MEMORIA PROYECTO BUSCADOR

AED1 Nov, 2024

Ángel Ruiz Fernández
Carla Ramos García
G2.2 B117

MEMORIA

1. Ánalisis del programa

1.1. Clases

- class Pagina
  Representa una página, almacena su url, titulo, relevancia y contenido.

- class PagListIt : public std::list<Pagina>::iterator
  Usa Pagina
  Iterador heredado que implementa el operador '<' para poder ser ordenado en un contenedor ordenado.
  Representa una referencia a elemento de std::list<Pagina>

- struct nodo_trie_t
  Usa PagListIt
  Representa un nodo del árbol trie de palabras. De este cuelgan hijos en un diccionario <char, nodo_trie_t>. Relaciona con una lista de referencias a página (PagListIt).

- class Árbol
  Usa PagListIt y nodo_trie_t
  Contiene la estructura árbol oculta, y posibles operaciones sobre el.

- class Diccionario
  Usa Pagina, PagListIt y Árbol
  Contiene la estructura de tabla de dispersión (std::list<Pagina>[N]), su función de hash, y una instancia de Árbol.
  Expone las posibles operaciones sobre la tabla, además de pasar las operaciones del árbol.

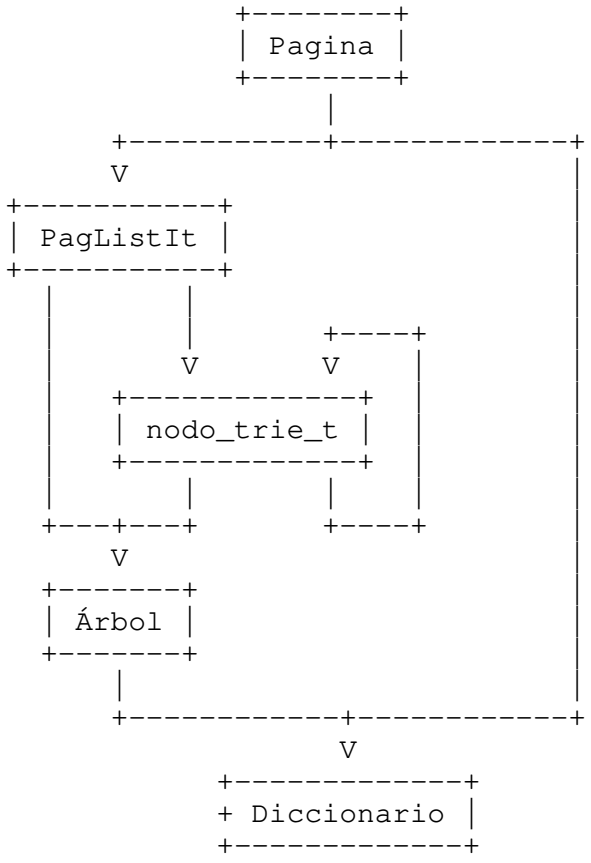

Fig. 1 Diagrama de clases

1.2. Módulos

Cada módulo (menos main) tiene un header asociado

- diccionario.hpp
  Contiene la declaración de todas las clases y structs, interfaz para ser usada por el interprete.

  diccionario.cpp
  Contiene la definición de todos los métodos de las clases, que especifican la estructura de datos y las operaciones asociadas de la base de datos.

- interprete.hpp
  Contiene la declaración de las funciones que se encargan de interpretar los comandos de la entrada.

  interprete.cpp
  Contiene la definición de las funciones de interpretación de comandos, que llaman a operaciones sobre el diccionario, que se le es pasado por referencia.

- main.cpp
  Contiene el bucle principal del programa, que lee comandos y llama al interprete. Es propietario de la instanciación del diccionario, donde se almacenan todos los datos de la aplicación.

