Análisis de Algoritmos 2016/2017

Práctica 3

David García Fernández y Antonio Martín Masuda, Grupo 1201 – Pareja 1

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica no tuvimos que tomar muchas decisiones de implementación. La implementación del TAD Diccionario fue bastante directa dado que no había grandes novedades respecto a la implementación de otros TAD y las funciones de búsqueda no suponían un gran reto al haberlas visto en teoría. Además, las funciones de tiempo se asemejaban mucho a las realizadas en la práctica uno, por lo que teníamos una idea clara de cómo realizar esta práctica.

2. Objetivos

2.1 Apartado 1

El objetivo de este apartado era implementar un TAD Diccionario que almacenase claves (en este caso números enteros positivos) que debían estar ordenados o no dependiendo de si el valor del dato orden de la estructura vale ORDENADO o NO_ORDENADO. Se podrán insertar las claves de una en una o una tabla completa siempre y cuando haya espacio en el diccionario. Una vez creado y llenado con claves, estas se podrán buscar mediante una búsqueda lineal, binaria (el diccionario ha de estar ordenado) o una búsqueda lineal automática (el dato buscado, si se encuentra, se cambia una posición adelante para que en posteriores búsquedas se encuentren antes, premiando las claves más buscadas).

2.2 Apartado 2

El objetivo en este apartado es implementar unas funciones para medir el tiempo promedio y el número de operaciones básicas máximas, mínimas y promedio que realiza cada función de búsqueda (mencionadas en el apartado anterior) en diferentes contextos de diccionarios e imprimir los datos en un fichero de texto pasado el nombre de este como argumento de entrada.

3. Herramientas y metodología

Ambos apartados han sido realizados en Linux y ejecutados con el comando Valgrind para comprobar que no se producen pérdidas de memoria. En el segundo apartado hemos usado Gnuplot para obtener las gráficas de tiempos y operaciones básicas que se muestran en la sección cinco de resultados y gráficas.

3.1 Apartado 1

La implementación del TAD Diccionario es muy directa, sin ningún elemento novedoso. Implementamos una función que crea memoria para la estructura del diccionario y otra que la libera. Una función que inserta un elemento al diccionario comprobando si hay espacio en él y lo ordena si el campo orden del diccionario vale ORDENADO. La ordenación se realiza mediante un código similar al del método InsertSort, dado que la clave se inserta al final y se compara con el elemento anterior hasta encontrar su sitio. La función insercion_masiva_diccionario inserta una tabla de enteros que recibe como argumento en el diccionario. Esta comprueba primero que hay espacio suficiente en el diccionario para insertarlos y llama en bucle a la función anterior hasta llegar al final de la tabla. Posteriormente realizamos las funciones de

búsqueda. La búsqueda lineal recorre el array entero del diccionario hasta que encuentra la clave o llega al final de este. La búsqueda lineal auto realiza el mismo procedimiento, pero al encontrar la clave, cambia su posición con el anterior, almacenando la nueva posición, para que los elementos más buscados estén al principio y cueste menos tiempo y comparaciones encontrarlos. Finalmente, la búsqueda binaria es una función recursiva. Para que funcione correctamente, el diccionario ha de estar ordenado. Primero, compara la clave con el elemento medio de la tabla. Si es igual ya la ha encontrado, si es menor se llama a sí misma con la tabla izquierda y si es mayor se llama a sí misma con la tabla de la izquierda. La recursión se acaba cuando encuentra la clave o cuando no hay más tabla. La función busca_diccionario recibe el método con el que se va a buscar una clave y se encarga de llamar a dicho método, guardar en el campo ppos la posición en la que se encuentra la clave y devuelve el número de operaciones básicas que se han realizado.

