Documentación del proyecto

Buscador web

José Miguel Sánchez Almagro josemiguel.sancheza@um.es

Tabla de contenido

Análisis y diseño del problema

Clases

Se han definido las siguientes clases:

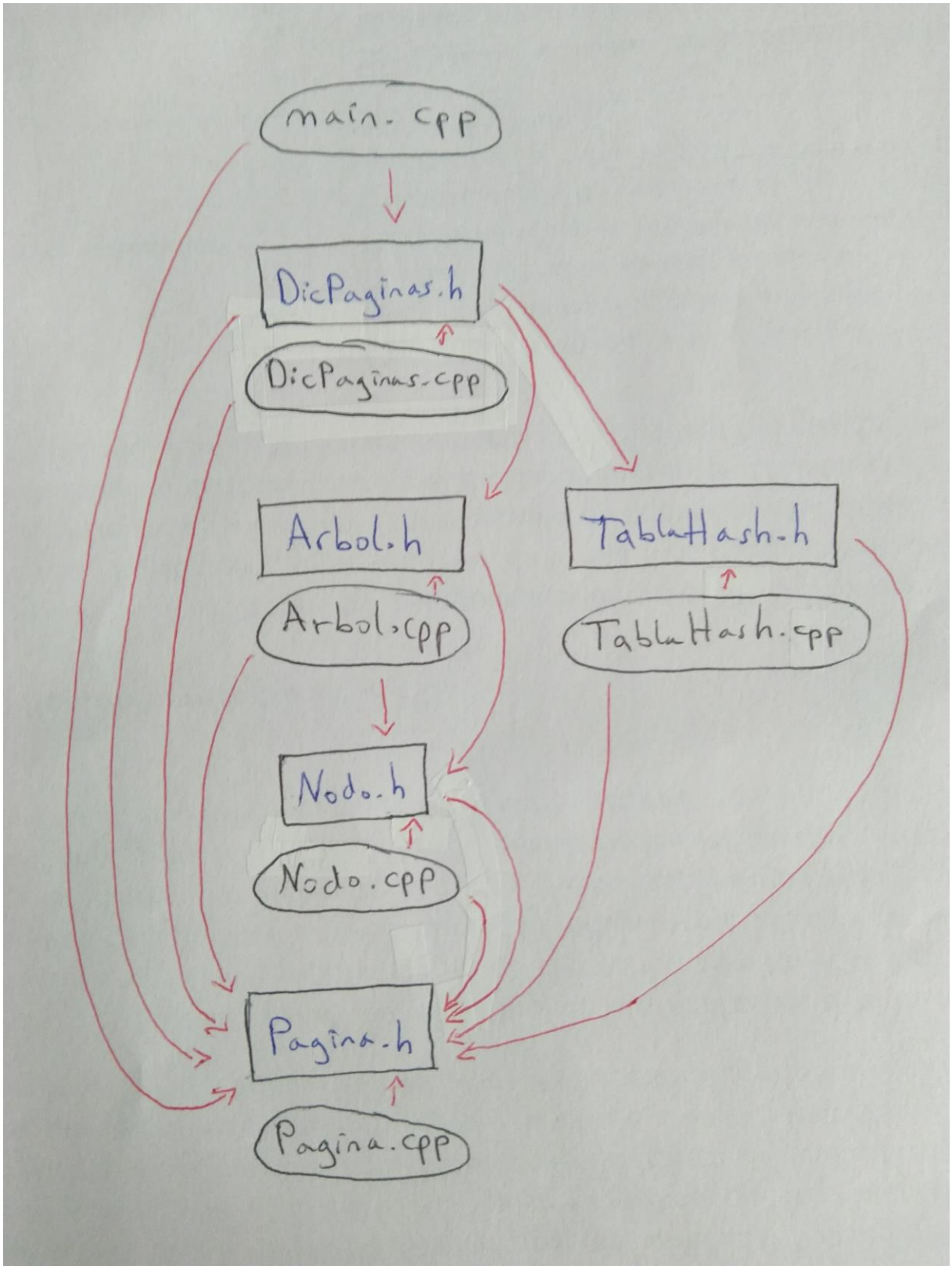
- Main: Es la clase principal del programa y contiene el intérprete de comandos.
- DicPaginas: Es el diccionario de páginas. Se encarga de llamar a la tabla de dispersión o al árbol según corresponda.
- Pagina: Es un tipo página que será usado en el resto de las clases (equivalente a una página web).
- TablaHash: Es una tabla de dispersión abierta que contendrá las páginas que se vayan almacenando.
- Nodo: Define el tipo Nodo que usará el árbol y las funciones que necesite.
- Arbol: Es un árbol Trie que usa los nodos definidos anteriormente.

