	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): Luis Ricardo López Villafán

Asignatura: Estructura de datos y Algoritmos

Grupo: 01

No de Práctica(s): 07:Práctica de las estructuras BST y Tablas Hash (Criptoanálisis en campo)

Integrante(s):

Lopez Cruz Jaciel Adrian

Miguel Mata Jared

No. de lista o brigada: NoCompila

Semestre: 2026-1

Fecha de entrega: 05/11/2025

Observaciones:

CALIFICACIÓN: _____

Análisis de Frecuencia:

Tomamos el mensaje cifrado:

(/-.-4%(+28.%#+2/($(6(#(3(8%.-/2(+(/(6.(

1. **Contamos las frecuencias de cada símbolo**:

- (→ 11
- / → 4
- . → 4
- - → 3
- % → 3
- + → 3
- 2 → 3
- 8, #, 6 → 2 cada uno
- 4, $, 3 → 1 cada uno

2. Total: 40 caracteres.

En español, la letra más frecuente suele ser **"e",** por lo que la mejor candidata para la **e** es el símbolo más repetido: (

De ahí sale el primer par:

(→ e

3. Luego miramos al **inicio del texto** como patrón de letras, no por significado todavía:
    - Cifrado (primeros 7 símbolos): (/-.-4%
    - Distintos símbolos, marcando repetidos:
        - (= 1
        - / = 2
        - - = 3
        - . = 4
        - - (repite 3)
        - 4 = 5
        - % = 6
    - Patrón: **1 2 3 4 3 5 6**
4. Ahora buscamos una palabra que pudiera ser orden militar y tenga **el mismo patrón**:
    - "elataque"
        - e = 1
        - l = 2

        - a = 3
        - t = 4
        - a (repite 3)
        - q = 5
        - u = 6
    - Patrón: **1 2 3 4 3 5 6**

Esto cuadraba perfecto con el inicio (/-.-4%.

Entonces asignamos:

(→ e

/ → l

- → a

. → t

4 → q

% → u

Y por tanto el mensaje empieza como:

(/-.-4% → elataque

5. Luego seguimos avanzando con el mismo truco de **patrones de repetición**:
   Después de “elataque” viene:
   (+28.%#+2/(
6. Probando con las letras asignadas y para generar palabras, llegamos a “nocturno”,que encajaba con la predicción que habíamos hecho para los anteriores caracteres:
    - + → n
    - 2 → o
    - 8 → c
    - . (ya es t)
    - % (ya es u)
    - # → r
    - + (n, ya visto)
    - 2 (o, ya visto)
    - / (l, ya visto)
    - ((e)

Eso da:
(+28.%#+2/(→ nocturnole

7. (luego vemos el “le”).

Siguiendo esa misma idea en todo el texto, obtenemos un mapeo **consistente 1 a 1** (sustitución simple):
(→ e

/ → l

- → a

. → t

4 → q

% → u

+ → n

2 → o

8 → c

→ r

$ → b

6 → s

3 → j

8. Y al aplicar esto a *todo* el cifrado, sale:
   elataquenocturnolebeserejecutaloeneleste

Cómo hayamos A:

En la práctica nos dicen que el enemigo usa **una función hash de multiplicación** con constante A (desconocida) y tamaño M = 31:

Instrucciones:

$$indiceA = [M \cdot \{kA\}] + 32$$

donde:

- k = código ASCII del carácter original.

- A = constante de dispersión (“la proporción áurea”).
- M=31 = número de cubetas.