3.2 Apartado 2

En este apartado implementamos las funciones que almacenan e imprimen en un fichero los siguientes datos: tamaño de la tabla del diccionario, tiempo promedio que tarda en buscar las claves, número de operaciones básicas promedio que realiza la función de búsqueda para encontrar dichas claves, el número máximo y mínimo de operaciones básicas realizadas. Esta tarea es realizada por tres funciones. La primera es tiempo_medio_busqueda que según sus parámetros de entrada, crea un diccionario y una permutación mediante la función genera_perm, ambos de tamaño n_claves. Inserta dicha permutación en el diccionario mediante insercion_masiva_diccionario y crea otra tabla de tamaño n_claves*n_veces que serán las claves a buscar. Dicha tabla se recorre en un bucle y se llama al método que recibe como argumento buscando cada clave. Esta función se encarga de medir el tiempo de cada búsqueda y de almacenar en una estructura de tiempo todos los datos anteriormente mencionados.

La función genera_tiempos_busqueda se encarga de generar memoria para una tabla de estructuras de tiempo. Llamará en un bucle que recorre todos los tamaños del diccionario, desde un número mínimo hasta un máximo y con un cierto incremento dados, a la función anterior y cada estructura almacenará los datos del tiempo y operaciones básicas para un tamaño dado del diccionario. Posteriormente, llamará a guarda_tabla_tiempos que se encarga de imprimir en un fichero dado toda la información de la estructura de tiempo. Esta función ya estaba implementada de las prácticas anteriores y no se ha necesitado modificarla.

4. Código fuente

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

```
PDICC ini_diccionario (int tamanio, char orden){
  DICC* diccionario;
  if(tamanio <= 0 || (orden != ORDENADO && orden !=  NO_ORDENADO)){
    return NULL;
  }
```

```
  diccionario = (DICC*)malloc(sizeof(DICC));
  if(!diccionario) return NULL;
  diccionario->tamanio = tamanio;
  diccionario->n_datos = 0;
  diccionario->orden = orden;
  diccionario->tabla = (int*)malloc(sizeof(int)*tamanio);
  if(!diccionario->tabla){
    free(diccionario);
    return NULL;
  }
  return diccionario;
}

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

int inserta_diccionario(PDICC pdicc, int clave){
  int i, buff, counter = 0;
  if(!pdicc || clave <= 0)
    return ERR;
  if(pdicc->n_datos == pdicc->tamanio)
    return ERR;
  pdicc->tabla[pdicc->n_datos] = clave;
  pdicc->n_datos++;
  if(pdicc->orden == ORDENADO){
    buff = pdicc->tabla[pdicc->n_datos-1];
    i = pdicc->n_datos - 2;
    while(i >= 0 && pdicc->tabla[i] > buff){
      pdicc->tabla[i + 1] = pdicc->tabla[i];
      i--;
      counter++;
    }
    if(counter > 0)
      pdicc->tabla[i+1] = buff;
  }
  return counter;
}

int insercion_masiva_diccionario(PDICC pdicc, int *claves, int
                                                    n_claves){
  int j, counter, cerror;
  if(!pdicc || !claves || n_claves <= 0 || ((pdicc->n_datos +
      n_claves) > pdicc->tamanio)){
    return ERR;
  }
  for(j = 0, counter = 0, cerror = 0; j < n_claves ; j++){
```

```
    cerror += inserta_diccionario(pdicc, claves[j]);
    if(cerror == ERR){
      return ERR;
    }
    counter += cerror;
  }
  return counter;
}

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

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

int bbin(int *tabla,int P,int U,int clave,int *ppos){

  int counter = 0, error, medio;
  if(!tabla || P > U){
    return NO_ENCONTRADO;
  }
  medio = (P+U)/2;
  if(tabla[medio] == clave){
      *ppos = medio + 1;
      return 1;
  }
  if(tabla[medio] < clave){
    P = medio + 1;
    /*U se queda igual*/
    counter++;
  }
  else{
    /*P se queda igual*/
    U = medio - 1;
    counter++;
  }
  error = bbin(tabla, P, U, clave, ppos);
```

```
  if(error == NO_ENCONTRADO){
    return NO_ENCONTRADO;
  }
  counter += error;
  return counter;
}
int blin(int *tabla,int P,int U,int clave,int *ppos){
  int i, counter = 0;
  if(!tabla || P < 0 || U < P || clave <= 0){
    return ERR;
  }
  for(i = P, counter = 0; i <= U && tabla[i] != clave;i++,counter++);
  if(i > U){
    return NO_ENCONTRADO;
  }
  *ppos = i + 1;
  counter++;
  return counter;
}

int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
  int i, counter, buff;
  if(!tabla || P < 0 || U < P || clave <= 0)
    return ERR;
  for(i = P, counter = 0; i <= U && tabla[i] != clave; i++,counter++);
  if(i > U)
    return NO_ENCONTRADO;
  if(i == P){
    *ppos = 1;
    return 1;
  }
  buff = tabla[i];
  tabla[i] = tabla[i-1];
  tabla[i-1] = buff;
  *ppos = i;
  counter++;
  return counter;
}
```

