Algoritmos y Estructuras de Datos I

Práctica 2
Evaluación continua temas 2 y 3

Curso 2023-2024

Grupo de prácticas: [2.*3*]

Autores:
Ortiz García, Juan Jesús | juanjesus.ortizg@um.es
Vicente Pérez, Gonzalo | gonzalo.vicentep@um.es
Usuario juez on-line:
B79

Índice

1. Análisis y diseño del problema

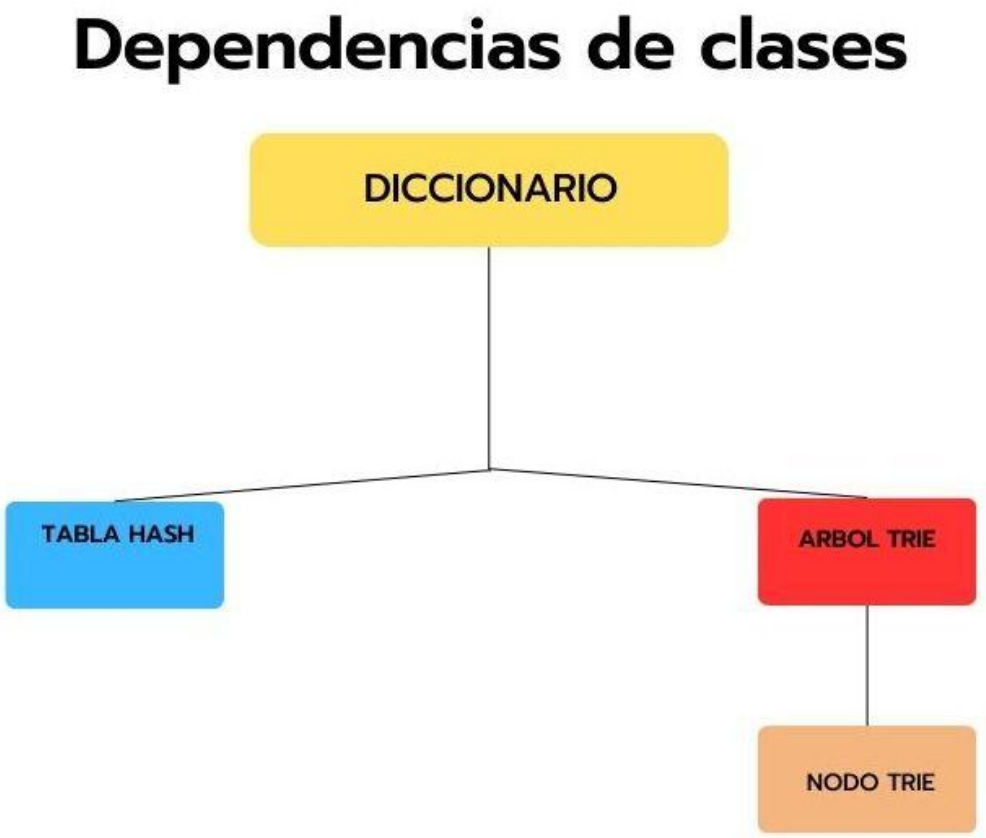
1.1. Descripción de las clases

Para la resolución de la práctica hemos definido un total de 4 clases.
-NodoTrie: Es la clase en la que se basa el árbol, por lo que es una clase interna del mismo . Contiene las operaciones básicas de un nodo, como son: insertar un caracter, consultarlo, poner la marca de fin, quitarla y comprobar si está. Al haber implementado el árbol con listas, el nodo tiene 3 campos: un caracter, y dos punteros, uno al siguiente nodo de la lista, y otro a su hijo, si tuviera. También hemos implementado el destructor y dos constructores, uno por defecto que pone los punteros a nulo, y otro con 3 argumentos para los 3 campos.

-ArbolTrie: Clase que se apoya esencialmente en el NodoTrie, usando sus operaciones. Está compuesta por un puntero a NodoTrie (la raíz) y una variable entera que guarda el número de elementos. En cuanto a las operaciones, están definidos su constructor y su destructor, y operaciones necesarias para el diccionario, como la inserción de una palabra, la consulta, el vaciado del árbol, la consulta del número de elementos, y el juego implementado con árboles (alargapalabras).