Módulos

Se han definido los siguientes módulos:

- main.cpp: Contiene el intérprete de comandos, encargado de ofrecer la interfaz al usuario para realizar las operaciones necesarias. Este se encarga de llamar al Diccionario de páginas.
- DicPaginas.h: Es el archivo posee las llamadas a las funciones de DicPaginas.cpp. Ofrece la interfaz a las clases que lo implementen.
- DicPaginas.cpp: Contiene las funciones de la clase DicPaginas, y que serán usadas por el archivo anteriormente descrito 'main.cpp'. Esta clase es la encargada de insertar y consultar sobre las páginas y palabras insertadas, y es la responsable de usar el Árbol o la Tabla de Dispersión según convenga.
- Pagina.h: Es el archivo posee las llamadas a las funciones de Pagina.cpp. Ofrece la interfaz a las clases que lo implementen.
- Pagina.cpp: Contiene las funciones de la clase Pagina, y que serán usadas por otras clases del proyecto. Esta clase define un tipo de objeto llamado Página, que posee los atributos 'Relevancia' (Entero que representa la relevancia de la página), 'url' (string con la dirección web de la página), 'titulo' (Título de la pagina).
- TablaHash.h: Es el archivo posee las llamadas a las funciones de TablaHash.cpp. Ofrece la interfaz a las clases que lo implementen.
- TablaHash.cpp: Implementa una tabla de dispersión abierta que contiene referencias a las páginas insertadas. Estas páginas no se insertan en la tabla de dispersión, sino que se crean dinámicamente. Esto es así para que al redimensionar la tabla las referencias a las páginas que tendrá el árbol no se pierdan. Además, la clase contiene otras funciones como buscar una palabra en la tabla.
- Nodo.h: Es el archivo posee las llamadas a las funciones de Nodo.cpp. Ofrece la interfaz a las clases que lo implementen.
- Nodo.cpp: Implementa la estructura que tendrán los nodos del árbol Trie que vamos a usar. Tendrán 4 atributos: Un char con la letra correspondiente a ese nodo; dos casillas con apuntadores a otros nodos, uno para el siguiente nodo y otro para su hijo; finalmente cada nodo contiene una lista con las referencias a páginas que corresponden a esa palabra. La lista solo contendrá referencias en los nodos 'marca de fin'.
- Arbol.h: Es el archivo posee las llamadas a las funciones de Arbol.cpp. Ofrece la interfaz a las clases que lo implementen.
- Arbol.cpp: Esta clase es un árbol Trie, con sus correspondientes métodos. Estos serán solamente 'buscar' e 'insertar'. Las operaciones que hay detrás de todo eso la realizan los nodos.

