y NO_ORDENADO si empleamos una tabla ordenado o desordenada como estructura de datos). La segunda fue libera_diccionario que hemos hecho de tal forma que libere toda la memoria reservada por las rutinas del TAD diccionario. En tercer lugar, hicimos inserta_diccionario, que introduce el elemento clave en la posición clave del diccionario definido por pdicc en función del valor que le hayamos dado al campo orden. Si el valor del campo orden es NO_ORDENADO se insertará al final de la tabla; si es ORDENADO, la rutina lo insertará en la posición adecuada de tal forma que preserve el orden de la tabla, es decir, lo insertará al final y lo irá comparando con los datos anteriores de tal manera que se acabará insertando en su posición correcta (sin la utilización de la función de la práctica anterior de InserSort). La siguiente función que programamos fue la de insercion_masiva_diccionario, la cual inserta todas las claves (que han sido pasadas como parámetro de la función mediante un array de enteros) haciendo sucesivas llamadas a la función de inserta_diccionario. En quinto lugar hicimos

una función llamada busca_diccionario, que como su propio nombre indica, buscará una clave en el diccionario definido por pdicc usando la rutina especificada por metodo(parametro que le pasamos como argumento) y devolverá la posición en la cual se encontraba la clave buscada. A continuación de esta función, hemos implementado una función que no se nos pedía, pero que hemos hecho para poder ver más claramente por consola cómo es la tabla que se nos ha creado y así poder comprobar más claramente que nos ha devuelto la posición de la clave correcta y el número de operaciones básicas. Siguiendo el guión que nos han dado, hemos implementado las tres rutinas que serán los tres algoritmos de búsqueda y parte principal de este ejercicio.

En primer lugar, se nos pidió que implementásemos el algoritmo de búsqueda binaria (bbin). En este algoritmo lo primero que hemos hecho ha sido comparar la clave que vamos a buscar con un elemento cualquiera de la tabla (en nuestro caso se trata del elemento medio de la tabla). Si el valor del elemento medio es mayor que el del elemento buscado se repite el procedimiento en la parte de la tabla que abarca desde el primer elemento hasta el elemento medio, en caso contrario se toma la parte de la tabla que va desde el elemento medio hasta el último. De esta manera hemos podido obtener subtablas cada vez más pequeñas hasta que hemos obtenido una tabla de un sólo elemento y, por tanto, indivisible. Ya habríamos encontrado nuestra clave, en el supuesto caso de que se encontrase dentro de la tabla. Si, por el contrario, no se trata de la clave a buscar, entonces determinamos que esa clave no se encuentra en nuestra tabla.

En segundo lugar, teníamos que implemetar el algoritmo de búsqueda lineal (blin). Este algoritmo se utiliza, generalmente, cuando la tabla está desordenada (campo orden tendría que valer NO_ORDENADO) . Este tipo de búsqueda consiste en ir comparando de una forma secuencial cada elemento de la tabla hasta llegar a la clave. En caso de que no esté la clave en la tabla dada, llegaría al final de la tabla sin obtener ingń resultado.

En tercer y úlimo lugar, se nos pide la realización de un algoritmo de busqueda llamado búsqueda lineal auto-organizada (blin_auto). Este método consiste en recorrer toda la tabla hasta encontrar la clave. Una vez encontrada la clave, realizamos un swap entre la propia clave y el elemento anterior a ella.

Las tres rutinas de búsqueda devolverán la posición de la clave encontrada en la tabla mediante el puntero ppos o si no se encontraba la clave en la tabla devolverán la constante NO_ENCONTRADO.

Hay que señalar que todas as funciones devuelven el número de Obs que hen sido empleadas o ERR en función de que su desarrollo fuera el deseado o no (exceptuando algunas funciones como ini_diccionario que devolverá NULL si ha ocurrido un error o bien el diccionario creado si no ha ocurrido ningún error).

Por último, para terminar la realización de este primer ejercicio de la práctica hemos tenido que modificar ejercicio1.c para que nos permtiese comprobar el correcto funcionamiento de los tres tipos de algoritmo de busque que hemos implementado. Esta modificación ha consistido prácticamente en llamar a la de ini_diccionario para crear un diccionario, pasándole nosotros el orden que queremos(si ORDENADO o NO_ORDENADO); y, posteriormente le asignamos a una variable(a la cual llamamos nob) el numero de operaciones básicas realizadas por la llamada a la función busca_diccionario, pasándole el metodo que queremos utilizar para buscar la clave en la tabla.

En el segundo ejercicio lo que hemos hecho ha sido implementar una serie de funciones para realizar las medidas de búsqueda desarrolladas en el ejercicio anterior. Para ello ha sido necesario añadir al fichero de tiempos.c (hecho en la práctica anterior) y su correspondiente declaración en el fichero tiempos.h de la función tiempo_medio_busqueda y la funcion genera_tiempos_busqueda .

La primera función sirve para automatizar la toma de tiempos. Esta rutina devolverá el valor de ERR en caso de error y OK en caso contrrio. Además, guardamos los resultados obtenidos en el fichero mediante la función que ya implementamos en la práctica anterior de guarda_tabla_tiempos. A esta función le pasamos como parámetros: el método de búsqueda a emplear, el orden que queremos, el fichero donde escribiremos los resultados, el número mínimo (num_min) y máximo (num_max) de claves a buscar, el incremento (incr), n_veces que es el número de veces que queremos buscar la clave(n_veces será igual a 1 la mayoría de las veces) y, por último, el generador. Para este último parámetro fue necesario realizar la inserción de dos funciones en busqueda.c llamadas : generador_claves_uniforme y generador_claves_potencial. La primera genera todas las claves de 1 a max de forma secuencial y si n_claves(parámetro pasado a la función junto con la tabla y el máximo) es igual al máximo tamaño de la tabla entonces se generan cada clave una unica vez. En cambio, la segunda genera siguiendo una distribucion aproximadamente potencial, siendo los valores más pequeños mucho más probables que los grandes (el valor 1 tiene una probabilidad del 50%, el dos del 17%, el tres el 9%, etc.).

El primer objetivo de este segundo ejercicio es la comparación del tiempo medio y las operaciones básicas entre las búsquedas lineal y binaria. Para el método de búsqueda lineal emplearemos diccionarios desordenados y en la búsqueda binaria utilizaremos diccionarios ordenados. Además debemos de buscar todas las claves contenidas en el diccionario una sola vez(n_veces =1) y la función generador de claves que debemos de emplear es la de generador_claves_uniforme.

El segundo objetivo de este apartado es hacer lo mismo que en el primer apartado pero con el metodo de búsqueda de búsqueda lineal auto-organizada. La gran diferencia es que la función generador que hemos de emplear es la de generador_claves_potencial. De este apartado se deberá comparar el tiempo medio y número de operaciones básicas medias para la búsqueda auto-organizada en diccionarios desordenados y la búsqueda binaria en diccionarios ordenados. Lo probaremos para diccionarios de 100 elementos cambiando el valor del parámetro n_veces al menos para los valores de 1, 10, 100, 1000 y 10000.

Por último no será necesario que cambiemos nada en ejercicio2.c ya que este fichero nos viene dado en el zip adjuntado en la práctica y no será necesario implementarlo.

2. Herramientas y metodología:

Aquí exponemos qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas hemos utilizado (Netbeans, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc) y qué metodologías de desarrollo y soluciones al problema planteado hemos empleado en cada apartado. Así como las pruebas que hemos realizado a los programas desarrollados.

