

|  |
| --- |
| Domain Name System |
| Complejidad Temporal, estructura de datos y algoritmos  Comisión: 2 |
| 29 agosto  Universidad Nacional Arturo Jauretche  Creado por: Alex Erik Wake  Profesor: Leonardo Javier Amet |



Índice

## [1-Cuerpo del trabajo](#_El_trabajo_está)

## [2-Módulo Administración- Ingreso y Almacenamiento](#_Modulo_Administración-_Ingreso)

## [3-Modulo Administración- Eliminación de nombre de equipos](#_Modulo_Administración-_Eliminación)

## [4-Módulo consultas](#_Módulo_consultas)

## [5-Módulo consultas – Primer consulta](#_Módulo_consultas_–)

## [6-Módulo consultas – Segunda consulta](#_Módulo_Consultas_–_1)

## [7-Módulo consultas – Tercer consulta](#_Módulo_consultas_–_2)

## [8-Métodos auxiliares](#_Métodos_auxiliares)

## [9-Métodos auxiliares - \_existe()](#_Métodos_auxiliares_-_1)

## [10-Métodos auxiliares - \_buscar()](#_Métodos_auxiliares_-)

## [11-Métodos auxiliares – igual()](#_Métodos_auxiliares_–)

## [12-Conclusiones](#_Conclusión_final)

## [13-Conclusión final](#_Conclusión_final)

## [14-Mejoras](#_Mejoras)

## [15-Condiciones](#_Condiciones)

Resumen

###### En este informe se verá el paso a pasó que realicé para realizar un mini programa de DNS en C# que trata de asociar información del recurso con un nombre de dominio y simular la conexión de red en un árbol general.

###### El trabajo está dividido en 3 partes principales las cuales son: Módulo de administración, Módulo de consulta y Métodos auxiliares.

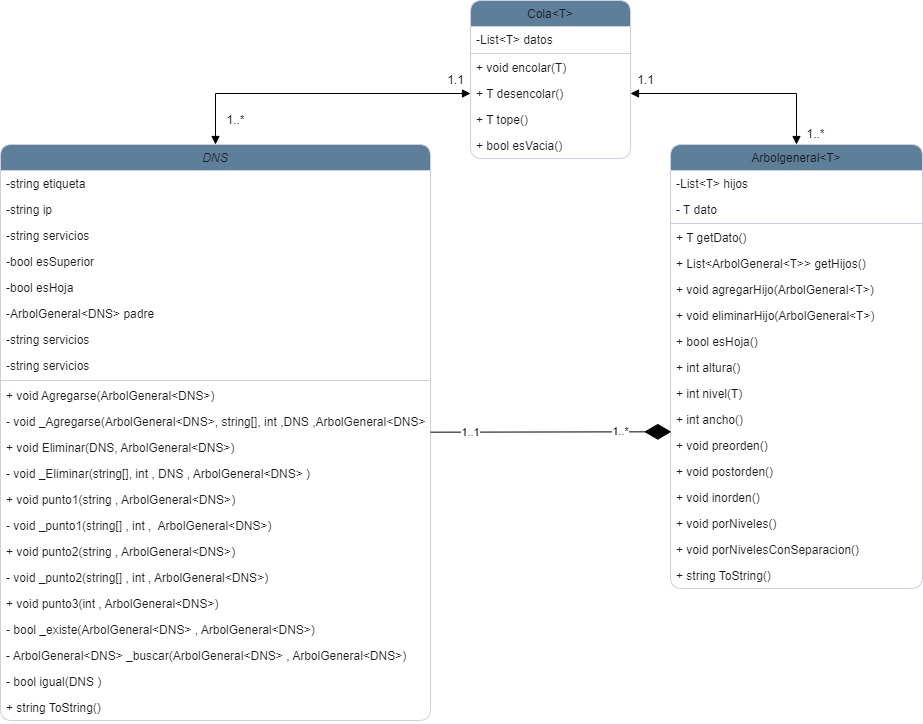
###### En cada uno de estos módulos explico y demuestro el funcionamiento del módulo, del código y cuento mi experiencia personal durante el desarrollo del proyecto y los problemas que fueron surgiendo a lo largo del mismo.