-TablaHash: Esta clase implementa nuestra tabla de dispersión. En nuestro caso se trata de una tabla con dispersión abierta, y tamaño fijo, por lo que está implementada como un array de listas de string. En la clase se encuentra la función de dispersión, las funciones de inserción, consulta y vaciado, y el juego implementado con la tabla (salsa césar).

-Diccionario: Clase en la que se basa el programa. Contiene un árbol y una tabla, las funciones de inserción, consulta, y vaciado, y los dos juegos.



1.2. Descripción de los módulos

En nuestro proyecto encontramos los siguientes módulos:

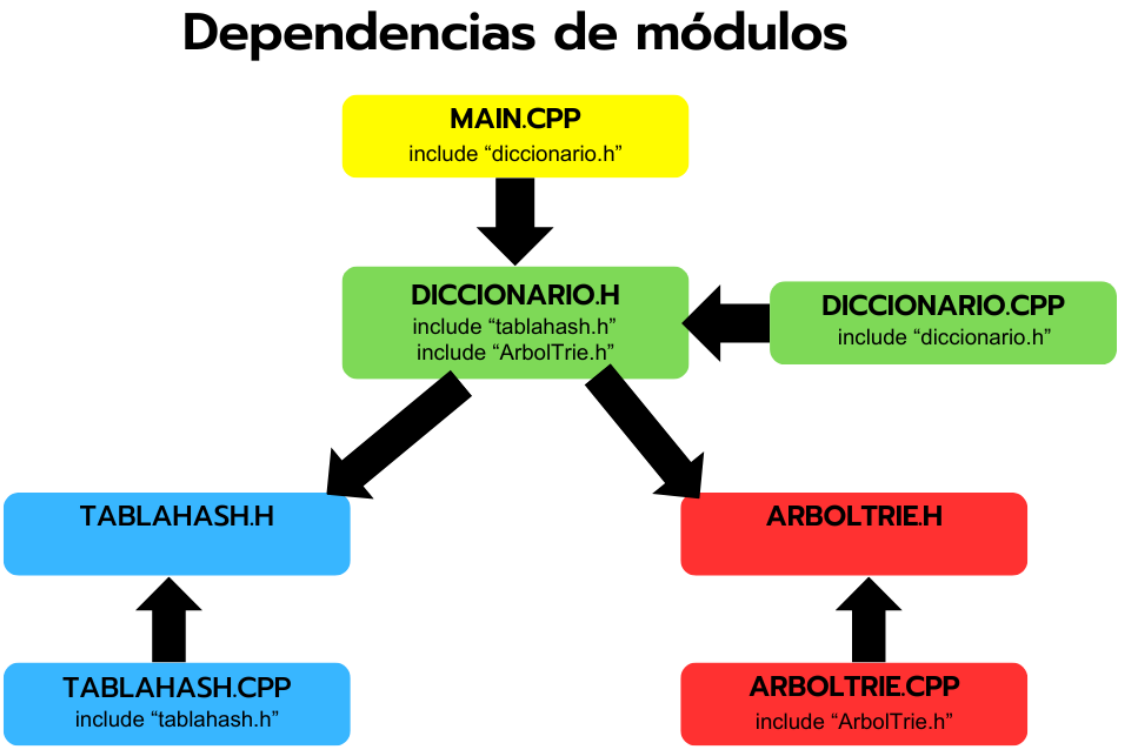
En **main.cpp** encontramos el funcionamiento base del programa, en el que encontramos la función de normalización, junto con la inserción, vaciado, búsqueda y los dos comandos opcionales (los juegos césar y juanagrama). El intérprete se encarga de gestionar las llamadas a las diferentes funciones por parte del usuario, reconociendo el comando introduciendo y llamando a la función correspondiente.

En **ArbolTrie.cpp** se aloja la lógica de implementación del Árbol Trie. Hemos definido una clase NodoTrie la cual contiene como clase amiga a la clase ArbolTrie (fue necesario para poder implementar el juego de alargapalabras). Por tanto, ArbolTrie contiene: a dicha clase NodoTrie, como atributos un puntero a NodoTrie, que es la raíz, y una variable entera donde se guarda el número de elementos, el constructor y destructor del árbol, así como la inserción, consulta, el vaciado, el número de elementos del árbol y el juego implementado.
Por otra parte, NodoTrie contiene la estructura del propio nodo, el constructor, el destructor y las operaciones. Como hemos escogido la implementación con listas, un nodo consiste en un caracter, y dos punteros, uno al siguiente nodo de la lista, y

otro al hijo, si tuviera. Las operaciones incluidas se refieren a aquellas sobre el propio nodo (es decir, manejan caracteres). Encontramos una operación de inserción, consulta, manejo de la marca final (poner, quitar y comprobar si existe) y el número de elementos.