2.1 Apartado 1:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución propuesta en este apartado es la implementación de cada una de las funciones explicadas y expuestas en la introducción. En donde hemos puesto especial atención ha sido en la implementación de las funciones de búsqueda del TAD Diccionario. En la función de búsqueda binaria nos hemos ayudado de un bucle while para el desarrollo del algoritmo. Dentro de este bucle lo que hemos hecho ha sido adjudicar a una variable el valor medio de la tabla y después hacer distintos casos en función de si el valor de esta nueva variable es igual, menor o mayor. Si resulta que es igual entonces asignamos a puntero ppos la posición de la variable ya que significará que hemos encontrado nuestra clave en el diccionario. Si es la clave menor que el valor del elemento que se encuentra en la variable, entonces establecemos como último valor de nuestra subtabla la variable menos uno(para posteriormente recórrelo de forma recursiva pero ahora sobre esta nueva tabla, más pequeña que la inicial). Y en el último caso, es decir, resulta que el valor de la clave es mayor que el del elemento que se encuentra en nuestra variable (la que habíamos asignado desde el principio y que es el elemento medio de la tabla), entonces lo que hacemos es crear una nueva subtabla (que posteriormente recorreremos de forma recursiva) donde el principio será el valor medio más uno. Para la implementación de la función de búsqueda lineal hemos empleado un bucle for, ya que el algoritmo es simplemente recorrer toda la tabla hasta que hayamos encontrado la clave. Por último, a la hora de programar la búsqueda lineal auto-organizada, hemos optado por la opción de emplear un bucle for para recorrer toda la tabla hasta llegar a la clave. En el momento en el que hayamos encontrado dicha clave lo que efectuamos es un swap entre la clave y el elemento inmediatamente anterior a él, exceptuando el caso en el que la clave se encontrase en la primera posición(no haríamos un swap con el anterior porque no hay ningún elemento que le preceda).

Finalmente para el desarrollo de este apartado lo que hemos tenido que hacer ha sido cambiar el ejercicio1, como ya apuntábamos en la introducción. Hemos tenido que añadir la sentencia en la cual creamos un diccionario llamando a la función de ini_diccionario. Posteriormente, utilizando una variable llamada nob (que previamente hemos declara como un entero) guardamos el número de operaciones básicas realizadas por el método el cual le pasemos como argumento a la función busca_diccionario. Si esta variable tiene un valor igual o superior a 0, el programa nos puede imprimir por pantalla cuál es la clave, la posición en la que se encuentra y el número de OBs realizadas (en caso de que esté la clave presente en la tabla); que la clave no ha sido encontrada en la tabla; o, por si acaso no se cumple ninguna de las dos anteriores (debido a que ha habido algún error), error al buscar la clave. Además hemos añadido un bucle for para que se nos muestre la pantalla cómo era la permutación de la tabla inicial y, posteriormente, otro for para la nueva tabla una vez hecho el algoritmo de búsqueda. Y, por último, liberamos la permutación y liberamos el diccionario para evitar las pérdidas de memoria.

2.2 Apartado 2:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos). Además hemos utilizado el programa Gnuplot para la realización de la gráfica de los tiempos mínimo, máximo y promedio de operaciones básicas y poder comparar esos valores con los valores teóricos, para cada una de las situaciones que se nos pide.

Hemos tenido que implementar en este ejercicio dos funciones en tiempo.c llamadas tiempo_medio_busqueda y la genera_tiempos_busqueda . La primera función se encarga de tomar los tiempos máximo, mínimo y medio de un algoritmo de búsqueda en una tabla dada. Para ello ha sido necesario que creasemos un diccionario (mediante la función ini_diccionario) y generásemos las permutaciones (mediante la llamada a la rutina de genera_perm). Después, insertamos dichas permutaciones en el diccionario mediante la función insercion_masiva_diccionario. Después llamamos a generador_claves_uniforme para que nos genere las claves (función explicada anteriormente en la introducción). A continuación entramos en un bucle for para llamar al método de búsqueda e introducimos el número de operaciones básicas en un array declarado previamente y hacemos las comprobaciones del número máximo y mínimo de OBs realizadas. Por último, calculamos el tiempo promedio y el tiempo medio de ordenación y asignamos a cada campo del puntero ptiempo el valor que le corresponde. La segunda función automatiza la toma de tiempos, que será distinto dependiendo del tipo de búsqueda que utilicemos. En la implementación de esta función tuvimos que reservar memoria para un puntero a tiempo que será el que nos guarde el tiempo que tarda en realizarse la búsqueda de la clave. Después utilizamos un bucle for para incrementar el tamaño de las permutaciones llamando a tiempo_medio_busqueda para que nos devuelva el tiempo de cada una de las permutaciones. Por último llamamos a una función que ya teníamos implementada de la práctica anterior que es la de guarda_tabla_tiempos para que imprima en un fichero los tiempos obtenidos. Para finalizar y como apunte, hemos tenido que cambiar ejercicio2.c porque hemos tenido qe eliminar el parámetro de n_claves y, por tanto, cambiar a 11 el número que comparamos con el argc.

Cabe destacar que en todos los apartados hemos utilizado las herramientas de valgrind y gcc para la detección de errores y el compilado de los programas. Además en ciertos apartados hemos tenido que emplear el debbuger para encontrar el punto en el cual se encuentra el error y así poder arreglarlo, y, además nos ha ayudado bastante a la comprensión del funcionamiento del algoritmo.

3.Código fuente:

Aquí ponemos el código fuente exclusivamente de las rutinas que hemos desarrollado nosotros en cada apartado.

Ejercicio 1:

En busqueda.c:

```
/**************************************************/
/* Funcion:ini_diccionario          Fecha:08/12/2016                  */
/* Autores:Lucia Colmenarejo Pérez y                                  */
/* Litzy Tatiana Rocha Avila                                          */
/* Función que crea un diccionario de un cierto                       */
/* tamaño ya sea ordenado o no ordenado                               */
/* Entrada:                                                 */
/* int tamanio:Tamaño de la tabla del diccionario                */
/* char orden: indica si la tabla es ordenada o no               */
/* Salida:                                                  */
/* PDICC: un diccionario                                              */
/**************************************************/

PDICC ini_diccionario (int tamanio, char orden){

        PDICC newdiccionario= NULL;
        if(tamanio>MAX_TAMANIO) return NULL;
        newdiccionario = (PDICC)malloc(sizeof(DICC));;
        if(! newdiccionario) return NULL;
        newdiccionario->n_datos = 0;
        newdiccionario->tamanio = tamanio;
        newdiccionario->orden = orden;
        newdiccionario->tabla = (int *)malloc(sizeof(int)*newdiccionario->tamanio);
        if(!newdiccionario->tabla){
   libera_diccionario(newdiccionario);
   return NULL;
        }

        return newdiccionario;
}
```

```
/*************************************************/
/* Funcion:libera_diccionario      Fecha:08/12/2016                */
/* Autores:Lucia Colmenarejo Pérez y                               */
/* Litzy Tatiana Rocha Avila                                       */
/* Función que libera la tabla de un diccionario                   */
/* Entrada:                                                        */
/* PDICC: el diccionario a liberar                                 */
/* Salida:                                                         */
/* No tiene (void)                                                 */
/*************************************************/

void libera_diccionario(PDICC pdicc){
        if(!pdicc) return;
        if(pdicc->tabla!=NULL){
        free(pdicc->tabla);
        }
        free(pdicc);

}
/*************************************************/
/* Funcion:inserta_diccionario     Fecha:08/12/2016        */
/* Autores:Lucia Colmenarejo Pérez y                               */
/* Litzy Tatiana Rocha Avila                                       */
/* Función que inserta el elemento clave en la                     */
/* posición correcta del diccionario                               */
/* Entrada:                                                        */
/* PDICC pdicc: diccionario donde introducimos la                  */
/* clave                                                           */
/* int clave: clave a insertar                                     */
/* Salida:                                                         */
/* int i: contador                                                 */
/*************************************************/


int inserta_diccionario(PDICC pdicc, int clave){
   int a,i,j;

   if(!pdicc || clave<0) return ERR;
```

```
    if(pdicc->orden==NO_ORDENADO){
            pdicc->tabla[pdicc->n_datos]=clave;
            pdicc->n_datos++;
    }
    else if(pdicc->orden==ORDENADO){
             pdicc->tabla[pdicc->n_datos]=clave;
             pdicc->n_datos++;
            a=pdicc->tabla[pdicc->n_datos-1];
            j=pdicc->n_datos -2;
            while(j>=0 && pdicc->tabla[j]>a){
            pdicc->tabla[j+1]=pdicc->tabla[j];
            j--;
            i++;
            }
    pdicc->tabla[j+1]=a;

    }

    return i;
}

/*****************************************************/
/*                                          Fecha: 08/12/2016*/
/* Funcion:inserta_masiva_diccionario                */
/* Autores:Lucia Colmenarejo Pérez y                 */
/* Litzy Tatiana Rocha Avila                         */
/* Función que inserta todas las claves en el        */
/* diccionario                                       */
/* Entrada:                                          */
/* PDICC pdicc: diccionario donde introducimos la    */
/* clave                                             */
/* int n_claves: número de claves a insertar         */
/* int *claves: claves a insertar                    */
/* Salida:                                           */
/* OK si inserta bien todas las claves, si no ERR    */
/*****************************************************/
```

```
int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves){
        int i;
   if(!pdicc || !claves|| n_claves<0 || n_claves>pdicc->tamanio|| n_claves+pdicc->n_datos>pdicc->tamanio){
        return ERR;
   }
        for(i=0;i<n_claves;i++){
        inserta_diccionario(pdicc, claves[i]);
        }
        return OK;
}

/***************************************************/
/* Funcion:busca_diccionario                Fecha: 08/12/2016 */
/* Autores:Lucia Colmenarejo Pérez y                */
/* Litzy Tatiana Rocha Avila                        */
/* Función que busca la clave en el diccionario     */
/* Entrada:                                         */
/* PDICC pdicc: diccionario donde introducimos la   */
/* clave                                            */
/* int clave: clave a buscar                        */
/* int *ppos: posición de la clave buscada          */
/* pfunc_busqueda metodo: metodo de búsqueda        */
/* Salida:                                          */
/* int i: número de OBs realizadas                  */
/***************************************************/

int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo){
   int i=-1;
   if(!pdicc || clave < 0 || !ppos ||!metodo){
     return ERR;
   }
   i=metodo(pdicc->tabla, 0,(pdicc->n_datos-1), clave, ppos);

   return i;
}
```

```
/************************************************/
/* Funcion:imprime_diccionario  Fecha: 08/12/2016      */
/* Autores:Lucia Colmenarejo Pérez y                   */
/* Litzy Tatiana Rocha Avila                           */
/* Función que imprime la tabla del diccionario        */
/* Entrada:                                            */
/* PDICC pdicc: diccionario a imprimir                 */
/* Salida:                                             */
/* No devuelve nada (void)                             */
/************************************************/

void imprime_diccionario(PDICC pdicc){
   int i;
   if(!pdicc) return;
   for(i=0;i<pdicc->n_datos;i++){
            printf("%d\n", pdicc->tabla[i]);
   }
}

/************************************************/
/* Funcion:bbin                          Fecha: 08/12/2016*/
/* Autores:Lucia Colmenarejo Pérez y                   */
/* Litzy Tatiana Rocha Avila                           */
/* Función que busca la clave en el diccionario        */
/* utilizando el algoritmo de búsqueda binaria         */
/* Entrada:                                            */
/* int *tabla: tabla donde buscar la clave             */
/* int P: primer elemento de la tabla                  */
/* int U: último elemento de la tabla                  */
/* int clave: clave a buscar                           */
/* int *ppos: posición de la clave buscada             */
/* Salida:                                             */
/* int i: número de OBs si está en la tabla            */
/* NO_ENCONTRADO si la clave no esta en la tabla       */
/************************************************/
```

```
int bbin(int *tabla,int P,int U,int clave,int *ppos){
        int m,i=0;
  if(!tabla|| P<0||U<0||clave<0||!ppos){
        return ERR;
  }
  while(P<=U){
        m=(P+U)/2;
        if(tabla[m]==clave){
        *ppos=m;
        i++;
        return i;
        }
        else if(clave<tabla[m]){
        U=m-1;
        }
        else {
        P=m+1;
        }
        i++;
  }
  return NO_ENCONTRADO;
}
/**************************************************/
/* Funcion:bbin                                   Fecha: 08/12/2016*/
/* Autores:Lucia Colmenarejo Pérez y                                */
/* Litzy Tatiana Rocha Avila                                        */
/* Función que busca la clave en el diccionario                     */
/* utilizando el algoritmo de búsqueda lineal                       */
/* Entrada:                                                         */
/* int *tabla: tabla donde buscar la clave                          */
/* int P: primer elemento de la tabla                               */
/* int U: último elemento de la tabla                               */
/* int clave: clave a buscar                                        */
/* int *ppos: posición de la clave buscada                          */
/* Salida:                                                          */
/* int j: número de OBs si se ha encontrado                         */
/* NO_ENCONTRADO si la clave no esta en la tabla                    */
/**************************************************/
```

```
int blin(int *tabla,int P,int U,int clave,int *ppos){
        int j=0, i;

   for(i=P;i<=U;i++, j++){
        if(tabla[i]==clave){
        *ppos=i;
        j++;
        return j;
        }

   }

   return NO_ENCONTRADO;
}

/***************************************************/
/* Funcion:bbin                          Fecha: 08/12/2016*/
/* Autores:Lucia Colmenarejo Pérez y                      */
/* Litzy Tatiana Rocha Avila                              */
/* Función que busca la clave en el diccionario           */
/* utilizando el algoritmo de búsqueda lineal             */
/* auto-organizada                                        */
/* Entrada:                                               */
/* int *tabla: tabla donde buscar la clave                */
/* int P: primer elemento de la tabla                     */
/* int U: último elemento de la tabla                     */
/* int clave: clave a buscar                              */
/* int *ppos: posición de la clave buscada                */
/* Salida:                                                */
/* int j: número de OBs si se ha encontrado               */
/* NO_ENCONTRADO si la clave no esta en la tabla          */
/***************************************************/

int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
   int i, j=0;
   int aux;
   if(!tabla||P<0||U<0||clave<0||!ppos){
```

```
            return ERR;
    }
    for(i=P; i<=U; i++,j++){
            if(tabla[i]==clave){
            j++;
            if(i==P){
                    *ppos=i;
            return j;
            }
            *ppos=(i-1);
            aux=tabla[i];
            tabla[i]=tabla[i-1];
            tabla[i-1]= aux;
            return j;
            }
    }
    return NO_ENCONTRADO;
}
```

En ejercicio1.c:

```
/*****************************************/
/* Programa: ejercicio1          Fecha: 09/12/2016      */
/* Autores: Litzy Tatiana Rocha Avila y          */
/* Lucia Colmenarejo Perez                       */
/* Programa que comprueba el funcionamiento de   */
/* la busqueda lineal                            */
/*                                               */
/* Entrada: Linea de comandos                    */
/*   -tamanio: numero elementos diccionario      */
/*   -clave:   clave a buscar                    */
/*                                               */
/* Salida: 0: OK, -1: ERR                        */
/*****************************************/

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

```
#include "ordenacion.h"
#include "busqueda.h"
#include "permutaciones.h"

int main(int argc, char** argv)
{
 int i, nob, pos=-1;
 unsigned int clave, tamanio;
 PDICC pdicc;
 int *perm;
 char algoritmo[64];
 int orden;
 srand(time(NULL));

 if (argc != 9) {
        fprintf(stderr, "Error en los parametros de entrada:\n\n");
        fprintf(stderr, "%s -tamanio <int> -clave <int> -algoritmo <string> -orden <int>\n", argv[0]);
        fprintf(stderr, "Donde:\n");
        fprintf(stderr, " -tamanio : numero elementos de la tabla.\n");
        fprintf(stderr, " -clave : clave a buscar.\n");
        fprintf(stderr, " -algoritmo : nombre del algoritmo de busqueda blin, bbin, blinauto.\n");
        fprintf(stderr, " -orden : orden del diccionario 0 para no ordenado, 1 para ordenado.\n");
        exit(-1);
 }

 printf("Practica numero 3, apartado 1\n");
 printf("Realizada por: Litzy Tatiana Rocha Avila y Lucia Colmenarejo Perez\n");
 printf("Grupo: 1201\n");