Introducción

###### Domain Name System o DNS(sistema de nombres de dominio) es un programa de nomenclatura jerárquica para recursos conectados en una red. La función más importante de un protocolo DNS es asociar una dirección de IP con un nombre de dominio (URL) más amigable para el ser humano para que sea más fácil de recordar y usar, es decir, un DNS traduce un nombre de dominio (por ejemplo: [un.dominio.cualquiera.com](http://www.google.com)) con la dirección IP del equipo servidor de la URL (por ejemplo : 255.255.1.1), así de esta forma, nosotros podemos comunicarnos con una página sin saber su dirección de IP, solamente sabemos su nombre de dominio y el DNS hace el resto del trabajo ya que en la red de internet nuestros dispositivos (por ejemplo: celulares, computadoras, etc.) se conectan e intercambian datos con aplicaciones o servidores a través de la dirección IP.

Cuerpo del trabajo

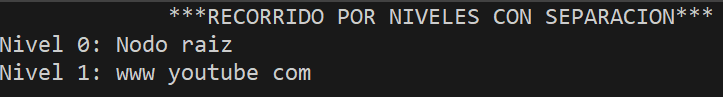
###### El trabajo está principalmente compuesto por una clase DNS que yo mismo desarrollé y 2 clases que fueron brindadas por la cátedra: ArbolGeneral<T> y Cola<T>, estas dos clases propuestas no fueron modificadas en casi nada, solamente agregué un método para facilitar la implementación y la depuración en la clase ArbolGeneral<T> pero que no es necesario y el sistema funcionaría incluso sin ella.

Diagrama UML final:

## Modulo Administración- Ingreso y Almacenamiento

###### El primer paso que tomé, luego de leer las pautas y consignas fue intentar crear el agregado de nodos DNS en un ArbolGeneral<T>, en sí no es dificil agregar información a un árbol, lo que si tiene un grado de dificultad mayor es estructurar la información para que esta misma se agregara de forma correcta.

###### El primer problema que tuve fue cuando al agregar información al árbol no se agregaba de forma correcta, se agregaba toda junta en vez de estructurarse y dividir la URL en secciones de dominio-superior, dominio y nombre de equipo, aunque si se dividían en 3 nodos distintos.



###### Logré solucionar este problema usando recursión y agregando los nodos DNS, separando la etiqueta y creando un ArbolGenaral<DNS> en cada recursión.

###### 

###### 

###### De igual manera el trabajo estaba por la mitad porque la idea principal es que si un nodo se repita, no se agregue al árbol y que el próximo nodo sea hijo de éste mismo que se está repitiendo. Como a este problema no lo pude solucionar, decidí seguir con las demás consignas.

###### Esta es la primer versión del código de agregado con el que estuve haciendo las pruebas que, si bien no cumplía con la consigna, si era útil más o menos útil para poder trabajar con el resto del trabajo.



###### Luego a lo largo del trabajo fui haciendo leves modificaciones para tener distintas nociones de como funcionaba, por ejemplo en vez de tener el nuevoarbol.agregarHijo(arbolaux) dentro del condicional ¡existe, lo moví fuera del condicional, es decir, coloqué esa línea de código en el cuerpo mismo del método para agregar todas las etiquetas de todas las URL con las que hice las pruebas y verificar funcionamiento de otros módulos.

###### Aunque no pudiera resolver la consigna, siempre tuve claro cuál es la parte la cual tenía que modificar y era el condicional existe en caso que sea true, el problema era que no sabía cómo modificar ese bloque de código para que en caso de que la etiqueta ya existiera se usara de padre para la que sigue sin agregarse nuevamente hasta que haciendo varias pruebas con la recursión y el depurador llegué hasta esta versión del algoritmo.

###### 

###### De igual manera este algoritmo falla, depurando encontré el problema, el problema está en que el método \_buscar() nunca puede ser 100% preciso ya que un mismo dominio puede estar almacenado en distintos niveles del árbol debido a su etiqueta padre y a la cantidad de etiquetas que posea el ArbolGeneral<DNS> completo a la hora de agregarse, el método \_buscar() retorna el primero que encuentra y el agregado sigue a partir de ahí sin preguntar.

## Modulo Administración- Eliminación de nombre de equipos

###### La eliminación de nombres de equipo no fue lo siguiente que desarrollé luego del ingreso y almacenamiento pero, de igual manera, el algoritmo de eliminación es un tanto muy similar al algoritmo del agregado.