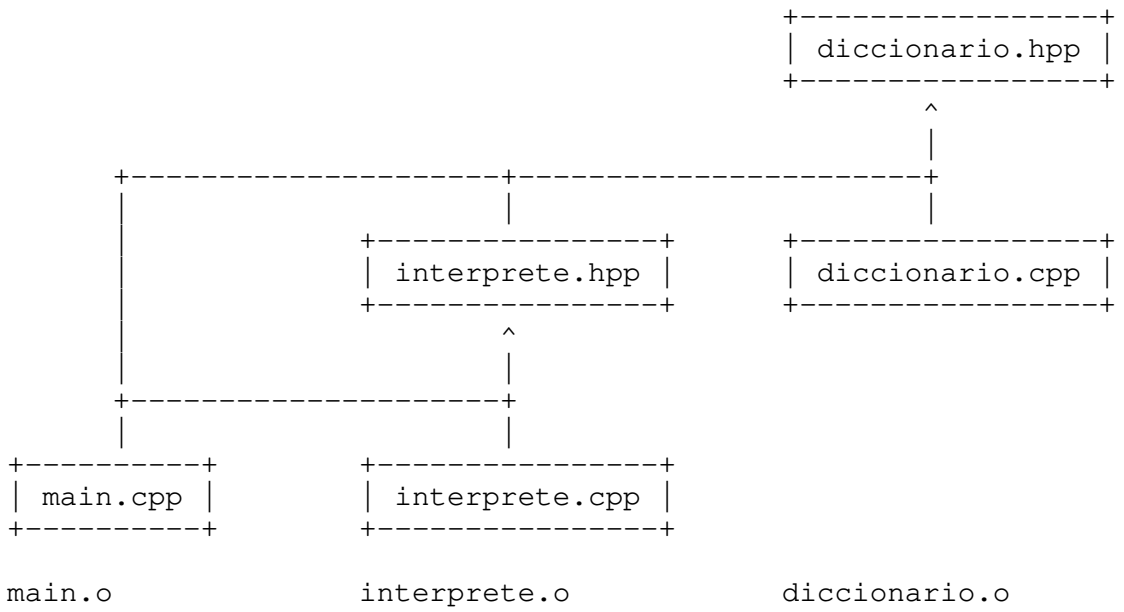

Fig 2. Diagrama de módulos y unidades de compilación

1.3. Makefile

En la Makefile, primero defino variables tales como el nombre del proyecto, el nombre de la salida, el compilador, y los parámetros de compilador y linker; además de automáticamente guardar en SRC los archivos .cpp, de los cuales se saca el nombre de los archivos de objeto con un patsubst.

La regla all depende del binario de salida, que se crea en una regla de link que depende de los archivos de objeto. Los archivos de objeto se compilan en una regla patrón: para cada ".o" que depende de un ".cpp" y/o ".hpp", con el mismo nombre base.

Adicionalmente tengo reglas PHONY para realizar el test con la entrada de prueba, limpiar, crear el tarball, y subir a Mooshak con un script en python automatizado (no incluido en el listado adjunto).

La makefile contiene todas las dependencias existentes.

1.4. Normalización

Parece que muchas personas no son conscientes de que el lenguaje C++ soporta UTF-8 nativamente. Usando las variantes 'wide' o 'multibyte' de tipos y operaciones de la STL que soportan operar con estos caracteres Unicode como std::wcin/wcout, std::wstring (std::basic_string<wchar_t>, y std::towlower(). Usando estas características, la normalización sería trivial, el problema es que la especificación del programa requiere unas conversiones muy especificas que difieren del comportamiento de std::towlower().

wchar_t std::towlower(wchar_t) convierte todos los caracteres con variante mayúscula a su variante minúscula, de todos los idiomas. La especificación indica que debemos normalizar solo los caracteres del Español, ignorando el resto, por tanto primero se usa std::tolower() que trabaja solo sobre ASCII (ignorando Unicode), y entonces después, se manejan los casos para los caracteres específicos del español, como las tildes y la 'ñ'.

1.5. Tabla de dispersión

1.5.1. Tipo

El tipo elegido para la tabla de dispersión es abierta, ya que es muy sencilla de implementar, y realmente porque las tablas cerradas no ofrecen ventajas significativas: con sets grandes, la optimización de memoria es insignificante, y para tablas del mismo tamaño, la cerrada es casi siempre más lenta.

1.5.2. Función de dispersión

Al principio mientras probaba, sumaba todos los caracteres en un entero y aplicaba el módulo, esto conlleva una dispersión bastante mala ya que las cadenas son de longitudes parecidas.

Así que se reemplazó por una variante de un hash iterativo, donde el valor inicial t es un primo (5381), y por cada carácter de la cadena a hashear, se le suma a t desplazado 5 bits a la izquierda (para aumentar la dispersión), con t, con el carácter. Finalmente se retorna t módulo tamaño de la tabla. Así la dispersión es mucho más uniforme.

1.5.3. Reestructuración

No se realiza reestructuración, se considera que el tamaño de la tabla es suficiente para el número de elementos.

1.5.4. Liberación

No hay liberación explicita. Gracias a que la tabla es una propiedad array de un contenedor de la STL en la clase Diccionario, esta se elimina profundamente (llamando automáticamente al destructor de cada contenedor) al destruirse la instancia, que al estar siendo instanciada en la función main en el stack, se destruye automáticamente al llegar al final de main(), antes de salir del programa, gracias a RAII de C++.

1.6. Árbol

1.6.1. Tipo

Se ha implementado un árbol Trie, ya que es muy simple de implementar, y lo bastante rápido para la aplicación. Buscar como máximo 26 elementos por carácter de palabra, usualmente menor que 15, es computacionalmente poco costoso. Implementar AVL no ofrecería una ventaja clara en velocidad, costaría mas de implementar y el balanceo es computación extra.