Diccionario.cpp contiene, análogamente, la implementación del diccionario. En esta clase encontramos los atributos principales que definen al diccionario: la tabla de dispersión y el árbol trie. Como operaciones públicas tenemos el constructor y las típicas de un diccionario: vaciado, inserción, consulta y número de elementos, junto a los dos juegos opcionales.

tablahash.cpp es el archivo donde se aloja la lógica de funcionamiento de la tabla. En este fichero encontramos, como atributos privados, al array de listas sobre el que está construida nuestra tabla (de dispersión abierta y estática) y el número de elementos. Las operaciones que tenemos son la función de dispersión para dirigir en las cubetas correspondientes a las palabras; las funciones de inserción, consulta, vaciado y número de elementos; y una serie de operaciones auxiliares para el manejo de las diéresis, junto al césar.



1.3. Makefile y sus dependencias

```
a.out: main.o diccionario.o tablahash.o ArbolTrie.o
   g++ main.o diccionario.o tablahash.o ArbolTrie.o -o a.out -g

main.o: main.cpp diccionario.h tablahash.h ArbolTrie.h
   g++ main.cpp -c -g

diccionario.o: diccionario.cpp diccionario.h tablahash.h ArbolTrie.h
   g++ diccionario.cpp -c -g

ArbolTrie.o: ArbolTrie.cpp ArbolTrie.h
   g++ ArbolTrie.cpp -c -g

tablahash.o : tablahash.cpp tablahash.h
   g++ tablahash.cpp -c -g
```

Como se puede ver, el Makefile contiene todas las dependencias mostradas en la imagen.

1.4. Tabla de dispersión

Hemos optado por usar una tabla de dispersión abierta sin reestructuración. Nos hemos decantado por no hacerla dinámica porque hemos considerado que la cantidad de memoria no es tan inasumible como para tener que dinamizarla, y ganamos bastante más en eficiencia al no tener que hacer una redistribución de toda la tabla. De todas formas, definimos el tamaño como una constante, siendo fácil ajustar el número de cubetas de la tabla, aunque nosotros siempre usamos números primos, ya que resulta útil para la función de dispersión (el número de cubetas es 9839, pues para el caso de prueba grande del 310, vimos que subir más de eso empezaba a no generar cambios en el tiempo apreciables).También tuvimos en cuenta usar una dispersión cerrada, pero los tiempos en eficiencia eran bastante inferiores al no encontrar funciones de redispersión lo suficientemente buenas como para evitar colisiones frecuentes. Al no ser una tabla dinámica, no tenemos que liberar nada, pues no reservamos memoria dinámica.

1.5. Función de dispersión

```
int TablaHash::hashfunction(string palabra)
{
   palabra = quitar_dieresis(palabra);
   unsigned long long indice = 0;
   int i = 0;
   while (i < palabra.size() - 1)
   {

       indice += 5381 * (int)palabra[i] * 1217 * (int)palabra[i + 1];
       i += 2;
   }
   if (i < palabra.size())
   {
       indice += 5381 * (int)palabra[i];
   }
   return (indice % TAM);
}
```

La función de dispersión que hemos usado se basa en el método de plegado. Vamos guardando la suma de los caracteres 2 a 2, siendo el primero multiplicado por 5381 y el segundo por 1217. Estos números no fueron los primeros que probamos, empezamos con números primos más pequeños, como el 53 o el 71, pero vimos que estos producían una mejor distribución de los elementos y en consecuencia menor tiempo de ejecución. Al ser números primos considerablemente grandes, decidimos que la variable `indice` donde guardábamos dicha suma, fuera `unsigned long long`, para asegurarnos que no teníamos problemas de desbordamiento o números negativos. En caso de número impar de caracteres, procesa el último caracter de forma individual. Finalmente, aplica el módulo del tamaño de la tabla hash.