La idea de “ingeniería inversa” es:

1. A partir del análisis de frecuencia, **postulamos el par e = “(“**
2. Suponemos que ese par se generó por la función de multiplicación. Entonces conocemos:
    - El ASCII del carácter **original** (por ejemplo, k = 101 para ‘e’).
    - El ASCII del carácter **cifrado**, llamémosle cc.
    - El tamaño de la tabla M = 31.
3. De la fórmula:
    - Primero restamos el desplazamiento 32:

      $h = indiceA - 32$
    - Sabemos que:

      $h = [M\{kA\}]$

      donde {kA} es la **parte fraccionaria**
    - Esto implica el siguiente **intervalo** para {kA}:

      $\frac{h}{M} \leq \{kA\} < \frac{h+1}{M}$
4. La parte fraccionaria se puede escribir como:

   $\{kA\}kA - [kA] = kA - q$

   para algún entero$q = [kA]$

   Sustituyendo en la desigualdad:

   $\frac{h}{M} \leq kA - q < \frac{h+1}{M}$

   Despejando A:

   $\frac{q+\frac{h}{M}}{k} \leq A \leq \frac{q+\frac{h+1}{M}}{k}$
5. Como **se sabe** (por diseño de hash) que A está en el intervalo (0,1), elegimos valores enteros de q que hagan que A quede en (0,1).

   Eso te da **un rango de posibles valores de A** para ese par.
6. Repetimos el procedimiento con **otro par** (otro carácter original conocido y su versión cifrada) y obtenemos **otro rango** para A.

   Al intersectar los rangos todos se acumulan alrededor de:

   $A \approx 0.61803...$

   que es justamente la **razón áurea**:

   $\varphi = \frac{\sqrt{5}-1}{2} = 0.61803...$

Complejidad de analizar_frecuencia: O(logN)

1. ¿Qué hace analizar_frecuencia?

- Recorre el mensaje carácter por carácter.
- Para cada carácter, **buscarlo / insertarlo en el BST**.
- Si ya existe el nodo, solo incrementamos la frecuencia; si no, creamos un nodo nuevo.

2. ¿Por qué buscar/insertar en un BST es O(log N)?

Sea:

- N = número de nodos distintos que hay en el BST (distintos caracteres cifrados).
- En un **BST balanceado**, la altura del árbol es aproximadamente

  $h = log_2 N$

  porque el número máximo de nodos de un árbol de altura h es del orden de 2h2h.

Cuando hacemos una operación de **búsqueda o inserción**:

- Empezamos en la **raíz**.
- En cada paso comparamos el carácter actual con el de ese nodo:
  - Si es menor, bajamos al hijo izquierdo.
  - Si es mayor, bajamos al hijo derecho.
- En el peor caso recorremos **un camino desde la raíz hasta alguna hoja**.

La longitud de ese camino es, como máximo, la **altura del árbol**, es decir h.

Entonces, el número de comparaciones en una búsqueda/insert es:

$T(N) \alpha h = log_2 N \Rightarrow T(N) = O(logN)$

3. ¿Y la función analizar_frecuencia completa?

Si:

- L = longitud del mensaje (número total de caracteres),
- N = número de símbolos distintos que acaban en el árbol,

entonces:

- Para **cada** uno de los L caracteres hago una operación de inserción/búsqueda de coste O(log N).

Por tanto, el coste total es:

$T(L,\ N)\ =\ L\ \bullet\ O(log\ N)\ =\ O\ (L\ log\ N)$

Código:

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

#define M 31         /* tamaño de la tabla hash */
#define MAX_NODOS 128  /* máximo de símbolos diferentes en el árbol */

// Nodo del árbol BST
typedef struct NodoBST {
   char simbolo;          // carácter cifrado
   int frecuencia;        // cuántas veces aparece ese carácter
   struct NodoBST *izq;
   struct NodoBST *der;
} NodoBST;

// Crea un nodo nuevo para el árbol
NodoBST *crearNodo(char c) {
   NodoBST *n = (NodoBST *)malloc(sizeof(NodoBST));
   if (n == NULL) {
      fprintf(stderr, "Error: sin memoria.\n");
      exit(1);
   }
   n->simbolo = c;
   n->frecuencia = 1;
   n->izq = NULL;
   n->der = NULL;
   return n;
}

// Inserta el símbolo en el árbol o le suma 1 a su frecuencia
// (en un árbol balanceado sería O(log N))
NodoBST *insertar(NodoBST *raiz, char c) {
   if (raiz == NULL) {
      return crearNodo(c);
   }
   if (c == raiz->simbolo) {
      raiz->frecuencia++;
   } else if (c < raiz->simbolo) {
      raiz->izq = insertar(raiz->izq, c);
   } else {
      raiz->der = insertar(raiz->der, c);
   }
   return raiz;
}