1.6.2. Definición de árbol y nodo

El nodo es un struct, que contiene un std::set ordenado de referencias a páginas, y un diccionario de <wchar_t, nodo> hijos, para hacer el árbol.

La clase Árbol esconde en privado un diccionario como <wchar_t, nodo> como raiz de donde cuelgan todos los hijos, y expone solo 2 operaciones, insertar y buscar.

1.6.3. Referencia a páginas

Las referencias a las páginas en los nodos del árbol son iteradores estándar de la STL, que usan punteros internamente.

1.6.4. Liberación

Al igual que en la tabla de dispersión, los diccionarios y conjuntos de los nodos se liberan automáticamente al destruirse la raíz, al destruirse la instancia de Árbol, al destruirse la instancia del Diccionario al final de main().

1.7. Globales

No se usa ningún tipo de variable ni constante global.

1.8. ChatGPT

En ningún momento se ha usado ChatGPT para ninguna parte del proyecto, solo se han usado herramientas de depuración serias como gdb y valgrind, y recursos deterministas convencionales como investigar documentación y posts en foros de desarrollo tales como StackoOverflow, escritos por el conocimiento, experiencia y sabiduría de personas humanas usadas a C++, gcc y sus intricaciones.

2. Listado completo del código

Véase documento adjunto.

3. Informe de desarrollo

El problema 001 fue muy trivial usando std::cin en un bucle.

Para el 002, como descrito en 1.4., me acordé de que C++ soporta Unicode nativamente, y usamos casos de caracteres Unicode sobre wchar_t en vez de sobrecomplicarnos intentando decodificar caracteres multibyte manualmente, una practica poco recomendable.

En el siguiente, 003, se empezó a modularizar, creando el módulo interprete. En el main() de la aplicación entonces, se abrió un bucle donde se lee un token mientras haya entrada disponible, y se llama al interprete para procesarlo.
El interprete, según el primer token en un switch, lee diferentes datos correspondiéndose a los diferentes comandos, para los que hay que dar salidas de placeholder.

En el 004, se abre el módulo de diccionario, que se implementa con una única gran lista ordenada en una nueva clase, Diccionario, que expone las apropiadas operaciones que se llaman desde el interprete.
En nuestro caso, decidimos usar un contenedor diccionario ordenado de la STL, std::map<std::wstring, Pagina> para simplificar las operaciones, .find() para buscar a modo de lista, y .insert_or_assign() para insertar o modificar cuando coincida la url.
El contenedor se ordena mediante un concepto un tanto extraño de C++, un objeto Compare de un contenedor ordenado. El struct comparador_páginas_url, que implementa un operator(), que recibe dos std::wstring y las compara usando el metodo indicado en la especificación (.compare() < 0). De esta forma, al insertar en el diccionario, este se encarga de ordenar los elementos automáticamente. Otro concepto de C++ que se usa es std::optional<>, para cuando no se encuentra una página por url, no devolver nada.
Finalmente se implementa el comando ’u’ en el interprete usando las operaciones del diccionario, siendo pasado por referencia desde main(), el propietario del diccionario.

Para el 200, se nos pide implementar el diccionario con una tabla de dispersión. Elegimos usar abierta por practicalidad, descrito en 1.5.1. Primero definimos la tabla como un array de cubetas de tipo std::vector (esto costó varias horas de depuración en el siguiente problema) tal que std::vector<Pagina> tabla[N] siendo N el tamaño de la tabla.
Después se implementó la función de hash de suma secuencial (también costó optimización que tuvimos que hacer después al mooshak reportar "Time limit exceeded"), y las operaciones de insertar y consultar en la tabla usando esa función de hash. Corregimos el excesivo tiempo escogiendo una función de hash de mas calidad, cuyas características son descritas en 1.5.1., y para optimizar el máximo posible, se revisaron ciertas cosas indicadas por la herramienta callgrind de valgrind, muy util.
Solo se tuvo que modificar el módulo diccionario para este problema.

En el 300, se definió el nodo del árbol, y la clase Árbol, de tipo trie (descrito en 1.6.1.). el struct nodo_trie_t tiene 2 miembros, un vector de referencias (iteradores, ya que son algo menos peligrosos que punteros de C, que no es buena practica usarlos en C++) a páginas tal que std::vector<std::list<Pagina>::iterator>; nótese std::list<Pagina>, pues tuvimos que cambiar las cubetas de la tabla de dispersión a std::list en vez de std::vector (el contenedor de referencias cambia en el problema siguiente).