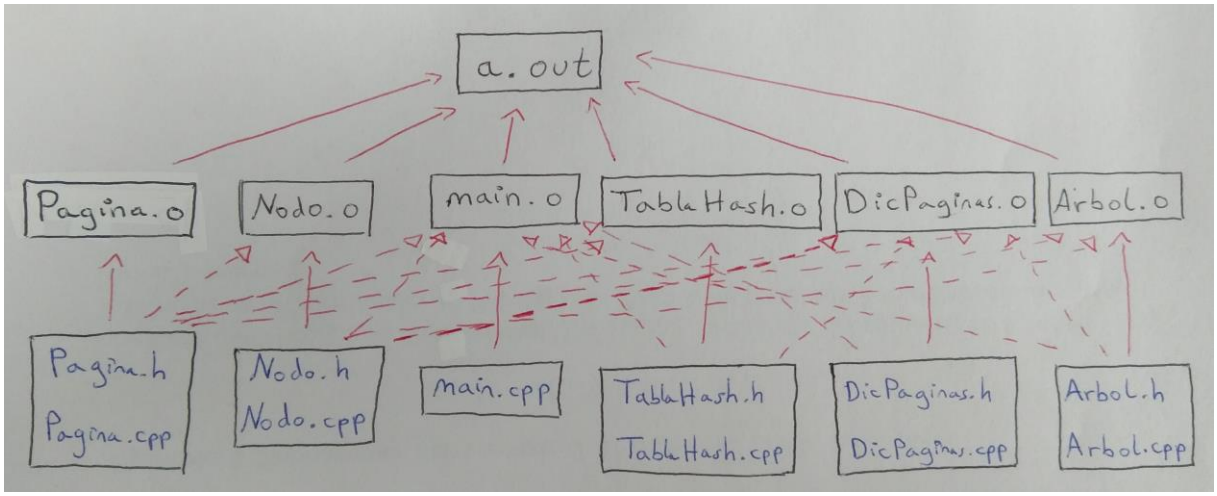
Representación gráfica de dependencias



Makefile

En este archivo tenemos que compilar todas las clases del proyecto al mismo tiempo y en el orden adecuado. Creamos un archivo objeto (.o) por cada clase. Este archivo se crea con los archivos de la propia clase (.h y .cpp), y con las cabeceras (.h) de las clases de las que depende (ya sean dependencias directas o indirectas). El archivo ejecutable final (.out) se crea compilando el archivo main.cpp con todos los .o compilados anteriormente.

Representación gráfica de dependencias Makefile



En el archivo comprimido del proyecto se incluye para poder ser revisado y consultado.

Normalización del texto

Se comprueba cada carácter del texto de entrada, realizando algunas operaciones si fuera necesario:

- Si es un carácter mayúsculas no se realiza nada.
- Si es el char asociado a los caracteres especiales se avanza un puesto y se analiza el siguiente. Una vez analizado y comprobado que carácter es, se sustituye por el corresponda.
- Si no es ninguno de estos dos casos no se realiza acción.

Finalmente se imprimen los caracteres que hayamos deducido en el orden correspondiente.

Tabla de dispersión y reestructuración

La tabla de dispersión elegida es abierta, y la función de dispersión usa suma posicional. La tabla se amplía cuando el número de elementos insertados en ella es mayor al doble del número de cubetas de la tabla, es decir, cuando con una dispersión perfecta, habría dos elementos por cubeta. El tamaño que tendrá la tabla en cada momento será siempre primo para facilitar la dispersión de los elementos, aunque en el caso concreto de esta práctica eso suponga una desventaja, ya que el coste de encontrar el siguiente número primo a uno dado es mayor que la ganancia obtenida al tener una tabla de dispersión de tamaño primo.

La redimensión de la tabla creará una nueva más amplia, insertando las referencias a las páginas que contenía la tabla anterior. Finalmente, la tabla antigua será eliminada. El tamaño de la nueva tabla será el siguiente número primo al doble del tamaño de la tabla anterior.

La elección de esta tabla de dispersión se debe a los requisitos del proyecto. Se indica que la eficiencia del proyecto debe primar en la rapidez en la realización de las operaciones, por lo tanto, una tabla de dispersión de abierta es más adecuada que una tabla de dispersión cerrada, ya que desperdicia más memoria a cambio de ofrecer mayor rapidez en las operaciones.

Ahora bien, la tabla no contiene las páginas que vamos insertando, sino referencias a ellas. Esto es así por la implementación del árbol en el proyecto. El árbol contendrá las referencias a las páginas adecuadas para cada palabra, y si las palabras se encuentran en nuestra tabla al realizar la redimensión (ampliación de la tabla) se perderían las

referencias. Para evitar esto las páginas que insertamos con el intérprete no se almacenan en la tabla, sino que se crean de manera dinámica (luego habrá que liberarlas) y se obtienen sus referencias para introducirlas en la tabla. De esta manera las páginas no se verán afectadas por los cambios realizados en la tabla, y las referencias en el árbol tampoco. Esta implementación sacrifica rapidez en las operaciones, pero es necesario para realizar la búsqueda por palabras con el árbol Trie implementado.

Función de dispersión

Dada una cadena de entrada, recorremos carácter a carácter dicha cadena. Para cada carácter, obtenemos su código ASCII, y lo sumamos a otras dos multiplicaciones. Estas multiplicaciones se realizan sobre la clave que tenemos en cada momento. De esta manera vamos obteniendo una clave cada vez más diferente a cada paso.