 /* comprueba la linea de comandos */
 for(i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-tamanio") == 0) {
        tamanio = atoi(argv[++i]);
        } else if (strcmp(argv[i], "-clave") == 0) {
        clave = atoi(argv[++i]);
        }else if (strcmp(argv[i], "-algoritmo") == 0) {
        strcpy(algoritmo, argv[++i]);
```

```
        }else if (strcmp(argv[i], "-orden") == 0) {
        orden = atoi(argv[++i]);
        } else {
        fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
        }
}

if(orden==0){
pdicc = ini_diccionario(tamanio,NO_ORDENADO);
 if (pdicc == NULL) {
        /* error */
        printf("Error: No se puede Iniciar el diccionario\n");
        exit(-1);
}
}
else if(orden==1){
        pdicc = ini_diccionario(tamanio,ORDENADO);
        if (pdicc == NULL) {
        /* error */
        printf("Error: No se puede Iniciar el diccionario\n");
        exit(-1);
        }
}
else{
        printf("Parametro -orden erroneo");
        exit(-1);
}



perm = genera_perm(tamanio);


if (perm == NULL) {
        /* error */
        printf("Error: No hay memoria\n");
        libera_diccionario(pdicc);
        exit(-1);
}
```

```
  nob = insercion_masiva_diccionario(pdicc, perm, tamanio);


  if (nob == ERR) {
          /* error */
          printf("Error: No se puede crear el diccionario memoria\n");
          free(perm);
          libera_diccionario(pdicc);
          exit(-1);
  }
  if(strcmp(algoritmo, "blin")==0){
  nob = busca_diccionario(pdicc,clave,&pos,blin);
  }
  else if(strcmp(algoritmo, "bbin")==0){
  nob = busca_diccionario(pdicc,clave,&pos,bbin);
  }
  else if(strcmp(algoritmo, "blinauto")==0){
          nob = busca_diccionario(pdicc,clave,&pos,blin_auto);
  }
  else{
          printf("Parametro -algoritmo erroneo");
          free(perm);
          libera_diccionario(pdicc);
          exit(-1);
  }
  if(nob >= 0) {
          printf("Clave %d encontrada en la posicion %d en %d op. basicas\n",clave,pos,nob);
  } else if (nob==NO_ENCONTRADO) {
          printf("La clave %d no se encontro en la tabla\n",clave);
  } else {
          printf("Error al buscar la clave %d\n",clave);
  }


  free(perm);
  libera_diccionario(pdicc);


  return 0;
}
```

Ejercicio 2:

En tiempos.c:

```
/**
 *
 * Descripcion: Implementacion de funciones de tiempo
 *
 * Fichero: tiempos.c
 * Autor: Carlos Aguirre Maeso
 * Version: 1.0
 * Fecha: 16-09-2016
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tiempos.h"
#include "ordenacion.h"
#include "permutaciones.h"
#include <limits.h>
/***************************************************/
/* Funcion:tiempo_medio_ordenacion Fecha:08/10/2016*/
/* Autores:Lucia Colmenarejo Pérez y               */
/* Litzy Tatiana Rocha Avila                       */
/* Función que calcula el tiempo que tarda un      */
/* algoritmo en ordenar una tabla                  */
/* Entrada:                                        */
/* pfunc_ordena metodo:Puntero a la función de     */
/* ordenación                                      */
/* int n_perms:Número de permutaciones a evaluar   */
/* int tamanio:Tamaño de las permutaciones         */
/* PTIEMPO ptiempo:Puntero a la estructura TIEMPO  */
/* definida en tiempos.h                           */
/* Salida:                                         */
/* OK en caso de que se el cálculo                 */
/* ERR en caso de error                            */
```

```
/***************************************************/
short tiempo_medio_ordenacion(pfunc_ordena metodo, int n_perms, int tamanio, PTIEMPO ptiempo, int algoritmo) {
        int ob[n_perms]; /*Array donde guardaremos el número de operaciones básicas para cada permutación*/
        int i = 0, k = 0;
        int **perm = NULL;
        int j;
        double medio = 0;
        int minimo = INT_MAX;
        int maximo = INT_MIN;
        double tiempo;
        clock_t t1, t2; /*Utilización del reloj para calcular el tiempo de ejecución*/

        if (!metodo || n_perms < 0 || tamanio < 0 || !ptiempo) return ERR; /*Comprobamos que nos pasan correctamente los argumentos de
entrada*/

        perm = genera_permutaciones(n_perms, tamanio); /*Generamos las permutaciones*/
        if (perm == NULL) return ERR;
        t1 = clock(); /*Tiempo antes de empezar la ordenación*/

        for (j = 0; j < n_perms; j++) {
        i = metodo(perm[j], 0, tamanio - 1, algoritmo); /*llamamos al algoritmo de ordenación*/
        if (i == ERR) {
        for (i = 0; i < n_perms; i++) {/*Liberamos la memoria utilizada en caso de error*/
        free(perm[i]);
        }
        free(perm);
        return ERR;
        }
        ob[j] = i; /*Introducimos el numero de operaciones básicas en nuestro array de ob*/
        if (ob[j] > maximo) { /*Comprobamos si el numero de ob realizadas para la permutación es mayor que el máximo actual*/
        maximo = ob[j];
        }
        if (ob[j] < minimo) { /*Comprobamos si el numero de ob realizadas para la permutación es menor que el mínimo actual*/
        minimo = ob[j];
        }
        }

        t2 = clock(); /*Tiempo tras terminar la ordenación*/
```

```
        tiempo = ((double) (t2 - t1)) / (n_perms * CLOCKS_PER_SEC); /*Tiempo promedio de ejecución*/

        for (i = 0; i < j; i++) {
        k = k + ob[i];
        }/*Sumamos los elementos de nuestro array de ob*/

        medio = k / j; /*Calculamos el tiempo medio de ob realizadas*/

        ptiempo->n_perms = n_perms;
        ptiempo->tamanio = tamanio;
        ptiempo->min_ob = minimo;
        ptiempo->max_ob = maximo;
        ptiempo->medio_ob = medio;
        ptiempo->tiempo = tiempo;

        for (i = 0; i < n_perms; i++) {/*Liberamos la memoria utilizada*/
        free(perm[i]);
        }
        free(perm);

        return OK;
}

/*************************************************/
/* Funcion:genera_tiempos_ordenación Fecha:08/10/16*/
/* Autores:Lucia Colmenarejo Pérez y             */
/* Litzy Tatiana Rocha Avila                     */
/* Función que calcula el tiempo que tarda un    */
/* algoritmo en ordenar permutaciones cuyo tamaño  */
/* esta definido entre dos numeros y se incrementa */
/* segun un valor establecido escribe el resultado */
/* en un archivo                                 */
/* Entradas:                                     */
/* pfunc_ordena metodo:Puntero a la función de   */
/* ordenación                                    */
/* char* fichero:Fichero donde escribir          */
/* int num_min:Número que delimita el tamaño de la */
/* permutación por abajo                         */
```

```
/* int num_max:Número que delimita el tamaño de la */
/* permutación por arriba                          */
/* int incr:Incremento del tamaño de la permutación*/
/* int n_perms:Número de permutaciones a evaluar   */
/* Salida:                                         */
/* OK en caso de que se realice el cálculo         */
/* ERR en caso de error                            */