Hemos probado más funciones como las siguientes (djb2), pero vimos que el resultado era muy similar al de la nuestra, y además la tuvimos que modificar de cara al juego del césar, lo que nos hizo descartarla.

```
unsigned long hash = 5381; // Un número primo grande como valor inicial
   for (char c : palabra)
   {
       hash = ((hash << 5) + hash) + c; // hash * 33 + c
   }
   return hash % 1187; //tamaño usado para el 200
}
```

1.6. Resolución del juego con tabla Hash (Salsa César - 203)

Para realizar este ejercicio, primero vamos a explicar unas cuantas funciones auxiliares que tuvimos que definir de cara al correcto manejo de la diéresis.

`string quitar_dieresis(string palabra).` Esta función elimina la diéresis en caso de que la palabra la tenga. Por ejemplo, si introducimos “CIGÜEÑA”, devuelve “CIGUEÑA”.

`bool tiene_dieresis(string palabra).` Esta función devuelve *true* en caso de que la palabra contenga algún caracter ‘Ü’, y *false* en caso contrario.

`bool tiene_u(string palabra).` Esta función devuelve *true* si la palabra contiene algún caracter ‘U’, *false* en caso contrario.

`list<string> consulta_dieresis(string palabra)` . Esta función lo que hace es entrar a la cubeta que corresponde a la palabra (que no tendrá ya diéresis), e ir recorriéndola comprobando palabra por palabra si son iguales (eliminando las diéresis, si las tuvieran). Si al transformar la ‘Ü’ por ‘U’ en ambas, son iguales, se insertan en la lista que luego se devuelve como resultado. Para que esto funcione correctamente, y el juego tome como iguales a ambos caracteres, tuvimos que modificar la función de dispersión de manera que primero quita la diéresis, y luego calcula el índice. Así, nos aseguramos que CIGUEÑA y CIGÜEÑA irían a la misma cubeta, y por tanto, el césar iba a encontrar ambas, tomando como iguales ambos caracteres.

Una vez explicadas estas 4 funciones auxiliares, pasamos a explicar cómo resolvimos dicho juego. Primero, se guarda en una variable (`cesar`) la palabra sin diéresis, luego declaramos una lista de string (`soluciones`) donde irán los cifrados cesar de la palabra introducida que estén en el diccionario, otra variable en la que se guardará cada paso del césar (`nueva`) y una variable (`copia`) en la que se irá calculando cada césar. Así, se entra al bucle for, que se ejecuta 27 veces, pues hay 27 caracteres en el alfabeto español contando la Ñ, `copia` se pone a la cadena vacía, y se va calculando su césar, teniendo cuidado con el manejo de la Ñ. Una vez el césar está en `copia`, se comprueba si tiene alguna ‘U’, y en ese caso, se llama a `consulta_dieresis`, se ordena la lista obtenida, y se va insertando palabra a palabra en `soluciones`. En caso de no tener ‘U’, se comprueba si el césar está en la tabla, y en ese caso se inserta en `soluciones`. `nueva` toma el valor de `copia`, lo que hace más fácil el cálculo del siguiente paso, pues salvo los casos especiales, es sumar un caracter, y se vuelve a repetir el proceso hasta terminar el bucle for. Una vez terminado, se ordena con `sort` la lista de soluciones, si está vacía se devuelve “”, y en caso contrario se inserta el espacio que debe ir detrás de “->”, y se va insertando en la variable `texto` cada palabra, separándolas por un espacio, y borrando el espacio detrás de la última para evitar errores de presentación en Mooshak.

1.7. Árbol empleado

Hemos usado un Árbol Trie porque hemos considerado que se ajusta perfectamente a lo pedido, y eran los que iban a funcionar mejor para el juego del alargapalabras, pues, al fin y al cabo el juego se basa en recorrer el prefijo y luego buscar la palabra más larga. Los árboles trie son bastante sencillos de implementar y tienen operaciones que se adaptan perfectamente a los requerimientos de un diccionario, pues son muy útiles para guardar palabras. El recorrido es sencillo y muchos ejercicios pueden resolverse simplemente tocando un poco el propio recorrido. Ofrece los mejores valores de eficiencia a la hora de trabajar con cadenas, por lo que aquí lo tuvimos bastante claro.

1.8. Definición del tipo árbol y tipo nodo

Optamos por usar una estructura de Árbol Trie con listas. Inicialmente, pensamos que sería bueno y sencillo hacerla con arrays, pero al implementarlo vimos que la cantidad de memoria que usamos era desorbitada para Mooshak (llegamos a reservar más de 100MB), y aunque conseguimos reducirla, decidimos cambiar a la representación con listas la cual es más eficiente en términos de memoria.