El programa sufría de fallos de segmento (página, en realidad), que ocurrían accediendo a las páginas mediante la referencia del árbol. gdb reportaba que las referencias al vector se volvían invalidas, tras un tiempo después de insertarlas. Tardamos un tiempo embarazoso en darnos cuenta de que, cuando se inserta a un std::vector, todos los punteros y iteradores a sus elementos pueden ser invalidados, al sufrir un realloc. De manera que decidimos usar un std::list, cuyas referencias no se invalidan al insertar.

El otro miembro del nodo es un diccionario de otros nodos hijos formando el árbol, con clave wchar_t, de forma std::map<wchar_t, nodo_trie_t>.

La nueva clase Árbol, en privado tiene la raiz del árbol, que es directamente un std::map<wchar_t, nodo_trie_t>, y expone las dos operaciones (descrito en 1.6.2.) usadas por la clase Diccionario propietaria de su instancia, que wrapea las operaciones en su propia interfaz. Al insertar al árbol, las referencias se ordenan mediante std::sort y una función de comparación que toma iteradores a std::list<Pagina>.

En el interprete, se implementa el comando ’b’ usando estas nuevas operaciones.

Decidimos realizar los problemas opcionales 301 y 302 por completitud.

Para realizar el 301 decidimos cambiar el contenedor de las referencias en el nodo del árbol a std::set, ya que así podíamos aprovechar las funciones std::set_intersection (y mas tarde std::set_union (aunque este es muy facil de implementar insertando)) de la STL en la implementación del comando ’a’, a partir del cual es trivial, iterando sobre los diferentes conjuntos de referencias, e irlos intersecando, de manera que queden solo los que están en todos, implementando así AND.

std::set al ser ordenado, ya no es necesario el uso de std::sort, pero si es necesario por ejemplo que el tipo del contenedor (iterador) tenga operator<() definido con el fin comparar y ordenar. Para ello creamos otra clase, PagListIt, heredada de std::list<Pagina>::iterator, que implementa un constructor default del padre, necesario, y el operador, y se cambia el contenedor de referencias de páginas en el nodo a std::set<PagListIt>.

Para el 302 fue trivial entonces usar std::set_union, uniendo los conjuntos de referencias en el comando ’o’, mismo método que en el 301.

En el problema 303 se tuvo que implementar una nueva operación en el Árbol para buscar todas las palabras con un prefijo. Este método, llama a una función recursiva para recorrer todas las palabras que cuelgan de la secuencia de prefijo y insertarlas a un vector de tuplas de las palabras y el número de referencias que contienen, tal que std::vector<std::pair<std::wstring, int>>. En el caller, ese vector resultante finalmente se ordena mediante std::sort y otra función de ordenación, y se devuelve.

Proyecto: Buscador Fecha de inicio: 14/10
Programadores: Ángel, Carla Fecha de finalización: 19/11

DÍA*	PROBLEMA	ANÁLISIS	DISEÑO	IMPLEMENTACIÓN	VALIDACIÓN
14/10	001	3	1	3	2
14/10	002	5	15	12	5
14/10	003	5	3	10	7
15/10	004	10	15	50	40
25/10	200	7	25	80	30
12/11	300	15	45	120	60
19/11	301	6	20	40	5
19/11	302	2	5	7	5
19/11	303	10	10	25	7
TOTAL 705m		58	139	347	161

Tabla 1. Dedicación temporal. Los tiempos están en minutos y están basados en datos estimados. *DÍA de finalización de implementación del problema.

Ambos integrantes participaron en el desarrollo del proyecto, aunque Ángel tuvo un rol más destacado en ciertos aspectos debido a su mayor experiencia y soltura en programación. Esto no solo contribuyó al éxito del proyecto, sino que también permitió que Carla Ramos aprendiera y mejorara sus habilidades en el proceso.

4. Conclusiones y valoraciones personales

Al principio los problemas resultaron fáciles, pero a partir de la tabla de dispersión y el árbol, hubo que emplear bastante tiempo depurando y optimizando. Pensarías que no es posible tener segfaults usando solo la STL pero si. Si lo miras con valgrind, la STL comete memory leaks insalvables por su diseño y uso de templates. Si querían que usáramos punteros para los árboles y listas, el lenguaje para eso es C, no C++.

Sin embargo, para usar C++, no se explican bien toda la funcionalidad que C++ ofrece. Parece que la mayoría no conoce si quiera que C++ soporta UTF-8; inmensamente útil para la normalización. No se explican los contenedores de la STL, cuando usarlos, sus propiedades y operaciones, tales como std::vector, std::list, std::deque, std::set, std::map...

Y tampoco se enseña el uso de depuradores como gdb y valgrind (callgrind), herramientas inmensamente útiles y indispensables para desarrollo de cualquier software mas grande que una prueba.

Finalmente, este proyecto, cuyo objetivo era el uso practico de tablas de dispersión y árboles, no es representativo del funcionamiento de un motor de búsqueda real.