/************************************************/
short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero, int num_min, int num_max, int incr, int n_perms, int algoritmo) {

        int i = 0, j = 0, k = 0, n = 0;
        PTIEMPO ptiempo;

        if (!metodo || !fichero || num_min < 0 || num_max < 0 || incr < 0 || n_perms < 0) return ERR; /*Comprobamos que nos pasan bien los argumentos de entrada*/

        n = ((num_max - num_min) / incr) + 1; /*Calculamos el numero de veces que se ejecutará tiempo_medio_ordenación*/

        ptiempo = (PTIEMPO) malloc(sizeof (TIEMPO) * n); /*Reservamos memoria para el puntero a TIEMPO*/
        if (!ptiempo) return ERR; /*Comprobamos que se reservó correctamente la memoria*/

        for (i = num_min, k = 0; i <= num_max; i = i + incr, k++) {/*Bucle que permite incrementar el tamaño de las permutaciones y
                                                                    llama a tiempo_medio_ordeacion para cada tamaño*/
        j = tiempo_medio_ordenacion(metodo, n_perms, i, ptiempo + k, algoritmo); /*Llamamos a tiempo_medio ordenación para calcular el tiempo*/
        if (j == ERR) {
        free(ptiempo);
        return ERR;
        }
        }
        if (guarda_tabla_tiempos(fichero, ptiempo, n) == ERR) return ERR; /*Llamamos a guarda_tabla_tiempos para que nos imprima los datos*/
        free(ptiempo); /*Liberamos la memoria utilizada*/

        return OK;
}

/************************************************/
/* Funcion: guarda_tabla_tiempos Fecha:13/10/2016  */
```

```
/* Autores:Lucia Colmenarejo Pérez y                    */
/* Litzy Tatiana Rocha Avila                    */
/* Funcion que escribe en un fichero los datos          */
/* de los tiempos de ordenación                 */
/*                                              */
/* Entrada:                                     */
/* char* fichero:fichero donde escribir         */
/* PTIEMPO tiempo:Puntero a la estructura TIEMPO   */
/* definida en tiempos.h                        */
/* int N:Numero de veces que se realizo la      */
/* ordenación                                   */
/* Salida:                                      */
/* OK en caso de que se realice la escritura    */
/* ERR en caso de error                         */

/*************************************************/
short guarda_tabla_tiempos(char* fichero, PTIEMPO ptiempo, int N) {
        int i;
        FILE* f = NULL;

        if (!fichero || !ptiempo || N < 0) return ERR; /*Comprobamos que nos pasan bien los argumentos de entrada*/

        f = fopen(fichero, "w"); /*Abrimos el fichero en modo escritura*/
        if (!f)return ERR; /*Comprobamos que se abrió correctamente*/

        for (i = 0; i < N; i++) { /*Bucle que imprime el tamaño de permutaciones y los tiempos del array de punteros a TIEMPO*/
        fprintf(f, "%d  %f  %f  %d  %d\n", (ptiempo + i)->tamanio, (ptiempo + i)->tiempo, (ptiempo + i)->medio_ob, (ptiempo + i)->min_ob,
(ptiempo + i)->max_ob);
        }

        fclose(f); /*Cerramos el fichero*/
        return OK;
}
/*************************************************/
/* Funcion:genera_tiempos_busqueda   Fecha:08/10/16*/
/* Autores:Lucia Colmenarejo Pérez y                    */
/* Litzy Tatiana Rocha Avila                    */
/* Función que calcula el tiempo que tarda un           */
/* algoritmo en buscar una clave en permutaciones  */
```

```
/* cuyo tamaño esta definido entre dos numeros y   */
/* se incrementa segun un valor establecido escribe*/
/* el resultado en un archivo                      */
/* Entradas:                                       */
/* pfunc_busqueda metodo:Puntero a la función de   */
/* busqueda                                        */
/* pfunc_generador_ claves generador: generador de */
/* claves                                          */
/* int orden: Nos indica si etá ordenada o no      */
/* char* fichero:Fichero donde escribir            */
/* int num_min:Número que delimita el tamaño de la */
/* permutación por abajo                           */
/* int num_max:Número que delimita el tamaño de la */
/* permutación por arriba                          */
/* int incr:Incremento del tamaño de la permutación*/
/* int n_veces:Número de permutaciones a evaluar   */
/* Salida:                                         */
/* OK en caso de que se realice el cálculo         */
/* ERR en caso de error                            */

/************************************************/

short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador, int orden, char* fichero, int num_min, int num_max,
        int incr, int n_veces) {
        int i = 0, j = 0, k = 0, n = 0;
        PTIEMPO ptiempo;

        if (!metodo || !generador || orden < 0 || !fichero || num_min < 0 || num_max < 0 || incr < 0 || n_veces < 0)return ERR;
        n = ((num_max - num_min) / incr) + 1; /*Calculamos el numero de veces que se ejecutará tiempo_medio_ordenación*/

        ptiempo = (PTIEMPO) malloc(sizeof (TIEMPO) * n); /*Reservamos memoria para el puntero a TIEMPO*/
        if (!ptiempo) return ERR; /*Comprobamos que se reservó correctamente la memoria*/

        for (i = num_min, k = 0; i <= num_max; i = i + incr, k++) {/*Bucle que permite incrementar el tamaño de las permutaciones y llama a
tiempo_medio_busqueda para cada tamaño*/
        j = tiempo_medio_busqueda(metodo, generador, orden, i, i, n_veces, (ptiempo + k)); /*Llamamos a tiempo_medio_busqueda para calcular
el tiempo*/
        if (j == ERR) {
        free(ptiempo);
```

```
            return ERR;
    }
            }

            if (guarda_tabla_tiempos(fichero, ptiempo, n) == ERR) return ERR; /*Llamamos a guarda_tabla_tiempos para que nos imprima los
datos*/

            free(ptiempo); /*Liberamos la memoria utilizada*/

            return OK;

}

/************************************************/
/* Funcion:tiempo_medio_busqueda   Fecha:08/10/2016*/
/* Autores:Lucia Colmenarejo Pérez y                        */
/* Litzy Tatiana Rocha Avila                        */
/* Función que calcula el tiempo que tarda un               */
/* algoritmo en buscar una clave en una tabla               */
/* Entrada:                                 */
/* pfunc_ordena metodo:Puntero a la función de              */
/* ordenación                                       */
/* pfunc_generador_ claves generador: generador de */
/* claves                                   */
/* int orden: Nos indica si etá ordenada o no               */
/* int tamanio:Tamaño de las permutaciones                  */
/* int n_claves: Número de claves a buscar                  */
/* int n_veces:Número de permutaciones a evaluar   */
/* PTIEMPO ptiempo:Puntero a la estructura TIEMPO  */
/* definida en tiempos.h                            */
/* Salida:                                  */
/* OK en caso de que se el cálculo                  */
/* ERR en caso de error                             */

/************************************************/

short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador, int orden, int tamanio, int n_claves, int n_veces,
            PTIEMPO ptiempo) {
```

```
int i = 0;
int *perm = NULL;
int *tabla = NULL;
int j;
int pos = -1;
long total =0, medio=0;
int minimo = INT_MAX;
int maximo = INT_MIN;
double tiempo;
PDICC diccionario = NULL;
clock_t t1, t2; /*Utilización del reloj para calcular el tiempo de ejecución*/



diccionario = ini_diccionario(tamanio, orden);
if (!diccionario) return ERR;
perm = genera_perm(n_claves); /*Generamos las permutaciones*/
if (!perm) return ERR;
if (insercion_masiva_diccionario(diccionario, perm, n_claves) == ERR) {
free(perm);
libera_diccionario(diccionario);
return ERR;
}

tabla = (int *) malloc(sizeof (int)*n_veces * n_claves);
if (!tabla) {
free(perm);
libera_diccionario(diccionario);
return ERR;
}

generador(tabla, n_claves*n_veces, n_claves);

t1 = clock(); /*Tiempo antes de empezar la ordenación*/

for (j = 0; j < n_claves*n_veces; j++) {
i = metodo(diccionario->tabla, 0, diccionario->n_datos -1, tabla[j], &pos); /*llamamos al algoritmo de ordenación*/
if (i == NO_ENCONTRADO) {
```

