# Final 2020

materia

# Estructuras de Datos y Algoritmos I

dicatada por

# Federico Severino Guimpel

en la carrera

# Licenciatura en Ciencias de la Computación

en

# Facultad de Ciencias Exactas, Ingeniería y Agrimensura

de la

# Universidad Nacional de Rosario

Facundo Emmanuel Messulam*

July 27, 2020

[0]Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Licenciatura en Ciencias de la Computación

# Eleccion de estructuras de datos utilizadas

## Parser

El parser es una funcion que devuelve `Metadatos`, esta estructura fue elegida por su simpleza. Implementar un sistema complejo para una serie tan simple de operaciones (que no tienen siquiera recursion en su gramatica), es demaciado complejo, sin mucha ganancia.

La idea usada es la de leer cada entrada del usuario dos veces, una para crear informacion sobre la entrada, con un automata de estado finito muy simple, que avanza de izquierda a derecha, y chequea errores de gramatica, y devuelve una estructura con toda la información para la siguiente pasada. La siguiente pasada, lee la estructura, y la usa para gestionar memoria y leer correctamente la entrada, precaracterizada como "error", "asignacion", "operacion" y sus correspondientes subdivisiones: los tipos de error (ver seccion errores más abajo); asignacion por extension y compresion; y las operaciones union, interseccion, resta y complemento, respectivamente.

## Mapeo alias-arbol

Para hacer el mapeo entre los aliases y los arboles, usé una estructura especialmente simple, y muy extendible: el trie.

En esta implementacion, el trie esta dividido en dos partes, el Mapa y el Trie. El Mapa es una tabla hash muy simple, que permite enlazar una letra a un nuevo Trie. El Trie es contiene informacion sobre si corresponde al ultimo caracter de una palabra, y en este caso tiene una referencia al ArbolIntervalos correspondiente, tambien tiene un Mapa para poder seguir avanzando.

El trie funciona agregando palabras letra por letra y en la ultima letra guardando el arbol. Cuando se quiere obtener el arbol correspondiente a un alias, el proceso es el mismo.

En el mapa se usa una funcion hash muy simple que mapea los caracteres a un entero unico, de tal forma que los enteros sean consecutivos y empiezen en 0.

En esta implementacion se decidio no hacer un eliminador de aliases. Y, cuando el arbol se elimina, los punteros a los arboles se eliminan tambien. Estos cambios son para simplificar el uso, y evitar operaciones innecesarias en la implementacion actual.

## Guardado de conjuntos

Esta es la parte más grande e importante del projecto. La estructura usada es un arbol Adelson-Velskii y Landis, sin embargo, tiene una gran cantidad de cambios para que sea util para este projecto.

La primera y más importante es que se usan intervalos en vez de elementos. Para esto se necesita un "maximo final de subarbol" en cada nodo, y luego los intervalos estan ordenados de acuerdo a el inicio del intervalo. Entonces, las operaciones del arbol orginal se pueden seguir haciendo, por lo que las caracteristicas de optimización se mantinen.

Valga aclarar que los intervalos en los arboles son siempre cerrados, esto simplifica el manejo, y disminuye el consumo de memoria.

Luego, para poder implementar correctamente las funciones de impresion y complemento, se necesito agregar una invariante: los intervalos en el arbol deben ser disjuntos. Esto no es dificil de implementar, una explicacion esta disponible en la seccion "Dificultades: Eleccion del arbol AVL".

El arbol AVL requiere para su rebalanceo que se tenga conocimiento del balanceo actual de cada subarbol. Es decir, se debe saber en cuanto varia la altura de cada subarbol del nodo para todos los nodos del arbol. A pesar de que solo se necesita la variacion, es mucho más simple de pensar e implementar si se guarda la altura de cada subarbol en todos los nodos. Esto fue lo que se hizo.