4.2 Apartado 2

```
short tiempo_medio_busqueda(pfunc_busqueda metodo,
                            pfunc_generador_claves generador,
                              int orden,
                              int tamanio,
                              int n_claves,
                              int n_veces,
                              PTIEMPO ptiempo){
  int *ob, *claves, *permutacion, error, min = INT_MAX, max = 0, ppos,
      ret = 0, ret2, i;
```

```
PDICC dicc;
double t_medio = 0, obmedio = 0;
clock_t t_ini, t_fin;

if(!metodo || !generador || (orden != ORDENADO && orden !=
    NO_ORDENADO) || tamanio <= 0 || n_claves <= 0 || n_veces <= 0 ||
    !ptiempo)
  return ERR;

dicc = ini_diccionario(tamanio, orden);
if(!dicc)
  return ERR;

ob = (int*)malloc(sizeof(int)*n_claves);
if(!ob){
  libera_diccionario(dicc);
  return ERR;
}

permutacion = genera_perm(n_claves);
if(!permutacion){
  libera_diccionario(dicc);
  free(ob);
  return ERR;
}

error = insercion_masiva_diccionario(dicc, permutacion, n_claves);
if(error == ERR){
  libera_diccionario(dicc);
  free(ob);
  return ERR;
}

claves = (int*)malloc(sizeof(int)*n_claves*n_veces);
if(!claves){
  free(ob);
  libera_diccionario(dicc);
  free(permutacion);
  return ERR;
}

generador(claves, n_veces*n_claves, n_claves);

for(i = 0; i < n_claves*n_veces; i++){
    t_ini = clock();
    ret2 = busca_diccionario(dicc, claves[i], &ppos, metodo);
    t_fin = clock();
    ret += ret2;
    if(ret2 < min )
      min = ret2;
```

```
      if(ret2 > max)
        max = ret2;
      t_medio += (double)(t_fin - t_ini)/CLOCKS_PER_SEC;
    }

    obmedio =(double) ret/(n_veces*n_claves);
    t_medio = t_medio/(n_veces*n_claves);

    /*Actualizamos la estructura de tiempo*/
    ptiempo->tamanio = tamanio;
    ptiempo->n_perms = n_claves;
    ptiempo->n_veces = n_veces;
    ptiempo->tiempo = t_medio;
    ptiempo->medio_ob = obmedio;
    ptiempo->min_ob = min;
    ptiempo->max_ob = max;
    free(ob);
    libera_diccionario(dicc);
    free(permutacion);
    free(claves);
