Análisis de Algoritmos 2018/2019

Práctica 3

Román García, 1271.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se propone la comparación de dos algoritmos de búsqueda.

A continuación, se muestra el proceso seguido para la elaboración de esta.

2. Objetivos

2.1 Apartado 1

En este apartado se pide la implementación del TAD DICC, correspondiente a un diccionario, así como los algoritmos de búsqueda “Búsqueda binaria” y “Busqueda lineal”

2.2 Apartado 2

En este caso se pide la implementación de las funciones necesarias para la comparación de tiempos y rendimiento de los algoritmos de búsqueda implementados en el apartado anterior

3. Herramientas y metodología

Para todos los apartados he utilizado la misma metodología y las mismas herramientas:

Para saber qué piden exactamente he usado un boceto en papel de lo que debería hacer cada una de las funciones a codificar.

Una vez analizado el problema y diseñada la solución, he usado la combinación del IDE “Visual Studio Code” para codificar, “GitHub” para mantener un repositorio de versiones y una “bash Ubuntu 18.04” de “Windows” que he utilizado para compilar y ejecutar el código.

Para la creación de gráficas he usado “Gnuplot” y bash script para el procesamiento de datos.

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)
{
    PDICC dic = NULL;
    int *tabla = NULL;
```

```
    if (tamanio < 1 || (orden != NO_ORDENADO && orden != ORDENADO))
    {
        return NULL;
    }

    dic = (PDICC)malloc(sizeof(DICC));
    if (!dic)
    {
        return NULL;
    }

    tabla = (int *)malloc(sizeof(int) * tamanio);
    if (!tabla)
    {
        return NULL;
    }

    dic->tamanio = tamanio;
    dic->orden = orden;
    dic->n_datos = 0;
    dic->tabla = tabla;

    return dic;
}

void libera_diccionario(PDICC pdicc)
{
    if (!pdicc)
    {
        return;
    }

    free(pdicc->tabla);
    free(pdicc);

    return;
}

int inserta_diccionario(PDICC pdicc, int clave)
{
    int j, A;
    int ip = 0;
    int ob = 0;
    int n_datos;
    if (!pdicc || clave < 0)
    {
        return ERR;
    }
```

```
    if (pdicc->n_datos >= pdicc->tamanio)
    {
        return ERR;
    }

    pdicc->tabla[pdicc->n_datos] = clave;
    n_datos = pdicc->n_datos;
    pdicc->n_datos++;

    if (pdicc->orden == ORDENADO)
    {
        A = pdicc->tabla[n_datos];
        j = n_datos - 1;

        while (j >= ip && pdicc->tabla[j] > A)
        {
            ob++;
            pdicc->tabla[j + 1] = pdicc->tabla[j];
            j--;
        }
        pdicc->tabla[j + 1] = A;
        if (j >= ip)
        {
            ob++;
        }
    }

    return ob;
}

int insercion_masiva_diccionario(PDICC pdicc, int *claves, int n_claves)
{
    int i;

    if (!pdicc || !claves)
    {
        return ERR;
    }

    for (i = 0; i < n_claves; i++)
    {
        if (inserta_diccionario(pdicc, claves[i]) == ERR)
        {
            return ERR;
        }
    }

    return OK;
}
```

```c
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda
metodo)
{
    pfunc_busqueda metodo2 = bbin;
    int ob = 0;

    if (!pdicc || !ppos || !metodo || clave < 0)
    {
        return ERR;
    }

    if (metodo == metodo2 && pdicc->orden == NO_ORDENADO)
    {
        return ERR;
    }

    ob = metodo(pdicc->tabla, 0, pdicc->n_datos - 1, clave, ppos);

    if (*ppos < 0)
    {
        *ppos = NO_ENCONTRADO;
        return NO_ENCONTRADO;
    }
    return ob;
}
int bbin(int *tabla, int P, int U, int clave, int *ppos)
{
    int ob = 0;
    int medio;

    while (P <= U)
    {
        medio = ((U - P) / 2) + P;
        if (tabla[medio] == clave)
        {
            *ppos = medio + 1; /* Porque medio es el indice y ppos la
posicion a mostrar*/
            return ob + 1;
        }
        else if (clave < tabla[medio])
        {
            U = medio - 1;
            ob++;
        }
        else if (clave > tabla[medio])
        {
            P = medio + 1;
            ob++;
```

```
        }
    }

    *ppos = NO_ENCONTRADO;
    return ERR;
}

int blin(int *tabla, int P, int U, int clave, int *ppos)
{
    int i;