Liberación de la tabla de dispersión

Para la liberación de la tabla primero debemos recorrer posición a posición toda la tabla, y en cada posición, hay que recorrer cada lista, para ir liberando las páginas que se definieron anteriormente. Una vez realizado esto, solamente tendríamos que liberar la memoria de la tabla, y esta se encargaría de liberar la memoria de cada cubeta (las listas tienen su propio destructor por lo que este paso se realizaría de manera automática) y la tabla en sí.

Árbol

El árbol usado se trata de un árbol Trie. Este árbol es ampliamente elegido para la inclusión de palabras en el árbol, ya que resulta útil en esos casos. Cada nodo tiene un carácter, dos punteros (uno al nodo siguiente de ese nivel y otro para el hijo, que continuará la palabra si el carácter no es una marca de fin) y una lista.

La lista de cada nodo tendrá referencias a las páginas correspondientes. Solo los nodos que sean marcas de fin de palabras tendrán referencias de páginas en sus listas, no así el resto de los nodos, ya que cuando se llega a este tipo de nodo, significa que la palabra que estábamos recorriendo se encuentra insertada.

El árbol solo contiene un nodo cabecera que tiene el carácter vacío: ‘ ‘.

Al insertar una nueva página, se recorre su texto palabra a palabra. Si esa palabra no se encuentra en el árbol se inserta y se almacena una referencia en su lista apuntando a la página correspondiente. Si ya se encuentra insertada simplemente se insertará de manera ordenada la página en su lista.

Al principio de la realización de la práctica se optó por usar un Árbol AVL, ya que su eficiencia no era mala y se quería hacer algo diferente, pero se encontraron con numerosos problemas de implementación y una elevada dificultad. Por lo tanto, se volvió a realizar un análisis y se observó que el árbol Trie es el más eficiente en estos casos, por lo que se abandonó la implementación del Árbol AVL y se diseñó un Árbol Trie, que resultó ser una buena decisión ya que no se encontraron grandes problemas y ha mostrado un buen rendimiento.

Liberación del árbol

El árbol se libera ejecutando la destrucción de su nodo cabecera. Este ejecutará el destructor del nodo, que a su vez ejecuta destructores para sus punteros, y los nodos se irán eliminando de una manera parecida a la recursividad.

Variables globales

No se usan en el proyecto.