# Formatos aceptados para los alias

Opté por otorgar la mayor flexibilidad posible al crear aliases. La idea es que el usuario no tenga que crear aliases muy largos, ni reemplazar 'ñ' por 'ni', por lo que se permiten todos los caracteres latinos, si es que el usuario tiene instalado el locale "ES_es". Para mayor facilidad de uso tambien se permiten usar caracteres numericos (ver seccion sobre compilacion y uso).

# Mensajes de error devueltos por el interprete

## Errores generales

El interprete posee un mecanismo para devolver un mensaje de error con la letra en la que falló. Por ejemplo, si el usuario escribiera:

```
a = {d}
```

Veria la salida:

```
Error de parseo: {d}
```

Esto quiere decir que el usuario cometio un error en ese punto. En este caso, ingreso una letra donde deberia haber un numero.

## Errores de alias

Si el usuario ingresara un comando bien escrito, pero que opera sobre un alias que no existe, el interprete avisa acordemente.
Veamos un ejemplo, si a nunca fue definido, y se ingresa el siguiente comando:

```
imprimir a
```

La consola devuelve:

```
Alias no existe: a
```

Para comandos con más de un alias, por cada alias faltante, el interprete avisa en una linea nueva.

## Errores de impresion

Para el comando "imprimir" el interprete avisa si faltan operandos. Esto es decir, si el usuario escribe:

```
imprimir
```

La consola imprime:

```
Error de parseo: impri
Uso: imprimir <alias>!
```

Indicando el uso correcto, y la posicion del error.

# Compilación y uso

## Compilación

Este paso es muy simple, despues de obtener el codigo fuente, hacer:

```
make all
```

Si el usuario tiene `gcc` (GNU C Compiler) entonces el no deberia haber errores.

El usuario podra tambien querer usar las letras del alfabeto latino, como eñe y letras acentuadas. En muchos casos, el usuario tendrá ya este soporte. Si el interprete (ver "Uso" más abajo) imprime:

```
Error: no tiene el locale es_ES.utf8 instalado, tildes y otros no van a funcionar
```

se debera descargar el locale "ES_es.utf8", como hacer esto depende del sistema operativo del usuario, deberá referirse al manual.

### Uso

El uso del programa es muy simple, una vez terminado el paso anterior (make) el usuario podra invocar:

```
./Interprete
```

Luego, el usuario podrá ingresar comandos.

#### Comandos disponibles

Asignacion de conjuntos:

```
<alias> = {<aliasInterno>: k <= <aliasInterno> <= k1}
<alias> = {k, k1, k2, ...}
```

Union:

```
<alias> = <alias1> | <alias2>
```

Interseccion:

```
<alias> = <alias1> & <alias2>
```

Resta:

```
<alias> = <alias1> - <alias2>
```

Complemento:

```
<alias> = ~<alias1>
```

Impresion:

```
imprimir <alias>
```

Salida:

```
salir
```

## Dificultades

Nota: Antes de leer esta seccion, puede convenir leer "Eleccion de estructuras de datos", más arriba.

### Eleccion del Arbol AVL

Para guardar los conjunto elegí la estructura arbol AVL, el problema surgió cuando habia que imprimir.

El arbol puede tener varias veces el mismo "elemento", esto parece ser falso a primera vista, ¿como puede un arbol de busqueda binario tener un elemento más de una vez? La respuesta es que el arbol en este caso contiene conjuntos, estos no pueden estar repetidos, pero esto solo quiere decir que no exite un intervalo con el mismo inicio y mismo final en otra parte del arbol. Pero puede ocurrir que exista en el arbol un conjunto {2, 3} y uno {3, 3}. Si yo imprimiera ese arbol tendria el 3 repetido.

Para resolver este problema decidí agregar una invariante: el arbol debe contener conjuntos disjuntos. Pero es más facil dicho que hecho. Para la implementación, modifiqué el agregado de conjuntos para que nunca agrege un conjunto que no es disjunto con todo el arbol.