    return OK;
}

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){
    TIEMPO** tiempo;
    int i, j, N;

    if(!metodo || !generador || !fichero || num_min <= 0 || num_max <
       num_min || incr <= 0 || n_veces <= 0 || (orden != ORDENADO &&
       orden != NO_ORDENADO))
      return ERR;

    N = ((num_max - num_min) / incr) + 1;

    tiempo = (TIEMPO**)malloc(sizeof(TIEMPO*)*N);

    for(i = num_min, j=0; i <= num_max ; i = i+incr, j++){
      tiempo[j] = (TIEMPO*)malloc(sizeof(TIEMPO));
              if(!tiempo[j]){
                    for(j = j - 1; j >= 0 ; j--){
                          free(tiempo[j]);
                    }
                    free(tiempo);
                    return ERR;
              }
```

```
    if(tiempo_medio_busqueda(metodo, generador, orden, i, i, n_veces,
      tiempo[j]) == ERR){
      for(i = 0; i < N ; i++){
        free(tiempo[i]);
      }
      free(tiempo);
      return ERR;
    }

  }
  if(guarda_tabla_tiempos(fichero, tiempo, N) == ERR){
    for(i = 0; i < N ; i++){
        free(tiempo[i]);
      }
    free(tiempo);
    return ERR;
  }
  for(i = 0; i < N ; i++){
    free(tiempo[i]);
  }
  free(tiempo);
  return OK;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

Ejecutamos el programa con la orden `>./ejercicio1 -tamanio 10 -clave 8` y obtenemos la siguiente respuesta:

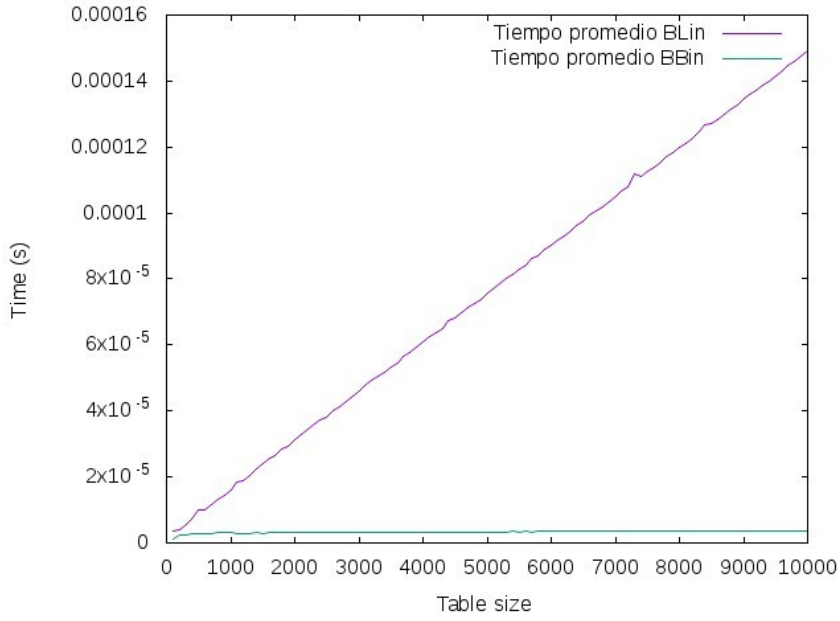
```
Estado original del diccionario no ordenado:
10:0:0[  ]
Estado del diccionario no ordenado tras la inserción:
10:10:0[ 3 9 6 2 4 10 1 5 7 8  ]
Búsqueda lineal:
Clave 8 encontrada en la posicion 10 en 10 op. basicas
Búsqueda lineal auto:
Clave 8 encontrada en la posicion 9 en 10 op. basicas
Estado del diccionario no ordenado tras ejecutar blin_auto:
10:10:0[ 3 9 6 2 4 10 1 5 8 7  ]
Estado inicial del diccionario ordenado:
10:0:1[  ]
Estado del diccionario ordenado tras la inserción:
10:10:1[ 1 2 3 4 5 6 7 8 9 10  ]
Búsqueda binaria:
Clave 8 encontrada en la posicion 8 en 2 op. basicas
Búsqueda lineal ordenada:
Clave 8 encontrada en la posicion 8 en 8 op. basicas
```

5.2 Apartado 2

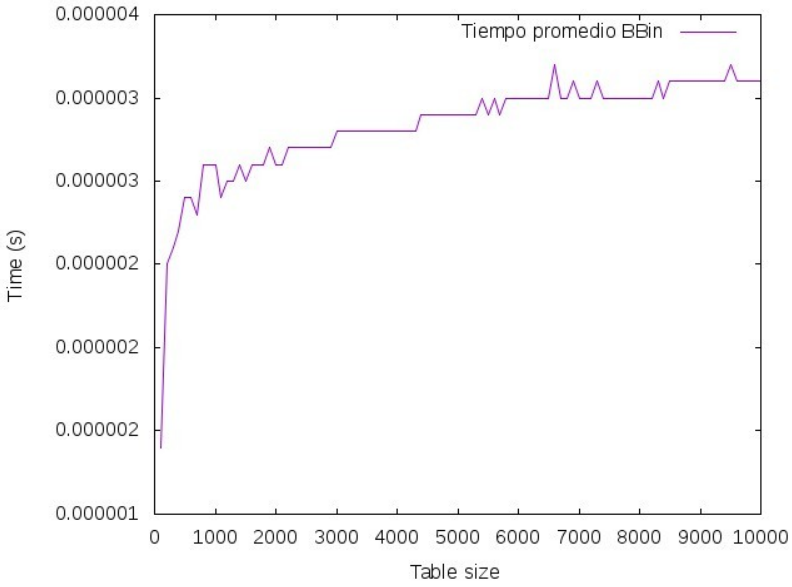
Hacemos las comparaciones que nos piden en el enunciado de la práctica:

Comparación de busqueda lineal (diccioanrios desordenados) y busqueda binaria (diccionarios ordenados), con n_veces = 1 y usando generador de claves uniforme. Los diccionarios tienen tamaños comprendidos entre los 1000 y 10000 elementos, con un incremento de 100.

Tiempo medio de búsqueda:



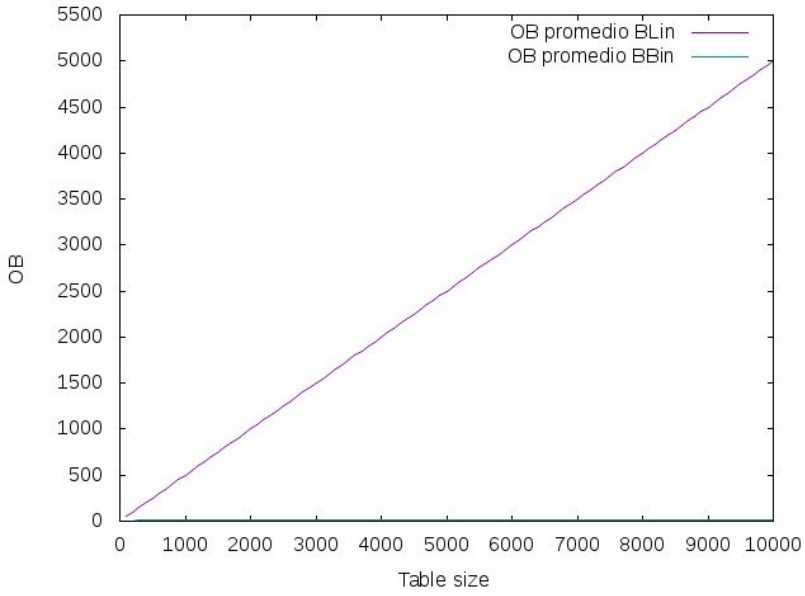
Como podemos ver en la gráfica el tiempo medio de búsqueda para la búsqueda lineal es una función claramente lineal. A medida que el tamaño de la tabla aumenta el coste de tiempo crece equitativamente. El promedio temporal de la busqueda binaria es mucho más pequeño y no se puede apreciar correctamente en esta gráfica comparativa por lo que los representaremos por separado en la siguiente:



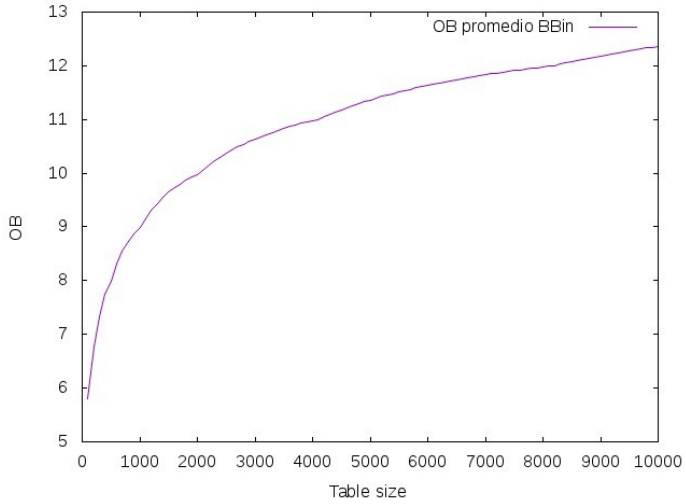
Aquí podemos observar como la busqueda binaria tiene un tiempo promedio de ejecución que se adapta claramente a la función $f(x) = \log(x)$ siendo x el tamaño de la tabla. El resultado es el esperado según los estudios previos que hemos realizado en la parte teórica.

OB promedio de búsqueda:

Utilizando los mismos parámetros hemos obtenido la siguiente salida en nuestro fichero:



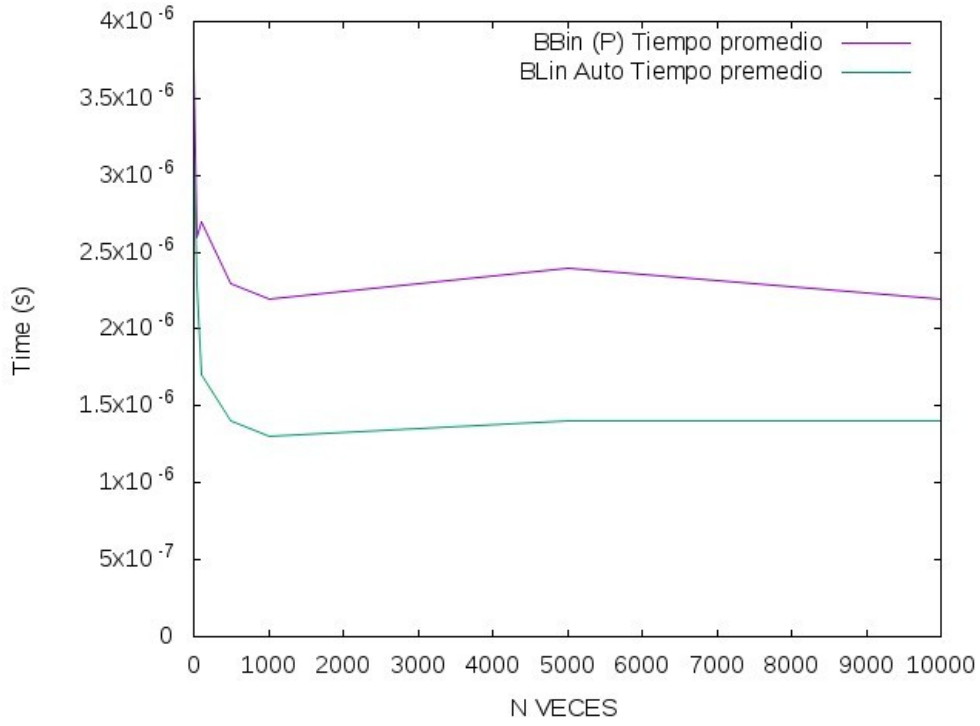
De nuevo no podemos apreciar bien los resultados de la búsqueda binaria pero sí que podemos ver que las OB promedio de la búsqueda lineal se rigen por la función $f(x) = \frac{1}{2}x$, lo cual tiene mucho sentido, ya que las OB mínimas posibles serían 1 en todos los casos y máximas el mismo tamaño de la tabla (x). De nuevo representamos por separado la búsqueda lineal para poder distinguir bien su comportamiento:



Aquí obtenemos una gráfica que, de nuevo se ciñe a la de la función $f(x) = \log(x)$ salvo por alguna pequeña constante como en el caso de la búsqueda lineal. Si nos fijamamos, en la parte teórica hemos visto que el caso peor de la búsqueda binaria es log(x), para x el tamaño de la tabla. En nuestra tabla de mayor tamaño (x = 10000) tenemos que el caso peor es de log(10000) = 13,287. El caso mejor es 1. Por lo que observamos que nuestra gráfica está comprendida entre esos valores y por tanto su comportamiento es el esperado.

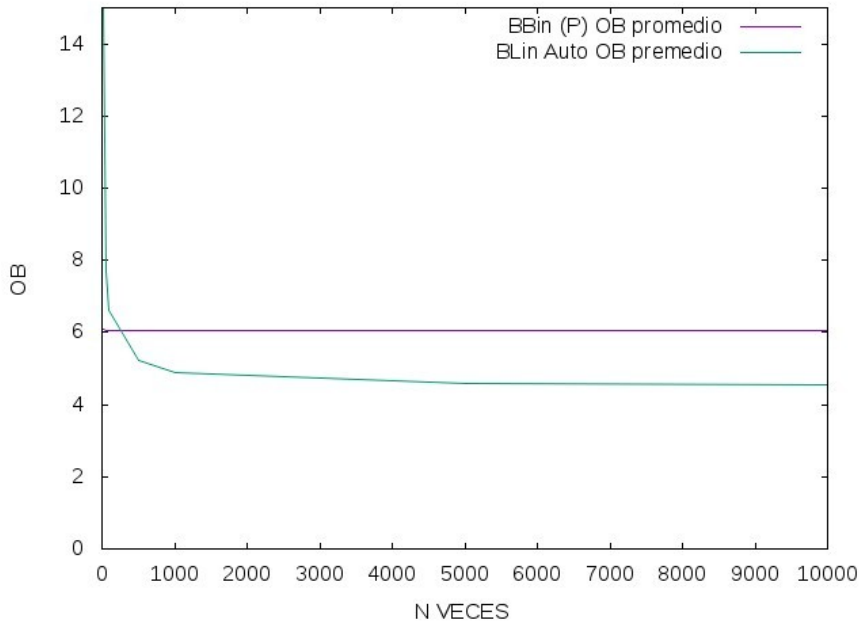
Ahora procedemos a hacer la comparación en diccionarios de 100 elementos, en los que variaremos el número de veces que buscamos las claves (previamente generadas con el generador de claves potencial) entre 1 y 10000 veces. Para la comparación utilizaremos las funciones busqueda lineal auto-organizaday búsqueda binaria:

Tiempo medio de búsqueda:



Se observa que el "gran cambio" se da en las 1000 primeras veces que se busca la clave. En un inicio el tiempo promedio es mayor para la busqueda lineal auto organizada pero cuando pasamos de las 1000 veces las gráficas ya se han estabilizado, ambas se mantienen prácticamente constantes pero la de la búsqueda binaria es mayor que la de la auto-organizada ya que como vimos la primera tiene un coste cercano a O(1) y la segunda un coste O(log(n)). La gráfica de la búsqueda binaria debería ser más estable pero quizá algún proceso interno del ordenador hizo que se tardase más en su ejecución, veremos que esta estabilidad sí aparece en las operaciones básicas promediadas en la misma ejecución del programa.

OB promedio de búsqueda



En esta ocasión se justifica la caída en el tiempo medio de ejecución en la búsqueda auto-organizada en las 1000 primeras repeticiones. Mientras que las OB promedio de la búsqueda binaria (como comentabamos anteriormente) se mantienen constantes ya que el tamaño del diccionario es constante también las de la blin-auto caen en picado. La explicación es que las claves más buscadas se colocan más al principio, y por tanto se encuentan mucho más rápido en las siguientes repeticiones.