Listado del código

Programa principal:

```
#include <list>
#include <iostream>
#include "Pagina.h"
#include "DicPaginas.h"

using namespace std;

string normalizar(string entrada){
    string salida;
    for(unsigned int i = 0; i < entrada.length(); i++){
        if ((entrada[i] >= 'A') && (entrada[i] <= 'Z')){
            salida = (salida + char(tolower(entrada[i])));
        } else if (entrada[i] == char(0xC3)){
            i++;
            if ((entrada[i] == char(0xA1)) || (entrada[i] == char(0x81))) salida = salida + 'a';
            else if ((entrada[i] == char(0xA9)) || (entrada[i] == char(0x89))) salida = salida + 'e';
            else if ((entrada[i] == char(0xAD)) || (entrada[i] == char(0x8D))) salida = salida + 'i';
            else if ((entrada[i] == char(0xB3)) || (entrada[i] == char(0x93))) salida = salida + 'o';
            else if ((entrada[i] == char(0xBA)) || (entrada[i] == char(0xBC)) || (entrada[i] == char(0x9A)) || (entrada[i] == char(0x9C))) salida = salida + 'u';
            else if ((entrada[i] == char(0xB1)) || (entrada[i] == char(0x91))) salida = salida + char(0xC3) + char(0xB1);
            else salida = salida + char(0xC3) + entrada[i];
        } else {
            salida = (salida + entrada[i]);
        }
    }
    return salida;
}

void insertar(DicPaginas &dic){

    string comodin, url, titulo, entrada_palabra;
    int relevancia = 0;
    int n_palabras = 0;

    cin >> relevancia;

    getline(cin, comodin);

    getline(cin, url);

    getline(cin, titulo);

    Pagina p(relevancia, url, titulo);

    Pagina* ref = dic.insertar(p);

    while ((cin >> entrada_palabra)&&(normalizar(entrada_palabra) != "findepagina")){
        n_palabras++;
        dic.insertar(normalizar(entrada_palabra), ref);
    }

    cout << dic.numPaginas() << ". " << url << ", " << titulo << ", Rel. " << relevancia << endl;
```

```
52      cout << dic.numPaginas() << ". " << url << ", " << titulo << ", Rel. " << relevancia << endl;
53      cout << n_palabras << " palabras" << endl;
54  }
55
56
57  void busquedaPalabra(DicPaginas &dic){
58      string palabra;
59      int resultados = 0;
60
61      cin >> palabra;
62      palabra = normalizar(palabra);
63
64      cout << "b"  << " " << palabra << endl;
65
66      list<Pagina*> palabras = dic.buscar(palabra);
67      list<Pagina*>::iterator iter = palabras.begin();
68
69      for (int i = 0; i < palabras.size(); i++){
70          Pagina* actual = *iter;
71          cout << (i + 1) << ". " << actual->getUrl() << ", " << actual->getTitulo() << ", Rel. " << actual->getRelevancia() << endl;
72          iter++;
73      }
74
75      cout << "Total: " << palabras.size() << " resultados" << endl;
76  }
77
78  void busquedaUrl(DicPaginas &dic){
79      string url;
80      int resultados = 0;
81
82      cin >> url;
83      cout << "u" << " " << url << endl;
84
85      Pagina* pagina = dic.consultar(url);
86      if(pagina == NULL){
87          resultados = 0;
88      } else {
89          cout << "1. " << url << ", " << pagina->getTitulo() << ", Rel. " << pagina->getRelevancia() << endl;
90          resultados = 1;
91      }
92
93
94      cout << "Total: " << resultados << " resultados" << endl;
95  }
96
97  void busquedaAnd(DicPaginas &dic){
98      string a;
99      int resultados = 0;
100
101     getline(cin, a);
102     a = normalizar(a);
103
```

```
100
101     getline(cin, a);
102     a = normalizar(a);
103
104     cout << "a"  << " " << a << endl;
105     cout << "Total: " << resultados << " resultados" << endl;
106 }
107
108 void busquedaOr(DicPaginas &dic){
109     string o;
110     int resultados = 0;
111
112     getline(cin, o);
113     o = normalizar(o);
114
115     cout << "o"  << " " << o << endl;
116     cout << "Total: " << resultados << " resultados" << endl;
117 }
118
119 void busquedaPrefijo(DicPaginas &dic){
120     string prefijo;
121     int resultados = 0;
122
123     cin >> prefijo;
124     prefijo = normalizar(prefijo);
125
126     cout << "p" << " " << prefijo << endl;
127     cout << "Total: " << resultados << " resultados" << endl;
128 }
129
130
131 void interprete(char comando, DicPaginas &dic){
132     if (comando == 'i') insertar(dic);
133     else if (comando == 'u') busquedaUrl(dic);
134     else if (comando == 'b') busquedaPalabra(dic);
135     else if (comando == 'a') busquedaAnd(dic);
136     else if (comando == 'o') busquedaOr(dic);
137     else if (comando == 'p') busquedaPrefijo(dic);
138 }
139
140 int main(void){
141     DicPaginas dic;
142     char comando;
143     while ((cin >> comando)&&(comando != 's')){
144         interprete(comando, dic);
145     }
146     if (comando == 's') cout << "Saliendo..." << endl;
147     return 0;
148 }
```

Makefile:

```
a.out: Pagina.o TablaHash.o Nodo.o Arbol.o DicPaginas.o main.o
	g++ Pagina.o TablaHash.o Nodo.o Arbol.o DicPaginas.o main.o

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

TablaHash.o: Pagina.h TablaHash.h TablaHash.cpp
	g++ -c TablaHash.cpp

Nodo.o: Pagina.h Nodo.h Nodo.cpp
	g++ -c Nodo.cpp

Arbol.o: Pagina.h Nodo.h Arbol.h Arbol.cpp
	g++ -c Arbol.cpp

DicPaginas.o: Pagina.h TablaHash.h Nodo.h Arbol.h DicPaginas.h DicPaginas.cpp
	g++ -c DicPaginas.cpp

main.o: Pagina.h TablaHash.h Nodo.h Arbol.h DicPaginas.h main.cpp
	g++ -c main.cpp
```

Informe del desarrollo

Práctica 001

Fue realizado de manera conjunta debido a su sencillez.

Envío 001: 151

Prácticas 002 y 003

Cada integrante del grupo realizó una parte del código y luego todo fue implementado en un mismo archivo.