El algoritmo toma el conjunto A = [1, 3] y genera A' = [1-1, 3+1] extendiendo A en ambos sentidos. Despues chequea interseccion en el arbol, y, por cada B con el que insterseca, A es A union B y elimina B del arbol. El algoritmo se repite hasta que no haya intersecciones, recalculando A' cada iteración. Luego el conjunto A es agregado al arbol.

### Soporte para eñes

Un problema bastante grande fue implementar los caracteres para la eñe, tildes y dieresís. Si bien el programa puede ser usable sin estos agregados, la intención es que un usuario no familiar con las limitaciones de una entrada basada en `char` pueda usar este programa sin problemas.

Existen extensiones de los caracteres ASCII que agregan eñes y demas, pero se elijió la ruta mas extendible y universal, UTF8. Para hacer uso de esta caracteristica, se requirió usar `wchar_t`, y las versiones "wide" de las funciones de manejo de caracteres.

El manejo de caracteres con el sistema "wide" permite que los caracteres sean, literalmente, más anchos en su representación en bytes. El tamaño exacto esta determinado por el compilador, pero la idea es que puede almacenar correctamente una gama más grande de caracteres.

Una vez que la salida se establece como "wide", no se puede usar `printf`, se debe usar `wprintf` o equivalente.

Las constantes literales en este nuevo formato deben empezar con una 'L':

```
L"a imprimir"
```

### git

Como todo programador sabe, en particular, yo, errores ocurren. Encontrar donde occurrieron suele ser una tarea que ocupa una cantidad importante de tiempo y requiere tener una idea de donde ocurrio, y cuando, el error.

Para simplificar esta tarea uso git, la herramienta permite saber cuando fue agregado cierto codigo, y que otro codigo fue agregado al mismo tiempo. Tambien tiene la funcionalidad de buscar el lugar (en el tiempo) en el que ocurrio un error y que codigo fue modificado en ese momento (git bisect).

### Tests, ValGrind y GitHub Actions

Una cuestion cuando uno programa es saber si el codigo de hecho anda, para esto se puede usar `assert.h`, y la funcion `assert()`. Para este fin se crea una funcion "de testeo", en mi caso, esta en tests.c. Sin embargo, esta funcion se debe, de hecho, ejecutar. Para no tener que ejecutarla manualmente con cada cambio, utilizé el sistema de chequeo autmatizado que provee GitHub Actions.

El sistema en cuestion permite chequear automaticamente commit por commit, y avisa cuando hay un error. Tambien se ejecuto `test_main()` desde valgrind, así cerciorandome de que la funciones de hecho andan, sin perder memoria, u otros errores menos visibles.

## Otra información

Soy consciente de que existe un articulo [2] explicando algoritmos optimizados para realizar las operaciones de conjuntos en este trabajo. Sin embargo, no creo necesaria la implementación de algorimos más complejos dado que los casos extermos para las implementaciones actuales son muy dificiles de alcanzar. Con esto me refiero a que dado que la entrada esta limitada a tipeo a mano, es

muy difil que un usuario llege a tener que unir, por ejemplo, el conjunto de todos los pares, con el conjunto de todos los impares.

El algoritmo utilizado lidia muy bien con los conjuntos que un usuario puede ingresar: La herramienta funciona muy bien para lo que se la requiere.

## Bibliografía


[1] G.M. Adelson-Velskii and E.M. Landis. *An algorithm for the organization of information*. 1962.

[2] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. "Parallel Ordered Sets Using Join". In: *CoRR* abs/1602.02120 (2016). arXiv: 1602.02120. URL: http://arxiv.org/abs/1602.02120.

[3] Donald Knuth. *The Art of Computer Programming: Searching and Sorting Algorithms*.

[4] J. J. Tenembaum A. N. Augenstein. *Estructuras de Datos en C*. Prentice-Hall, 1991.