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

    *ppos = NO_ENCONTRADO;
    return ERR;
}
```

4.2 Apartado 2

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

  int i, ob, mob;
  clock_t t_ini, t_fin;
  double secs;
  PDICC dic;
  int ppos;
  int *perm;
  int *claves;

  if (n_claves < 1 || !ptiempo)
    return -1;

  dic = ini_diccionario(n_claves, orden);
  if (!dic)
  {
    return ERR;
  }
```

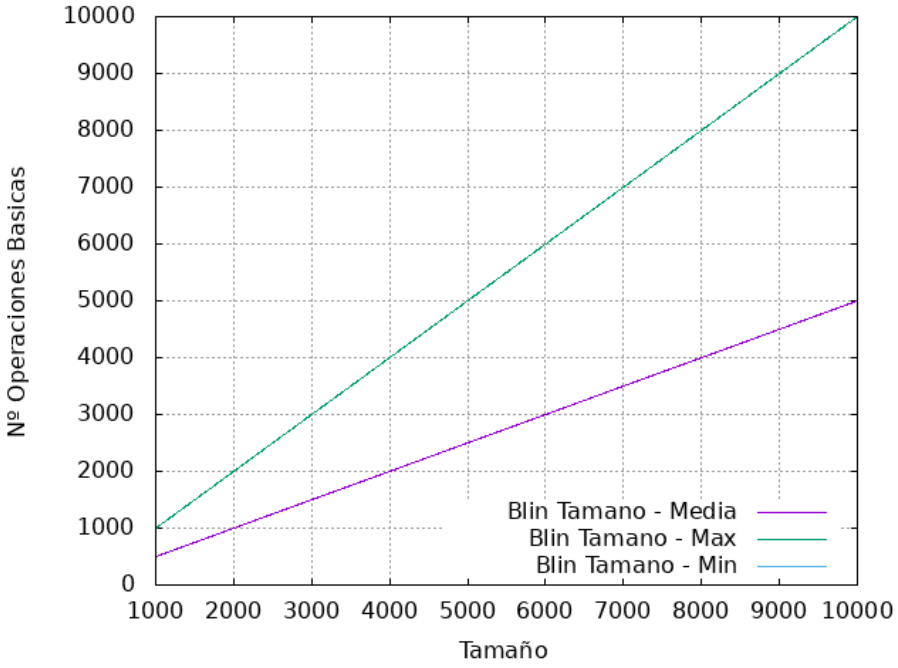
```c
  perm = genera_perm(n_claves);
  if (!perm)
  {
    libera_diccionario(dic);
    return ERR;
  }

  if (insercion_masiva_diccionario(dic, perm, n_claves) == ERR)
  {
    free(perm);
    libera_diccionario(dic);
    return ERR;
  }

  claves = ((int *)malloc(sizeof(int) * n_claves * n_veces)); /*tabla de
claves a buscar*/
  if (!claves)
  {
    free(perm);
    libera_diccionario(dic);
    return ERR;
  }

  generador(claves, (n_claves * n_veces), n_claves);

  for (i = 0, ob = 0, mob = 0, secs = 0; i < n_veces * n_claves; i++)
  {

    t_ini = clock();
    ob = busca_diccionario(dic, claves[i], &ppos, metodo);
    t_fin = clock(); /*Solo queremos el tiempo de busqueda del metodo*/
    if (ob > ERR)
    {
      mob += ob;

      if (i == 0)
      {
        ptiempo->min_ob = ob;
        ptiempo->max_ob = ob;
      }
      else
      {
        if (ob < ptiempo->min_ob)
        {
          ptiempo->min_ob = ob;
        }

        if (ob > ptiempo->max_ob)
        {
```

```
          ptiempo->max_ob = ob;
        }
      }
      secs += (double)(t_fin - t_ini) / ((double)CLOCKS_PER_SEC /
1000000000);
    }
  }
  free(claves);
  free(perm);
  libera_diccionario(dic);

  ptiempo->tiempo = (secs / (n_veces * n_claves)); /*media de lo que
tarda en buscar n_veces una clave*/

  ptiempo->medio_ob = mob / (n_veces * n_claves);
  ptiempo->n_veces = n_veces;
  ptiempo->tamanio = n_claves;

  return OK;
}

short genera_tiempos_busqueda(pfunc_busqueda metodo,
pfunc_generador_claves generador, char orden, char *fichero, int num_min,
int num_max, int incr, int n_veces)
{

  int n_claves, i;
  PTIEMPO ptiempo;

  if (!fichero || num_max < num_min || incr < 1 || n_veces < 1)
  {
    return ERR;
  }

  ptiempo = (PTIEMPO)malloc(sizeof(TIEMPO) * ((num_max - num_min) / incr
+ 1));
  if (!ptiempo)
  {
    return ERR;
  }

  for (i = 0, n_claves = num_min; n_claves <= num_max; n_claves += incr,
i++)
  {

    tiempo_medio_busqueda(metodo, generador, orden, n_claves, n_veces,
&ptiempo[i]);
  }
```