El árbol está compuesto por nodos, de los cuales vamos primeramente a definir su estructura. Un nodo está compuesto por un caracter y dos punteros, uno llamado `siguiente` que apunta al nodo hermano y otro llamado `puntero` que apunta al hijo. Tenemos dos constructores, uno sin argumentos que inicializa ambos punteros a nulo y pone como caracter ‘?’, como marca de cabecera, y otro que recibe 3 argumentos e inicializa cada campo del nodo a su argumento correspondiente. El tipo árbol está compuesto por el número de elementos y un puntero a NodoTrie `raiz`. De este modo, a la hora de construir un árbol se manejarán operaciones de inserción que directamente referenciarán a las inserciones en nodos de manera recursiva.

1.9. Resolución del juego con árboles (Alargapalabras - 307)

Para realizar este ejercicio hemos definido una función auxiliar, llamada ***buscarPalabraMasLarga*** que, como su nombre indica, busca la palabra más larga desde un nodo, y llevando ya el prefijo pasado como argumento. Esta función contiene la base lógica del ejercicio, pues una vez se recorre el prefijo, habrá que llamarla.

A esta función se le pasa como parámetros un nodo, que es el nodo de inicio sobre el que se realiza la búsqueda, y una variable `palabraActual`, que indica el prefijo a partir del cual se busca la más larga. Se crea también una lista vacía de cadenas, cuya lógica cobra sentido a la hora de ejecutar el orden alfabético, y una variable entera `longmax`. A continuación, se tienen los dos casos base: si el nodo es nulo, se devuelve `palabraActual`. Si hay marca y no hay más hermanos (`nodo->siguiente` es nulo), devuelve también `palabraActual`.

Hecha esta comprobación, se declara una variable de tipo string que va almacenando la palabra más larga que se encuentra, y continúa el proceso de búsqueda.
Posteriormente, la función itera sobre los nodos hermanos del nodo actual, omitiendo la cabecera, lo que se hace asignando el siguiente del nodo pasado como parámetro a una variable temporal llamada `tmp`. En cada iteración, se hace una llamada recursiva a `buscarPalabraMasLarga` con el puntero al siguiente nodo (`tmp->puntero`) y la palabra actual concatenada con el caracter del nodo (`palabraActual + tmp->car`). Esto permite explorar todas las posibles combinaciones de palabras que se pueden formar desde el nodo actual.

La palabra encontrada en cada iteración recursiva se compara con la palabra más larga actual (`palabraLarga`). Si la longitud (calculada con una función especial que no cuenta la Ñ y la Ü como dos caracteres) de la nueva palabra encontrada es mayor o igual a la de `palabraLarga`, entonces se actualiza el valor de `longmax` con la longitud de esta nueva palabra. Además, se añade la nueva palabra más larga a la lista. Se avanza al siguiente hermano, y se repite el proceso en las siguientes iteraciones.

Al finalizar el recorrido por todos los nodos sucesores, se tiene una lista de palabras (`palabras`) que contienen todas las palabras más largas encontradas, que posteriormente se ordena para devolver la más pequeña alfabéticamente hablando cuya longitud coincida con `longmax`.

Se hace entonces la llamada a **`alargapalabras`**, que esencialmente se encarga de recorrer el árbol hasta llegar al final del prefijo (en caso de que no esté el prefijo, se devuelve cadena vacía) y de llamar a **`buscarPalabraMasLarga`,** que recorrerá todos los hijos del nodo que contiene el último caracter del prefijo, en busca de la palabra más larga.

1.10. Liberación del árbol

En la clase `NodoTrie`, cada nodo tiene un destructor que se encarga de liberar la memoria. Este destructor, `NodoTrie::~NodoTrie()`, se activa cuando un nodo es eliminado. Dentro del destructor, hay dos instrucciones de eliminación: `delete this->puntero` y `delete this->siguiente`. Estas instrucciones liberan la memoria de los nodos a los que apuntan *puntero* y *siguiente*, respectivamente, pues son los dos campos a los que se le puede asignar memoria dinámica.

El destructor del árbol `ArbolTrie::~ArbolTrie()` simplemente hace `delete raiz`, pues el árbol es un puntero a nodo, que es lo que hay que eliminar. El método *`vaciar()`* de la clase `ArbolTrie` complementa esta gestión de memoria. Al llamar a `delete raiz`, se libera la raíz, y por consiguiente todos sus nodos. Después de esta eliminación, `vaciar()` reinicializa el Trie al crear un nuevo nodo raíz y resetear el contador de elementos *nElem* a cero.