```
        free(perm); /*Liberamos la memoria utilizada en caso de error*/
        free(tabla);
        libera_diccionario(diccionario);
        return ERR;
        }


        total += i; /*Introducimos el numero de operaciones básicas en nuestro array de ob*/

        if (i > maximo) { /*Comprobamos si el numero de ob realizadas para la permutación es mayor que el máximo actual*/
        maximo = i;
        }
        if (i < minimo) { /*Comprobamos si el numero de ob realizadas para la permutación es menor que el mínimo actual*/
        minimo = i;
        }
        }

        t2 = clock(); /*Tiempo tras terminar la ordenación*/
        tiempo = ((double) (t2 - t1))*(double)1000 / (n_veces * n_claves* CLOCKS_PER_SEC); /*Tiempo promedio de ejecución*/


        medio = total / (n_veces*n_claves); /*Calculamos el tiempo medio de ob realizadas*/

        ptiempo->n_perms = n_veces;
        ptiempo->tamanio = tamanio;
        ptiempo->min_ob = minimo;
        ptiempo->max_ob = maximo;
        ptiempo->medio_ob = medio;
        ptiempo->tiempo = tiempo;
        ptiempo->n_veces = n_veces;
        free(perm); /*liberamos la memoria utilizada*/
        free(tabla);
        libera_diccionario(diccionario);
        return OK;
}
```

En ejercicio2.c:

```
/*****************************************************/
```

```
/* Programa: ejercicio2 Fecha:     09/12/2016              */
/* Autores:Litzy Tatiana Rocha Avila y                     */
/* Lucia Colmenarejo Perez                                 */
/* Programa que escribe en un fichero                      */
/* los tiempos medios del algoritmo de                     */
/* busqueda                                        */
/*                                                 */
/* Entrada: Linea de comandos                              */
/* -num_min: numero minimo de elementos de la tabla                */
/* -num_max: numero minimo de elementos de la tabla                */
/* -incr: incremento                               */
/* -fclaves: numero de claves a buscar                     */
/* -numP: Introduce el numero de permutaciones a promediar */
/* -fichSalida: Nombre del fichero de salida               */
/*                                                 */
/* Salida: 0 si hubo error                         */
/*              -1 en caso contrario                       */
/*********************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "permutaciones.h"
#include "busqueda.h"
#include "tiempos.h"

int main(int argc, char** argv)
{
  int i, num_min,num_max,incr,n_veces, orden;
  char nombre[256], algoritmo[64], generador[64];
  short ret;

  srand(time(NULL));

  if (argc != 17) {
                fprintf(stderr, "Error en los parametros de entrada:\n\n");
                fprintf(stderr, "%s -num_min <int> -num_max <int> -incr <int>\n", argv[0]);
```

```
		fprintf(stderr, "\t\t -n_veces <int> -algoritmo <string> -orden <int> \n");
		fprintf(stderr, "\t\t -generador <string> -fichSalida <string> \n");
		fprintf(stderr, "Donde:\n");
		fprintf(stderr, "-num_min: numero minimo de elementos de la tabla\n");
		fprintf(stderr, "-num_max: numero minimo de elementos de la tabla\n");
		fprintf(stderr, "-incr: incremento\n");
		fprintf(stderr, "-n_veces: numero de veces que se busca cada clave\n");
		fprintf(stderr, "-algoritmo : nombre del algoritmo de busqueda blin, bbin, blinauto.\n");
		fprintf(stderr, "-orden : orden del diccionario 0 para no ordenado, 1 para ordenado.\n");
		fprintf(stderr, "-generador: generador de claves uniforme, potencial \n");
		fprintf(stderr, "-fichSalida: Nombre del fichero de salida\n");
		exit(-1);
}

printf("Practica numero 3, apartado 2\n");
printf("Realizada por: Litzy Tatiana Rocha Avila y Lucia Colmenarejo Perez\n");
printf("Grupo: 1201\n");

/* comprueba la linea de comandos */
for(i = 1; i < argc ; i++) {
		if (strcmp(argv[i], "-num_min") == 0) {
		num_min = atoi(argv[++i]);
		} else if (strcmp(argv[i], "-num_max") == 0) {
		num_max = atoi(argv[++i]);
		} else if (strcmp(argv[i], "-incr") == 0) {
		incr = atoi(argv[++i]);
		} else if (strcmp(argv[i], "-n_veces") == 0) {
		n_veces = atoi(argv[++i]);
		} else if (strcmp(argv[i], "-algoritmo") == 0) {
		strcpy(algoritmo, argv[++i]);
		} else if (strcmp(argv[i], "-orden") == 0) {
		orden = atoi(argv[++i]);
		} else if (strcmp(argv[i], "-generador") == 0) {
		strcpy(generador, argv[++i]);
		} else if (strcmp(argv[i], "-fichSalida") == 0) {
		strcpy(nombre, argv[++i]);
		} else {
		fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
```

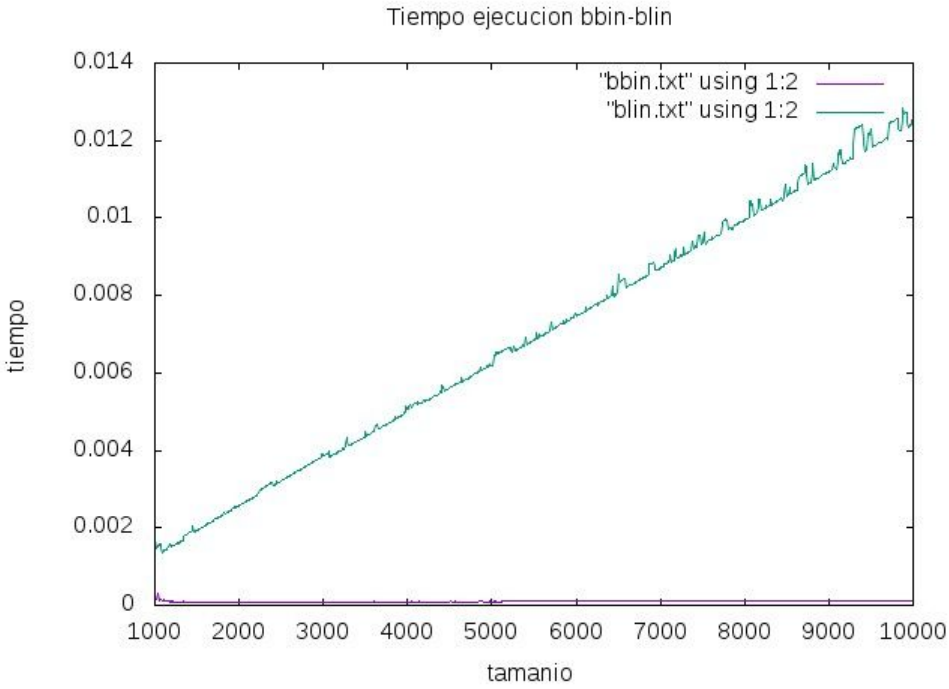
```
		exit(-1);
		}
}
if(strcmp(algoritmo, "blin")!=0 && strcmp(algoritmo, "bbin")!=0 && strcmp(algoritmo, "blinauto")!=0){
		printf("Parametro algoritmo erroneo\n");
		exit(-1);
}
if(strcmp(generador, "uniforme")!=0 && strcmp(generador, "potencial")!=0){
		printf("Parametro generador erroneo\n");
		exit(-1);
}
if(orden!=1 && orden !=0){
		printf("Parametro orden erroneo\n");
		exit(-1);
}

if(strcmp(algoritmo, "blin")==0){
		if(strcmp(generador, "uniforme")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(blin, generador_claves_uniforme, ORDENADO,nombre, num_min, num_max, incr, n_veces);
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(blin, generador_claves_uniforme, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
		else if(strcmp(generador, "potencial")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(blin, generador_claves_potencial, ORDENADO,nombre, num_min, num_max, incr, n_veces);
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(blin, generador_claves_potencial, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
}
else if(strcmp(algoritmo, "bbin")==0){
		if(strcmp(generador, "uniforme")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(bbin, generador_claves_uniforme, ORDENADO,nombre, num_min, num_max, incr, n_veces);
```