Envío 002: 152

Envío 003: 527

Prácticas 004 y 200

Fueron realizados de la misma manera, pero se repartía el trabajo de distinta manera y siendo ayudado por el otro compañero, debido a la complejidad de la práctica y las dificultades que van surgiendo.

Envío 004: 919

Envío 200: 1521

Práctica 300

En la práctica final se encontró una mayor dificultad, a esto se sumó la falta de apoyo de mi compañero, que abandonó todo trabajo y no mostró ninguna ayuda. La primera dificultad estuvo en la reflexión de que decisiones tomar, y después en la reflexión del código, no en la implementación. Después de comenzar la implementación, se abandonó la idea del Árbol AVL como se ha explicado anteriormente, y se diseñó otro árbol. Esto acarreó una pérdida importante de tiempo, afectando a la confección del resto de la práctica. Finalmente se encontró una solución que funcionaba correctamente, pero no hacía uso de los requisitos pedidos, por lo que fue suspendida.

Envío 300: 1581

Revisión práctica 300 (Enero)

Después de la debida entrevista se trataron los fallos que contenía la anterior entrega y se realizó una actualización del proyecto cambiando la estructura de la tabla de dispersión. Esta actualización utiliza debidamente la memoria dinámica. En la realización de esta práctica se cometió el grave error de no dejar a la tabla de dispersión redimensionarse ni siquiera una vez, por lo que los fallos que contenía no salieron a la luz hasta la entrevista.

Envío revisión 300: 1990

Reentrega Práctica 300 (Junio)

Esta es la última convocatoria a la que se presenta esta práctica. Se ha cambiado todo lo que andaba mal en las dos entregas anteriores, tanto lo referente a la memoria dinámica, como en retoques en el Árbol Trie, Makefile, etc. Antes de la corrección de la práctica 300, se profundizó teóricamente sobre la memoria dinámica, para solventar la falta de conocimientos en este tema que habían llevado a dos entregas erróneas. Además, se asistió a una tutoría para resolver el resto de dudas que quedaban pendientes.

Envío reentrega: 2254

Conclusiones y valoraciones personales

La realización de las primeras prácticas fue algo sencillo y nos ayudó a adaptarnos al lenguaje. Al llegar a la práctica 004 comenzaron algunos contratiempos, pero fueron rápidamente resueltos.

La práctica 200 supuso un punto de inflexión, un mal planteamiento de inicio nos llevó a un código erróneo y con múltiples fallos que fue resuelto con un nuevo punto de vista y más cuidado en la realización del código. Finalmente, aunque un poco fuera de plazo, conseguimos resolverla.

La última práctica supuso un calvario. Inicialmente se comenzó diseñando un AVL, pero su implementación suponía un sinfín de errores, y una vez mejor planteado el problema se decidió cambiar el diseño del árbol a un Árbol Trie, por ofrecer una implementación mas sencilla y una mayor eficiencia debido a la gran cantidad de palabras que se van a introducir. Esto es una gran ventaja ya que se introducirán muchos prefijos compartidos.

La práctica ha supuesto un reto en muchos apartados y ha resultado difícil entenderse entre los integrantes del grupo en ocasiones, pero finalmente se acabó resolviendo bien el proyecto. Bueno eso cuando el compañero estaba dispuesto a trabajar, que a partir de la última práctica su ayuda fue nula.

La entrega de diciembre proponía una solución completamente diferente a la que busca la asignatura, por lo que no era válida. En enero se propuso otra alternativa, pero tenía múltiples fallos que no se habían visto antes, por lo que tampoco se aprobó. En esta última entrega, la tabla de dispersión comienza con tamaño 1, lo cual es muy ineficiente, pero asegura que funciona correctamente.

Nunca una práctica había supuesto un reto tan grande, las numerosas dificultades encontradas vinieron por culpa de una falta de base teórica, que impedía que se pudiera realizar nada correctamente. Finalmente, cuando se entendieron los conceptos todo fue más fácil.

Además, la ausencia de compañero en el peor tramo de la práctica fue un duro golpe, una falta de ayuda y tiempo que llevó al fracaso. Y esta es la tónica de cualquier trabajo de cualquier carrera de la Universidad. Algo que no ayuda en nada a los alumnos trabajadores y si a los que quieren aprobar todo con la ley del mínimo esfuerzo.