## Módulo consultas

###### Las consultas en principio son similares entre sí, por lo tanto, la implementación también es similar, la dificultad principal que tuve en este módulo fue que al no tener bien implementado el Ingreso y Almacenamiento durante el desarrollo, las consultas nunca iban a ser un 100% correctas, entonces por más que intentara de todo, nunca tuve la certeza.

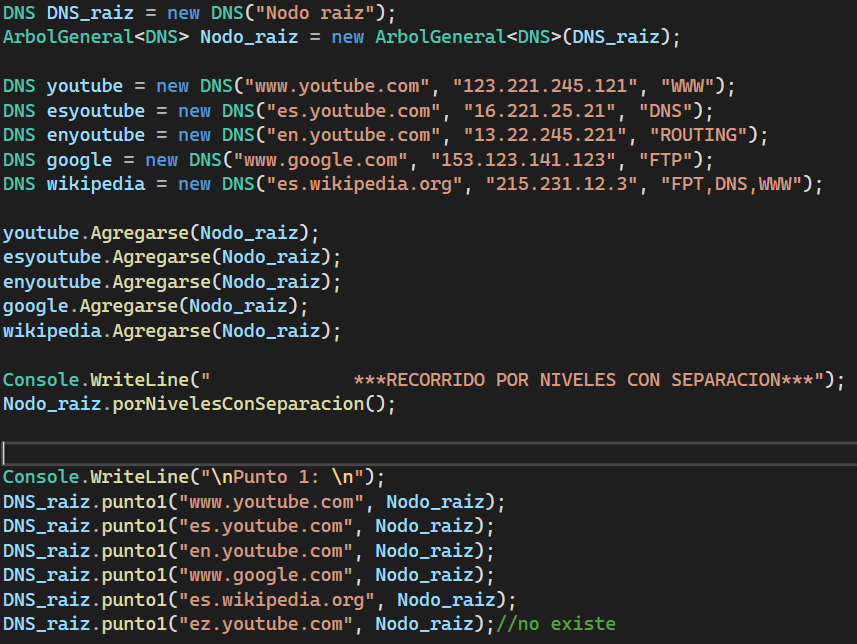
## Módulo consultas – Primer consulta

###### La primer consulta consiste en que, según una URL, imprimir su dirección IP y los servicios que provee. De todas las consultas esta me resultó la más difícil no tanto por el código en sí, si no por la estructura errónea que tenía el ArbolGeneral<DNS> debido al mal funcionamiento del método de ingreso y almacenamiento.

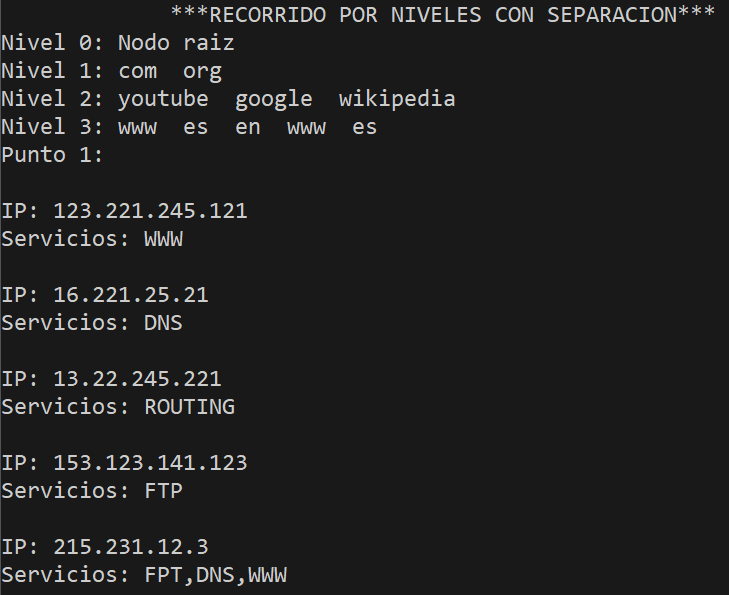
###### Para realizar esta consulta desarrollé el siguiente código:

###### 

###### El algoritmo consiste en recorrer la lista de hijos de un nodo raíz y comparar la etiqueta pasada por parámetro con la etiqueta ubicada en el dato del árbol, si existe una etiqueta igual, el algoritmo sigue buscando la próxima etiqueta hasta encontrarse con la última, en caso que corresponda la etiqueta y que sea la última, imprimir por pantalla la dirección IP y los servicios que ésta provee. Ejemplo: Dada esta estructura



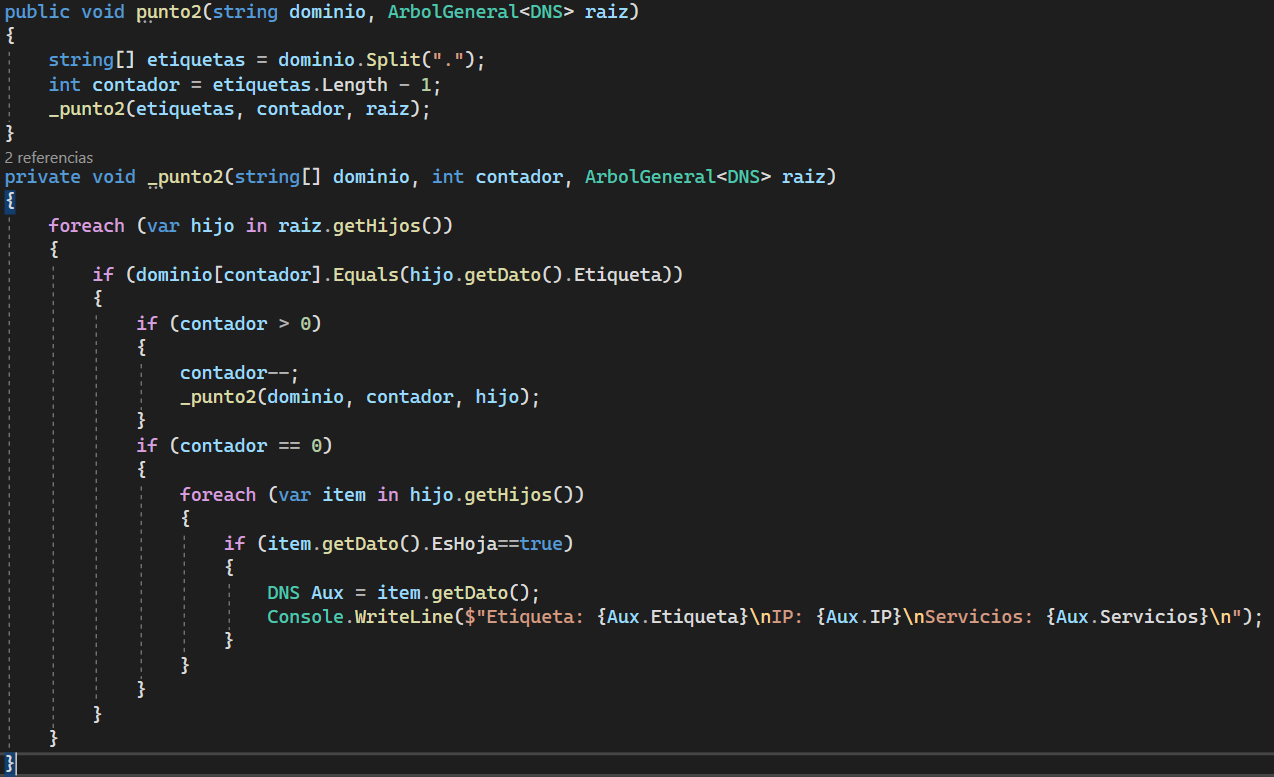
###### Devuelve estos resultados:



## Módulo Consultas – Segunda consulta

###### La segunda consulta consiste en dado un dominio correspondiente a un subdominio imprimir los equipos que tenga. La implementación en código que desarrollé para realizar esta consulta es muy similar a la anterior, aunque si tuve varios problemas con las condiciones y que resultados mostrar por pantalla.

###### Para realizar esta consulta desarrollé el siguiente código:



###### El algoritmo es similar al anterior, pero con la diferencia en la condición para imprimir y la información que imprime. Acá tuve mi primer problema porque no tenía clara que condición poner para imprimir sus nodos hijos, la otra dificultad que tuve después fue que desarrollé un recorrido postorden dentro de la clase DNS para invocarlo en la consulta, pero cuando invocaba el método con el recorrido postorden había un problema y el dato se pasaba sin hijos, entonces no imprimía nada, así que decidí implementar un recorrido en la misma consulta.