Al principio pensamos que tendríamos que hacer el destructor del árbol como un recorrido recursivo, borrando las hojas y posteriormente subiendo por el árbol. Pero vimos que no hacía falta, y comprobamos con *Valgrind* que estábamos liberando todo correctamente.

1.11. Variables globales

Como única variable global en el main, usamos `Diccionario d`, y evitamos el uso de otras variables globales mediante la modularidad y las dependencias.

2. Listado del código

Nombre Fichero	Descripción
Makefile	Fichero Makefile para la compilación del proyecto
diccionario.h	Fichero de cabecera para la clase Diccionario
diccionario.cpp	Fichero de implementación para la clase Diccionario
tablahash.cpp	Fichero de implementación para la clase TablaHash
tablahash.h	Fichero de cabecera para la clase TablaHash
ArbolTrie.cpp	Fichero de implementación para la clase ArbolTrie
ArbolTrie.h	Fichero de cabecera para la clase ArbolTrie
main.cpp	Fichero de implementación del programa principal

3. Prueba de aceptación en Mooshak

#	País	Equipo	Problemas																				Total	Puntos
			001	002	003	004	200	201	202	203	204	205	206	207	300	301	302	303	304	305	306	307	310	
1		G2 ORTIZ GARCIA, JUAN JESUS	1 (4)	1 (9)	2 (3)	3 (1)	4 (7)			3 (4)					4 (11)							3 (3)	12 (4)	9	33

Resultado	Problemas																				Total
	001	002	003	004	200	201	202	203	204	205	206	207	300	301	302	303	304	305	306	307	310	
Accepted	2	2	1	1	1			2					1							1		11
	4.3%	4.3%	2.2%	2.2%	2.2%			4.3%					2.2%							2.2%		23.9%

4. Informe de desarrollo

Proyecto: Wild Word Games				**Fecha de inicio:** 06/10/2023	
Programadores: Juan Jesús Ortiz García y Gonzalo Vicente Pérez				**Fecha de fin:** 30/11/2023	
Día/Mes	Análisis	Diseño	Implement.	Validación	TOTAL
06/10	5	10	-	-	15
07/10	15	15	-	-	30
08/10	-	20	30	-	50
09/10	-	-	-	-	-
10/10	-	-	-	-	-
11/10	-	30	10	10	50
12/10	-	-	-	-	-
13/10	10	20	-	-	30
14/10	-	-	-	-	-
15/10	-	-	-	-	-
16/10	-	5	30	40	75
17/10	-	-	-	30	30
18/10	-	-	-	-	-
19/10	5	10	15	40	70
20/10	-	-	-	-	-
21/10	-	-	-	-	-
22/10	-	-	-	-	-
23/10	15	10	-	-	25
24/10	-	-	-	-	-
25/10	20	20	30	60	130
26/10	-	-	-	75	75

27/10	-	-	-	45	45
28/10	-	-	-	-	-
29/10	-	-	-	-	-
30/10	30	30	-	-	60
31/10	-	-	-	-	-
1/11	-	-	-	-	-
2/11	-	30	20	30	80
3/11	-	-	-	-	-
4/11	-	-	-	-	-
5/11	-	-	-	-	-
6/11	-	-	-	-	-
7/11	-	-	-	60	60
8/11	-	-	-	30	30
9/11	-	-	-	60	60
10/11	15	20	20	-	55
11/11	-	-	-	-	-
12/11	-	-	-	-	-
13/11	-	-	-	-	-
14/11	-	-	30	-	30
15/11	-	-	15	30	45
16/11	-	-	-	-	-
17/11	-	-	-	-	-
18/11	-	-	-	-	-
19/11	-	-	-	-	-
20/11	-	-	-	-	-
21/11	-	-	-	-	-
22/11	-	-	-	-	-
23/11	-	-	-	-	-

24/11	-	-	-	120	120
25/11	-	-	-	-	-
26/11	15	45	45	120	225
27/11		20	30	90	140
28/11	20	30	60	90	200
29/11	15	30	30	120	195
30/11	-	30	30	90	150
TOTAL (minutos)	165	375	395	1140	2075
MEDIAS (porcentaje)	7.95%	18.07%	19.04%	54.94%	100%

En total hemos tardado 34 horas y 35 minutos. Respecto a la coordinación del trabajo, no nos hemos repartido los ejercicios individualmente, sino que hemos ido a la vez con todos los ejercicios. Normalmente creábamos una sesión compartida de Visual Studio para tener los dos el mismo programa en frente y trabajamos en conjunto para resolver cada uno de los ejercicios, aportando cada uno sus ideas, pero trabajando casi siempre en paralelo. Puntualmente hacíamos trabajo individual cada uno en su casa, aunque lo normal ha sido avanzar juntos, especialmente en las fases de análisis, diseño e implementación.