// FASE 1: análisis de frecuencias con BST

// Recorre el mensaje, mete cada símbolo al árbol
// y guarda en L cuántos caracteres útiles hay
NodoBST *analizar_frecuencia(const char *mensaje, int *L) {
   NodoBST *raiz = NULL;
   int i;
   *L = 0;

   for (i = 0; mensaje[i] != '\0'; i++) {
      char c = mensaje[i];

      // Si hay espacios o saltos de línea, los ignoramos
      if (c == ' ' || c == '\n' || c == '\t') {
         continue;
      }
      raiz = insertar(raiz, c);
      (*L)++;
   }
   return raiz;
}

// Estructura para guardar símbolo y frecuencia en un arreglo
typedef struct {
```

```
    char simbolo;
    int frecuencia;
} Frec;

// Recorre el árbol en orden y llena el arreglo
void volcarBST(NodoBST *raiz, Frec *arr, int *idx) {
    if (raiz == NULL) return;
    volcarBST(raiz->izq, arr, idx);
    arr[*idx].simbolo = raiz->simbolo;
    arr[*idx].frecuencia = raiz->frecuencia;
    (*idx)++;
    volcarBST(raiz->der, arr, idx);
}

// Ordena el arreglo por frecuencia de mayor a menor (burbuja)
void ordenarPorFrecuencia(Frec *arr, int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (arr[j].frecuencia > arr[i].frecuencia) {
                Frec tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }
    }
}

// Muestra la tabla final de frecuencias
void imprimirFrecuencias(Frec *arr, int n, int L) {
    int i;
    printf("=== FASE 1: ANALISIS DE FRECUENCIAS (BST) ===\n");
    printf("L (longitud del mensaje) = %d\n\n", L);
    printf("Simbolo  Frecuencia  Porcentaje\n");
    printf("---------------------------------\n");
    for (i = 0; i < n; i++) {
        double p = (100.0 * arr[i].frecuencia) / L;
        printf("   %c       %3d       %6.2f%%\n",
               arr[i].simbolo, arr[i].frecuencia, p);
    }
    printf("\n");
}

// FASE 2: búsqueda de la constante A

// Busca un valor de A en [0,1] tal que al aplicar el hash
// de multiplicación a 'simboloClaro' salga 'simboloCifrado'.
//
//   indice_A = floor(M * frac(k * A)) + 32
//   k = ASCII del carácter claro.
double buscar_constante_A(char simboloClaro, char simboloCifrado) {
    int k = (int)simboloClaro;
    int objetivo = (int)simboloCifrado;
    double paso = 0.00001;
    double A;

    for (A = 0.0; A < 1.0; A += paso) {
        double x = k * A;
        double frac = x - floor(x);
        int indiceA = (int)floor(M * frac) + 32;
        if (indiceA == objetivo) {
            return A; // primer A que funciona para este par
        }
    }
    return -1.0; // si no encuentro nada, regreso -1
}

// Imprime cómo se ve el hash para cada letra con el A encontrado
void construir_patron_hash(double A) {
    int k;

    printf("=== FASE 2: PATRON HASH (con A ≈ %.5f) ===\n", A);
    printf("Letra  ASCII  idx_mult  ch  idx_mod  ch\n");
    printf("-----------------------------------------\n");

    for (k = 'a'; k <= 'z'; k++) {
        // aquí no agrego la ñ para no complicar el ejemplo
        double x = k * A;
```

```
        double frac = x - floor(x);
        int idx_mult = (int)floor(M * frac) + 32;
        int idx_mod  = (k % M) + 32;
        printf("  %c    %3d   %3d   %c   %3d   %c\n",
               (char)k, k,
               idx_mult, (char)idx_mult,
               idx_mod,  (char)idx_mod);
    }
    printf("\n");
}