###### Ejemplo: dada esta estructura

###### 

###### Devuelve los siguientes resultados

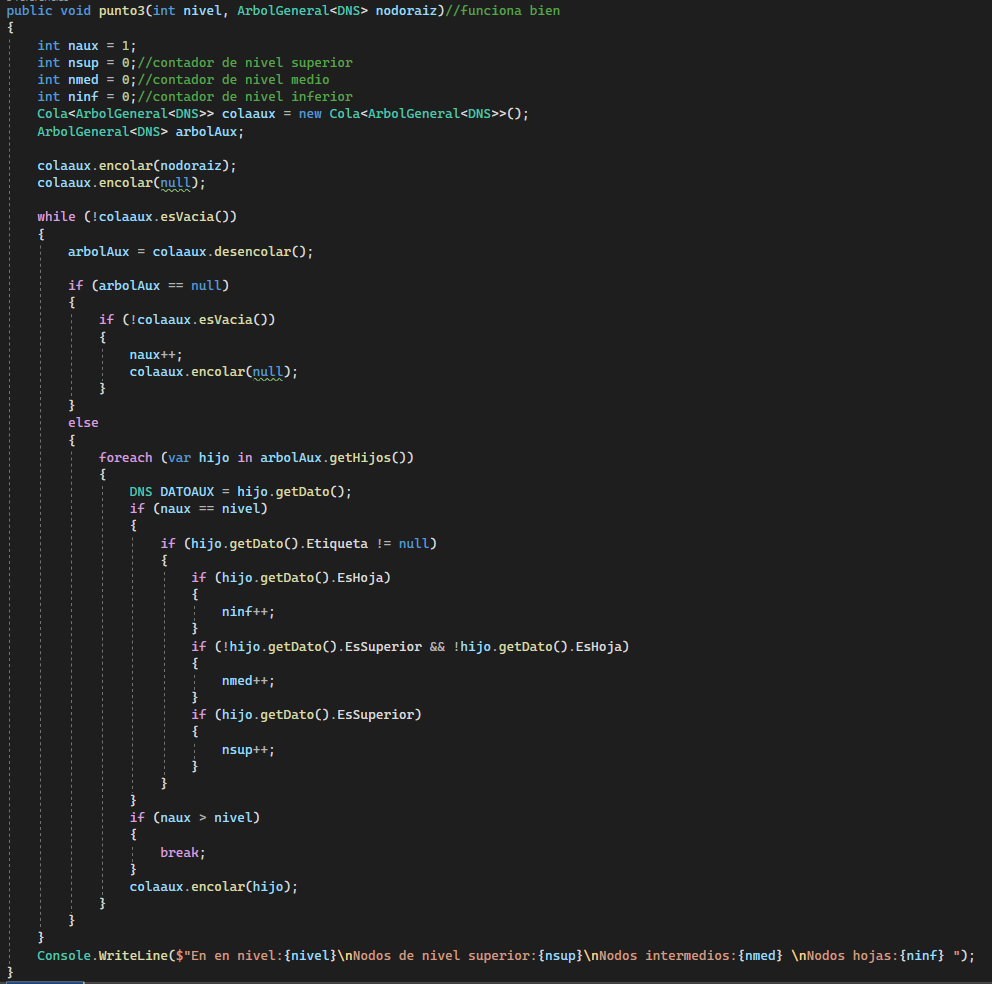
## 

## Módulo consultas – Tercer consulta

###### La tercer consulta consiste en que, dado un nivel de profundidad, debía imprimir la cantidad de nodos si corresponde a un nodo de nivel superior, nodo dominio, o nodo equipo.