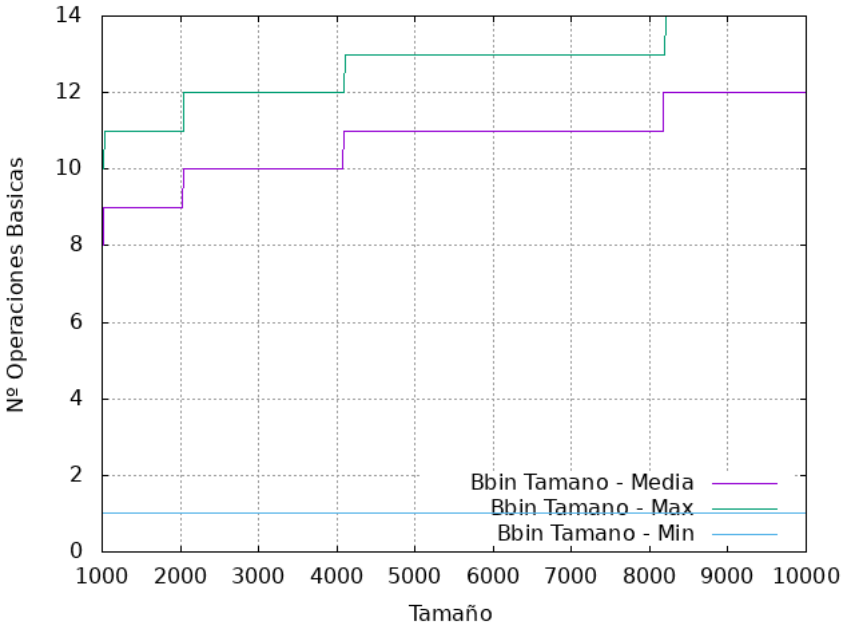
```
  ptiempo->n_perms = i;

  guarda_tabla_tiempos(fichero, ptiempo, i);

  free(ptiempo);

  return OK;
}
```

5. Resultados, Gráficas

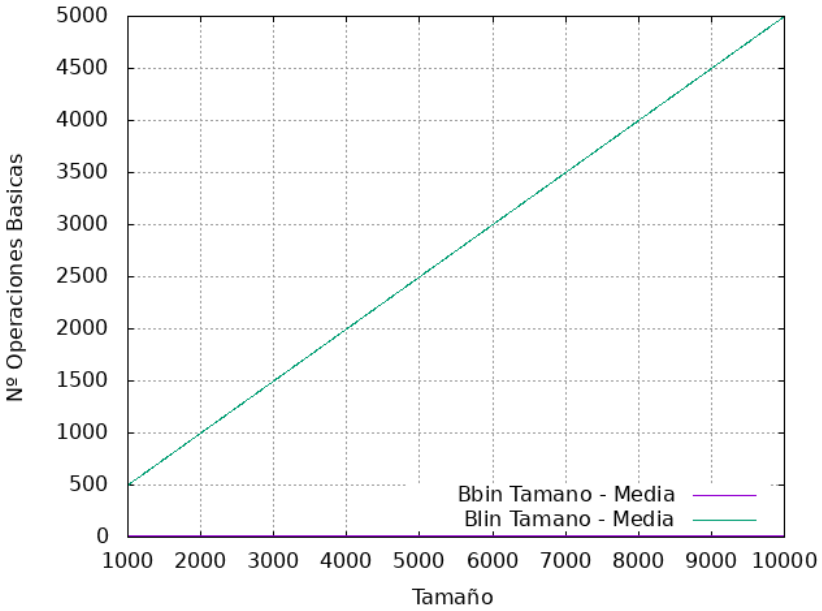
A continuación, podemos ver la gráfica correspondiente a las ob medias, máximas y mínimas de blin
Como se puede apreciar, el mínimo es muy pequeño en comparación al máximo y la media. Esto es debido a que el mínimo en bbin es 1.