// FASE 3: descifrado final

// Mapeo que salió al combinar:
// frecuencias + hash + sentido en español
typedef struct {
    char cifrado;
    char claro;
} ParSust;

// Solo pongo los símbolos que sí aparecen en el mensaje
ParSust mapeo[] = {
    {'(', 'e'},
    {'/', 'l'},
    {'-', 'a'},
    {'.', 't'},
    {'4', 'q'},
    {'%', 'u'},
    {'+', 'n'},
    {'2', 'o'},
    {'8', 'c'},
    {'#', 'r'},
    {'$', 'b'},
    {'6', 's'},
    {'3', 'j'}
};

const int TAM_MAPEO = sizeof(mapeo) / sizeof(mapeo[0]);

// Regresa la letra "real" que le toca a un símbolo cifrado
char descifrar_caracter(char c) {
    int i;
    for (i = 0; i < TAM_MAPEO; i++) {
        if (mapeo[i].cifrado == c) {
            return mapeo[i].claro;
        }
    }
    // Si el símbolo no está en la tabla, lo dejo igual
    return c;
}

// Recorre el mensaje cifrado y va armando el mensaje claro
void descifrar_mensaje(const char *cifrado, char *salida) {
    int i;
    for (i = 0; cifrado[i] != '\0'; i++) {
        salida[i] = descifrar_caracter(cifrado[i]);
    }
    salida[i] = '\0';
}

// Libera la memoria del árbol BST
void liberarBST(NodoBST *raiz) {
    if (raiz == NULL) return;
    liberarBST(raiz->izq);
    liberarBST(raiz->der);
    free(raiz);
}

// Programa principal
int main(void) {

    const char *mensaje_cifrado =
        "(/-.-4%(+28.%#+2/($(6(#(3(8%.-/2(+(/(6.(";

    int L = 0;
    clock_t inicio, fin;
    NodoBST *raiz = NULL;
    Frec frecuencias[MAX_NODOS];
```

```
    int nNodos = 0;

    // FASE 1: frecuencias con BST
    inicio = clock();
    raiz = analizar_frecuencia(mensaje_cifrado, &L);
    fin = clock();

    // Paso el árbol a un arreglo y lo ordeno por frecuencia
    volcarBST(raiz, frecuencias, &nNodos);
    ordenarPorFrecuencia(frecuencias, nNodos);
    imprimirFrecuencias(frecuencias, nNodos, L);

    double tiempo_seg = (double)(fin - inicio) / CLOCKS_PER_SEC;
    printf("Tiempo aproximado de analizar_frecuencia: %.8f s\n", tiempo_seg);
    printf("Complejidad teorica de analizar_frecuencia: O(L * log N)\n\n");

    // FASE 2: buscar la constante A
    // Supongo que el símbolo más frecuente '(' es la letra 'e'
    double A_encontrada = buscar_constante_A('e', '(');

    printf("=== FASE 2: BUSQUEDA HEURISTICA DE A ===\n");
    if (A_encontrada > 0.0) {
        printf("A encontrada ~ %.5f\n", A_encontrada);
        printf("Valor teorico esperado (phi - 1) ≈ 0.61803\n\n");
        construir_patron_hash(A_encontrada);
    } else {
        printf("No se pudo encontrar una A consistente con el par (e -> '(').\n\n");
    }

    // FASE 3: descifrado final
    char mensaje_descifrado[256];

    descifrar_mensaje(mensaje_cifrado, mensaje_descifrado);

    printf("=== FASE 3: RUPTURA FINAL DEL CODIGO ===\n");
    printf("Mensaje descifrado ( letra por letra):\n");
    printf("%s\n\n", mensaje_descifrado);

    printf("Interpretacion en espanol (corrigiendo palabras):\n");
    printf("el ataque nocturno debe ser ejecutado en el este\n");

    // Libero la memoria que usó el árbol
    liberarBST(raiz);

    return 0;
}
```

Mensaje cifrado:

“(/-.-4%(+28.%#+2/($(6(#(3(8%.-/2(+(/(6.(”

Mensaje descifrado:

“elataquenocturnolebeserejecutaloeneleste”