En el modelo de la memoria se nos propone comparar el promedio de tiempo y OB entre la búsqueda lineal, la búsqueda lineal auto y la búsqueda binaria. A partir de los resultados vistos podemos decir que la búsqueda lineal y la auto no se diferencian en nada si utilizamos el generador de claves no potencial y lo hacemos para un número de veces fijo. En tal caso las gráficas que obtendríamos son las mismas que en el primer apartado.

5. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

5.1 Pregunta 1

La operación básica de las tres funciones de búsqueda es la comparación de claves. En el caso de la búsqueda lineal y búsqueda lineal auto, la operación básica se realiza en el bucle for (dentro del paréntesis) donde se compara tabla[i] con la clave. En cambio, en la búsqueda binaria se compara en el primer if si tabla[i] es igual que la

clave. Aunque, en el siguiente if también se realiza una comparación, hemos considerado que solo se hace una comparación de clave en cada llamada, dado que en realidad está comparando lo mismo y el motivo por el cual se realiza es comprobar si está en la subtabla izquierda o en la subtabla derecha.

5.2 Pregunta 2

Búsqueda lineal:

$W_{BLin}(N) = N$ ya que debemos comprobar si la clave es cualquiera de los N elementos de la tabla, si no está en la N-ésima comparación es que no se encuentra en el diccionario.

$B_{Blin}(N) = 1$ ya que la clave podría estar en en la primera posición.

Búsqueda binaria:

$W_{BBin} = E(\log(N))+1$ (parte entera de log(N) + 1 o techo de log(N)) ya que a cada llamada recursiva dividimos la tabla a la mitad.

$B_{BBin} = 1$ ya que en la primera comparación (en la posición media de la tabla) podría encontrarse la clave.

5.3 Pregunta 3

En la búsqueda lineal auto, los elementos más buscados van adelantando posiciones en la tabla según avanza las búsquedas. Este método puede ser muy ventajoso en los casos en los que hay unos pocos elementos que se buscan muy a menudo y el resto que se buscan muy de vez en cuando.

5.4 Pregunta 4