```
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(bbin, generador_claves_uniforme, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
		else if(strcmp(generador, "potencial")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(bbin, generador_claves_potencial, ORDENADO,nombre, num_min, num_max, incr, n_veces);
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(bbin, generador_claves_potencial, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
}
else if(strcmp(algoritmo, "blinauto")==0){
		if(strcmp(generador, "uniforme")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(blin_auto, generador_claves_uniforme, ORDENADO,nombre, num_min, num_max, incr, n_veces);
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(blin_auto, generador_claves_uniforme, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
		else if(strcmp(generador, "potencial")==0){
		if(orden==1){
		ret = genera_tiempos_busqueda(blin_auto, generador_claves_potencial, ORDENADO,nombre, num_min, num_max, incr, n_veces);
		}
		else if(orden==0){
		ret = genera_tiempos_busqueda(blin_auto, generador_claves_potencial, NO_ORDENADO ,nombre, num_min, num_max, incr, n_veces);
		}
		}
}
if (ret == ERR) {
		printf("Error en la funcion genera_tiempos_busqueda\n");
		exit(-1);
}
```

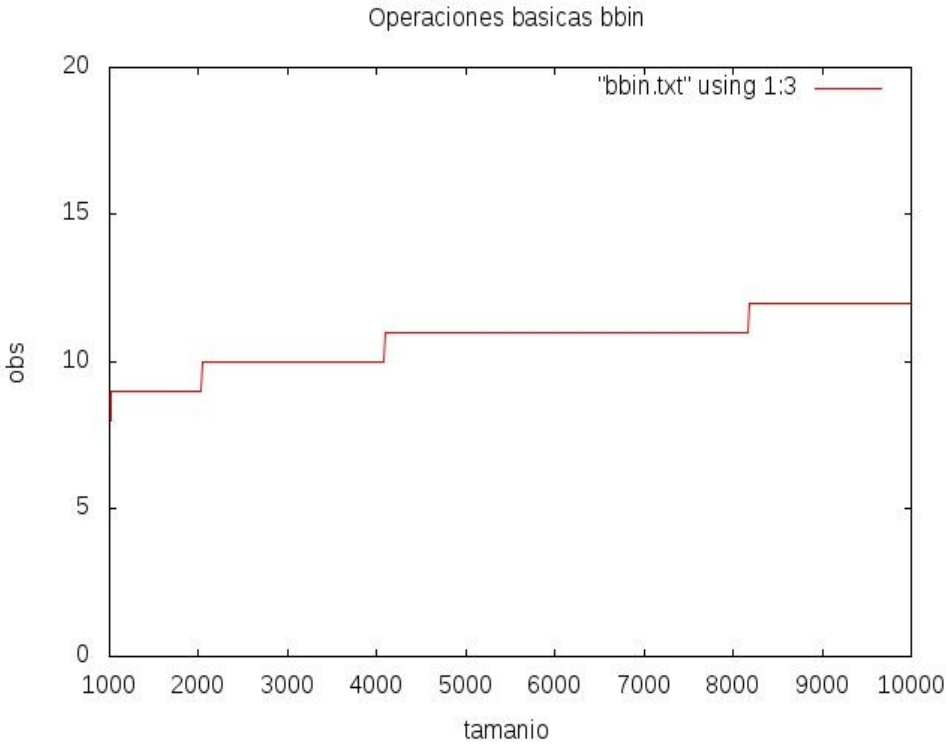
```
 printf("Salida correcta \n");


 return 0;
}
```

4.Resultados:

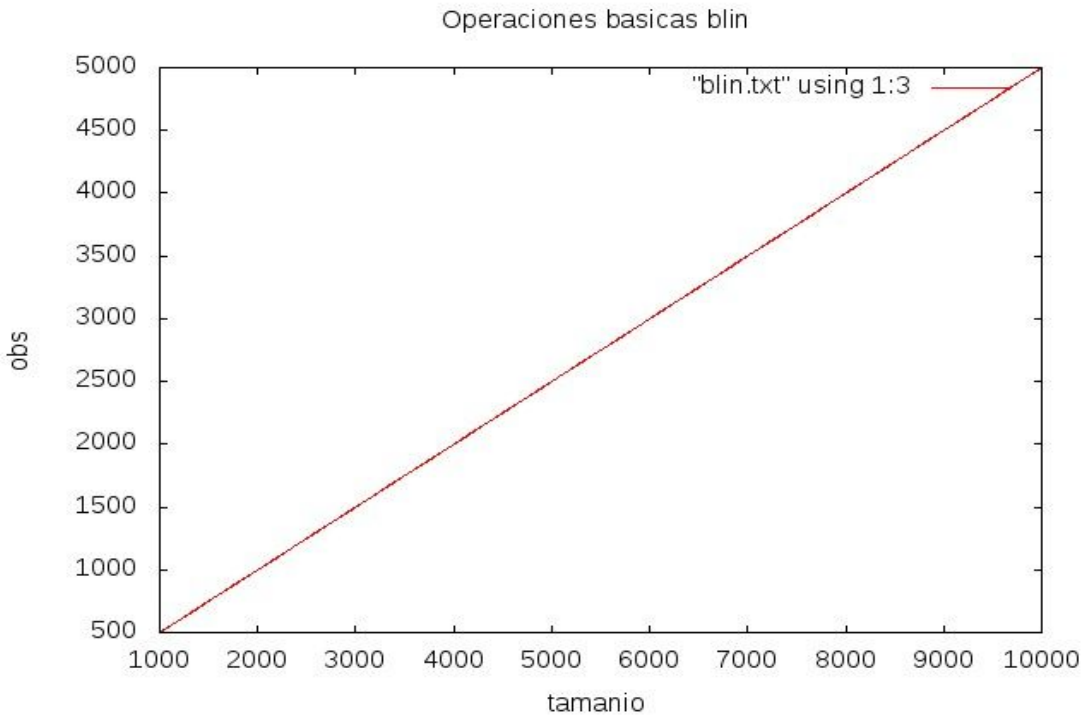
Apartado 1:



Se aprecia claramente la diferencia entre el algoritmo de búsqueda binaria ya que este algoritmo va dividiendo la tabla hasta encontrar la clave; mientras que el algoritmo de búsqueda lineal va recorriendo la tabla, por lo tanto, es lógico que tarde mucho más que el método de búsqueda binaria.

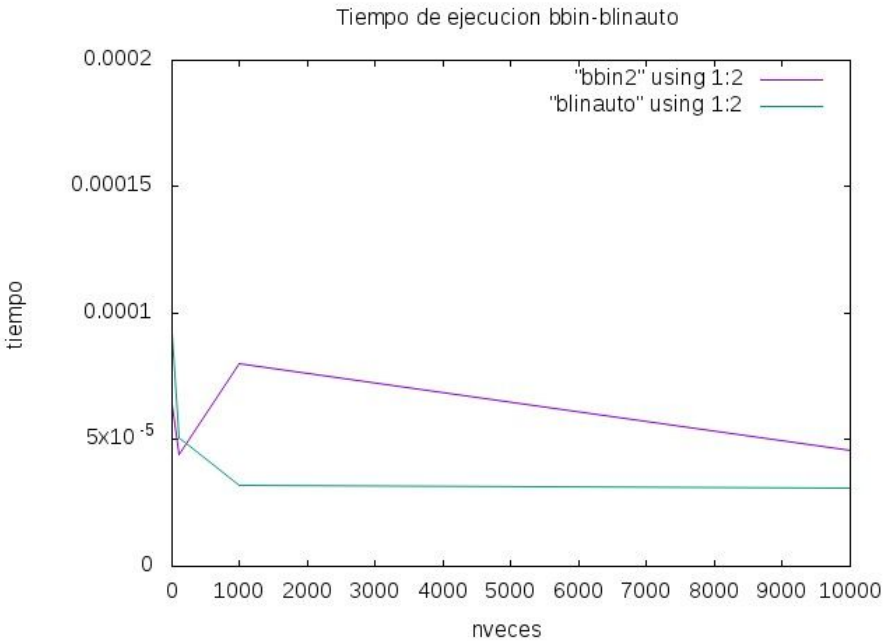


En esta gráfica podemos apreciar cómo las operaciones básicas es O(log(n)) para el método de búsqueda binaria.

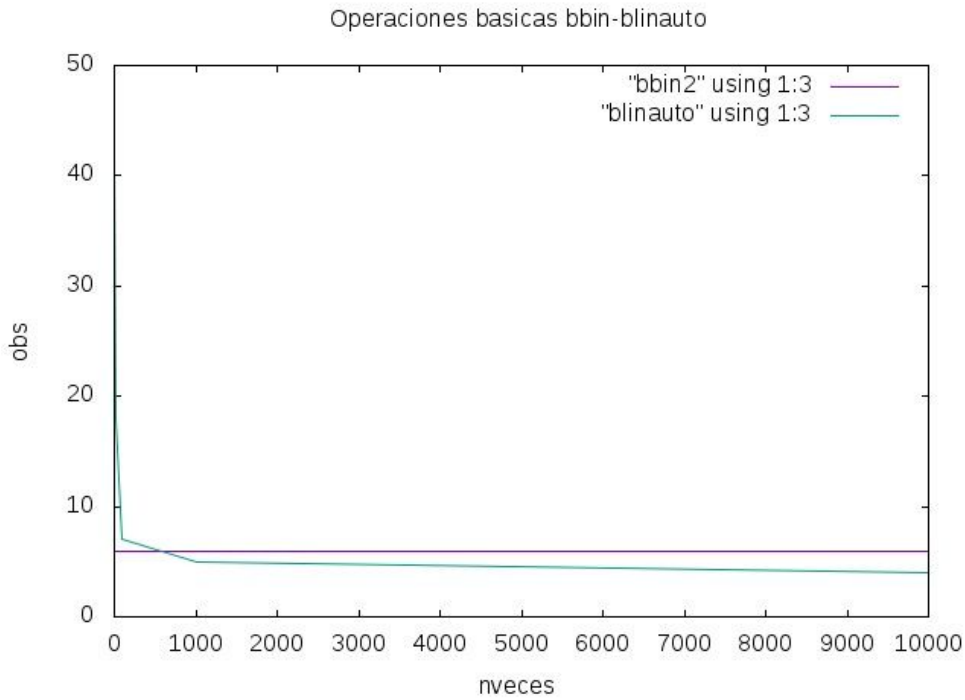


En esta gráfica podemos apreciar cómo las operaciones básicas es O(n) para el método de búsqueda lineal.

Apartado 2:



Podemos ver que los tiempos de ejecución de ambas son bastante cercanos, al contrario que en blin por lo que esto confirma la teoría de que blinauto es mejor.



El número de operaciones básicas en blinauto se mantiene constante, mientras que en bbin va disminuyendo el número de OBs.

5.Cuestiones:

1**. ¿Cuál es la operación básica de bbin, blin y blin_auto?**

La operación básica de la búsqueda binaria (bbin) es:

```
if(tabla[m]==clave){
        ….
}
else if(clave<tabla[m]){
        …
}
else{
        …
}
```

Aunque haya dos operaciones las contamos como una.

La operación básica de la búsqueda lineal (blin) es:

tabla[i]==clave

Esta se encuentra dentro del bucle que va recorriendo toda la tabla hasta que encuentra la clave(que es la comparación que realiza nuestra operación básica).

La operación básica de la búsqueda lineal auto-organizada es:

tabla[i]==clave

En este caso es la misma que en la búsqueda lineal porque en lo que se diferencian es que después de haber encontrado la clave, la búsqueda lineal lo que hace es devolverte el valor y la posición de la clave; mientras que la búsqueda lineal auto-organizada hace además un swap con el elemento anterior(salvo en el caso de que la clave se encuentre en la primera posición).

2**.Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor W SS (n) y el caso mejor B SS (n) de BBin y BLin. Utilizar la notación asintótica (O, Θ, o, Ω,etc) siempre que se pueda.**

BSS(n) de BLin = O(1) (caso en el que la clave se encuentra en la primera posición)

WSS(n) de BLin = O(n) (caso en el que la clave se encuentra en la última posición)

BSS(n) de BBin = O(1) (caso en el que la clave se encuentra en la posición del medio de la tabla)

WSS(n) de BBin = O(log(n)) (caso en el que la clave se encuentra en una de las dos posiciones de los extremos de la tabla, es decir, en una de sus esquinas)

3. **Cuando se utiliza blin auto y la distribución no uniforme dada ¿Cómo varía la posición de los elementos de la lista de claves según se van realizando más búsquedas?**

El algoritmo de blin auto consiste en buscar la clave en un tabla y un vez encontrada la cambia con el elemento que la precede(exceptuando que se tratase del primer elemento, que no se hace swap). De esta manera, lo que podemos determinar es que a medida que realicemos sucesivas búsquedas de claves sobre una misma tabla , lo que podemos observar es que las claves se van encontrando cada vez en una posición más cercana a la primera (encontraremos antes la clave).

4. **¿Cuál es el orden de ejecución medio de blin auto en función del tamaño de elementos en el diccionario n para el caso de claves con distribución no uniforme dado? Considerar que ya se ha realizado un elevado número de búsquedas y que la lista está en situación más o menos estable.**

Si consideramos un número elevado de búsquedas y que la lista está más o menos, entonces el orden de ejecución medio de blin auto en función del tamaño de elementos en el diccionario n para el caso de claves con distribución no uniforme es O(1) .

5. **Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el por qué busca bien) del algoritmo bbin.**

En la búsqueda binaria cada vez que hacemos una iteración (exceptuando el caso en el que el elemento se encuentra en la posición media de la tabla) el nuevo tamaño de la tabla es menor que la mitad que el previo. Dependiendo de si el valor es mayor o menor que el elemento medio de la tabla nos quedaremos con la subtabla izquierda o derecha y, así, sucesivamente. De tal forma que nos queda una iteración que sería:

$N \rightarrow N/2 \rightarrow N/2^2 \rightarrow N/2^3 \rightarrow \ldots\ldots\rightarrow N/2^k <= 1$ k=número máximo de divisiones por 2, es decir, número máximo de iteraciones.

Llegamos al punto en el que es menor o igual que 1 porque llega un momento en el que la tabla sólo tiene un elemento(que si coincide con la clave que buscamos ya hemos terminado el algoritmo) o tiene menos ya que la clave no ha sido encontrada en la tabla.

6.Conclusiones finales:

En esta práctica hemos reforzado nuestros conocimientos acerca de los algoritmos de búsqueda (búsqueda lineal, búsqueda binaria y búsqueda lineal auto-organizada). Además, también hemos aprendido a utilizar los diccionarios, ya que hemos tenido que implementar el TAD Diccionario para obtener las tablas necesarias sobre las cuales buscar nuestras claves (haciendo todas las funciones que ello conlleva como crear, insertar, liberar…).

Además hemos reforzado nuestro conocimiento en punteros ya que teníamos que devolver la posición de la clave mediante el uso de un puntero. Por otra parte, también hemos reforzado el tema referido a la reserva y liberación de memoria a la hora de tener que reservar memoria para nuestras tablas.

En caso de que existiese algún tipo de interés en saber cómo arreglamos el código, detectamos el fallo en nuestro array de operaciones básicas y lo cambiamos por un entero.