5. Conclusiones y valoraciones personales

Gonzalo Vicente Pérez:
En mi opinión el trabajo ha sido un desafío y algo complicado, especialmente a partir del ejercicio 200. Cuando hicimos los 3 primeros ejercicios no encontré del todo la relación con la asignatura de AED en los programas que se nos pedían, pero a partir del ejercicio 004 sí que me di cuenta para qué servía lo que habíamos estado haciendo y ya vi como todo se empezaba a poner más difícil. Respecto a la implementación del diccionario mediante tablas hash, me resultó compleja pero tras dedicarle el tiempo necesario acabé entendiendo el uso de estas tablas en C++, y el ver las tablas de dispersión aplicadas a un programa directamente me ayudó a entender la teoría, algo bastante importante para mí. En el ejercicio con tablas de dispersión, el césar, tuvimos el algoritmo básico más o menos claro, pero tuvimos que dedicar muchísimo tiempo al tratamiento de las “ü” y de las “ñ”, prácticamente la totalidad del tiempo que tardamos en hacer la función. Este es uno de los aspectos que menos me ha gustado en general del proyecto, porque creo que el tratamiento especial de estos caracteres se aleja un poco del propósito de la asignatura, y realmente no era una cuestión trivial, sino que en muchos casos había que dedicar mucho tiempo a problemas relacionados con este aspecto. La implementación de los Árboles Trie sí que me pareció más liosa, porque no fue tan fácil implementar las operaciones en pseudocódigo de las diapositivas como pensaba, especialmente las de listas. No obstante, el ejercicio con árboles me pareció bastante más fácil que el de las tablas de dispersión. En general, ha sido un proyecto con el que he llegado a sufrir bastante por sacar adelante pero que una vez hecho ha dado una satisfacción tremenda, y probablemente ha sido el mayor desafío ante el que me he enfrentado de momento en la carrera.

Juan Jesús Ortiz García:
El proyecto me ha supuesto un reto bastante grande. Al inicio quise llevar todo al día, y pudimos hacerlo con los primeros ejercicios, pero a partir de ahí ya fue más difícil. Los primeros ejercicios eran bastante sencillos, y no tuve problema en hacerlos ni en razonarlos, y cuando empezamos a hacer el menú ya la cosa empezó a cambiar. Con el ejercicio 200 fue entretenido buscar la función de dispersión que mejor nos fuera y eso es algo que me gustó, pero luego en el ejercicio 203, que fue el que escogimos, tuvimos que estar mucho tiempo pensando en cómo tratar las ñ y las ü. Seguramente esto fue lo que más me costó y lo que menos me gustó, porque hemos perdido muchísimo tiempo intentando solucionar los problemas que nos daban, y aunque podíamos haber cambiado de problema, como ya teníamos todo el resto del algoritmo hecho, decidimos seguir intentándolo. El problema con las ñ y las ü es que a mi parecer no tiene demasiada relación con las estructuras de datos y si hubiésemos usado un diccionario con caracteres en inglés podríamos haber tardado la mitad en hacer el proyecto. La solución que pensamos de calcular todos los césar

en pasos de 1 en 1, comprobar si estaban en el diccionario, e insertarlos, era bastante buena, y la vi bastante rápido, pero claro, la Ü no la tratábamos bien y tuvimos que hacer modificaciones en la tabla para solucionarlo. La implementación del árbol me ha gustado algo más aunque también tuvimos que cambiarla, porque al principio quisimos hacerla con arrays y se nos desbordaba la memoria. En el ejercicio de árboles que hemos hecho no ha habido tanto problema con los caracteres especiales, aunque también hemos tenido que tocar un par de cosas para que funcionara bien. En general, me ha gustado el proyecto y lo veo útil, aunque también me ha resultado complejo, sobre todo porque ha sido un proyecto bastante extenso, con implementaciones de tipos de datos que no había visto antes.