En la gráfica obtenida hemos podido observar que para un tamaño de 100 elementos nuestra gráfica de ejecución se mantiene constante cuando ha alcanzado al estabilidad en torno a las 5 OB (si bien incluso continúa bajando). Al ir colocando las claves según “necesidad” conseguimos que las más solicitadas se obtengan en las primeras iteraciones por lo que el coste final llega a ser de O(1).

5.5 Pregunta 5

La búsqueda binaria es el mejor método de búsqueda en tablas ordenadas. Este algoritmo compara el elemento medio de la tabla con la clave. En caso de que sea justo la clave, termina el algoritmo. En caso contrario, mira si la clave es menor que el elemento medio o no. En caso afirmativo, se vuelve a llamar a sí misma con la subtabla izquierda y realiza el mismo procedimiento. En caso negativo, se llama a sí misma con la subtabla izquierda. Así sucesivamente hasta que encuentra la clave o bien no hay más tabla. Es decir, si la clave a buscar no está en la tabla, llega un momento en el que se tiene que llamar a sí misma con una primera posición de la tabla mayor que la última posición de la tabla (la tabla se ha acabado).

6. Conclusiones finales.

En esta última práctica hemos abordado los algoritmos de búsqueda, alejándonos de los algoritmos de ordenación con los que habíamos trabajado prácticas anteriores. En este caso la implementación ha sido bastante sencilla, el TAD diccionario y los algoritmos de búsqueda son bastante sencillos, quizá la búsqueda binaria sea algo más rebuscada por ser recursiva pero bastante sencilla aún así. En cuanto a la medición de los tiempos de búsqueda el método ha sido análogo al de ordenación por lo que hemos podido impementar os conocimientos obtenidos en anteriores prácticas.

Esta práctica nos ha servido para ilustrar los resultados teóricos (caso mejor, peor y medio de cada algoritmo) y para aprender sobre nuevos métodos de búsqueda para determinadas situaciones, como la búsqueda lineal automatizada que no habíamos visto en clase pero cuya utilidad hemos comprendido y hemos sido capaces de implementarla y entender a la perfección su funcionamiento y su ventajas y desventajas respecto a otros tipos de búsquedas.