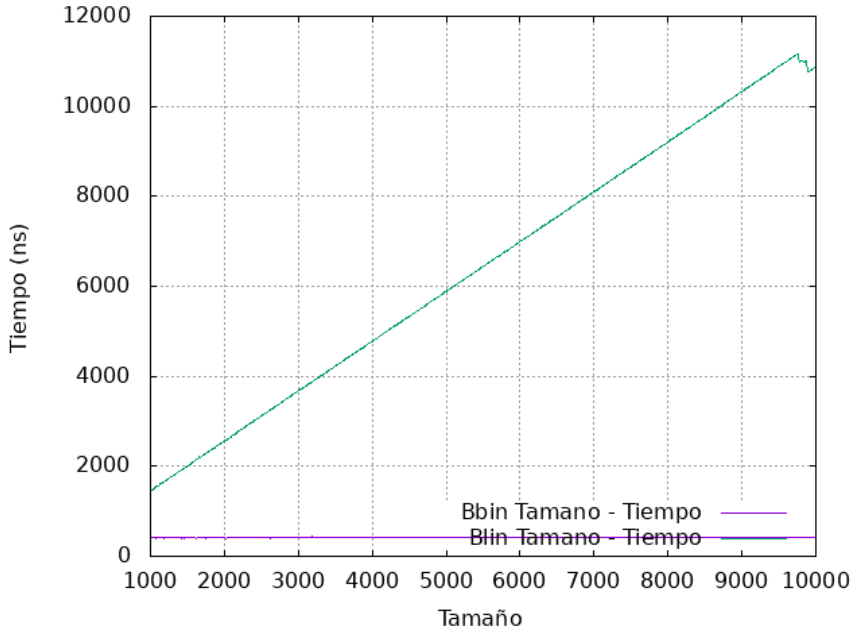
En la siguiente grafica se ven representadas las ob con respecto al tamaño del algoritmo bbin.
Existen varios dientes de sierra debido a que, al tener un rendimiento medio logarítmico, la escala debería representarse en décadas para eliminar estos cambios bruscos.
De nuevo, el mínimo es 1



La siguiente gráfica corresponde a la comparación de obs de los algoritmos bbin y blin
Como podemos observar, hay una diferencia abismal para tamaños grandes.
Aunque aparentemente la media de bbin esta representada como una recta situada en el 0, realmente tiene un crecimiento logarítmico pero los valores son cercanos a 10, por lo que, en comparación a blin, tiene varios ordenes de magnitud menos.



Por últimos tenemos una gráfica que compara los tiempos entre ambos algoritmos. Comparando ambos, podemos determinar que blin tiene un rendimiento con crecimiento lineal mientras que bbin tiene un crecimiento, casi imperceptible, log-lineal.



5. Respuesta a las preguntas teóricas.

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

5.1 ¿Cuál es la operación básica de bbin y blin?

La comparación de clave. En blin se realiza para cada elemento precedente al buscado y en bbin para cada nodo padre.

5.2 Dar tiempos de ejecucion en funcion del tamaño de entrada n para el caso peor Wss(n) y el caso mejor Bss(n) de bbin y blin. Utilizar la notación asintótica siempre que se pueda.

	Wss	Bss
Busqueda binaria	$W_{BBin}(N) = \lceil \lg(N) \rceil$	O(1)
Busqueda lineal	$W_{BLin}(N) = N$	O(1)

5.3 ¿Cuál es el orden de ejecucion medio de blin y bbin en funcion del tamaño de elementos en el diccionario?

	Ass
Busqueda binaria	$A^{e}_{BBin}(N) = \lg(N) + O(1)$
Busqueda lineal	$A^{e}_{BLin}(N) = \sum_{i=1}^{N} n_{BLin}(k = T[i])\, p(k == T[i]) \sim \frac{S_N}{C_N}$

5.4 Justifica lo mas formalmente que puedas la corrección de los algoritmos bbin y blin

Busqueda lineal, con tablas ordenadas, tiene una comparación de clave por cada elemento menor que el buscado mas el propio. Funciona porque va elemento a elemento hasta que encuentra el que se esta buscando o no quedan mas elementos en la tabla

Busqueda binaria juega con estructurar los elementos en forma de árbol, de forma que, a un lado de un nodo sean todos menores que ese elemento y, al otro lado, mayores. Con esta estructura de datos es mucho más fácil encontrar un elemento cuando nos encontramos con una gran cantidad de ellos, dado que, cuanto más profundizamos, más precisión tenemos sobre la existencia o no del elemento. Por lo tanto, si se llega a la máxima profundidad, o a una incongruencia como un elemento mayor al buscado realizando la búsqueda por el lado izquierdo, sabemos fácilmente y sin tener que recorrer la tabla completa, que no existe el elemento.

6. Conclusiones.

En esta practica hemos estudiado el rendimiento de dos algoritmos de búsqueda. Existe una gran diferencia entre ambos, cuando nos enfrentamos a grandes cantidades de datos. No obstante, la diferencia se acorta, hasta invertirse, cuando la cantidad de datos es suficientemente pequeña.

Podemos concluir que, no hay un algoritmo de búsqueda ideal para todos los casos, aunque es cierto que para la mayoría de los casos hay una alta probabilidad de que uno de ellos sea mucho mejor que el otro.