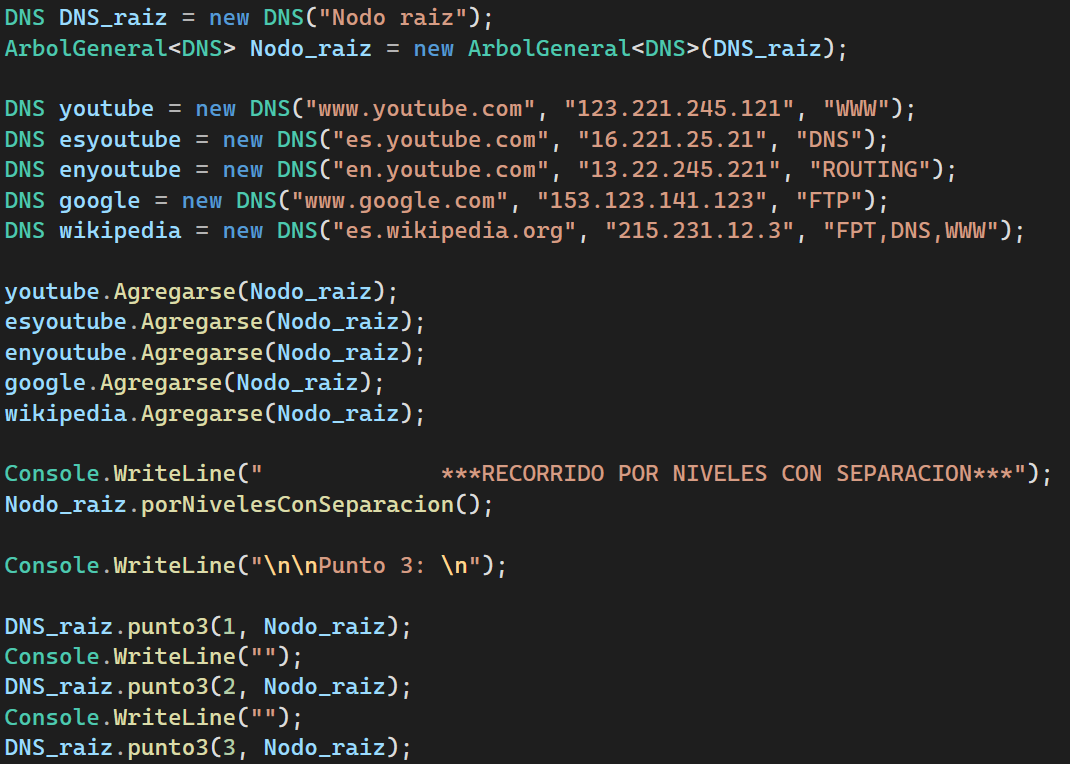
###### Esta consulta no me resultó difícil ya que simplemente tenía que modificar el método nivel() de la clase ArbolGeneral<T>.

###### Para realizarla desarrollé el siguiente código:

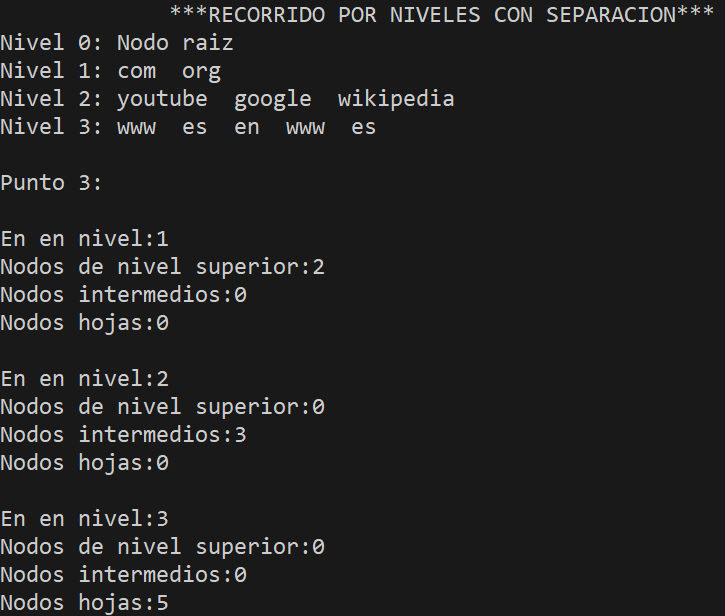


###### Lo que hace este algoritmo es realizar un recorrido por niveles con separación en un ArbolGeneral<DNS> hasta llegar al nivel que es pasado por parámetro, una vez llega al nivel, analiza todos los nodos que hay en el nivel y cuenta los distintos tipos de nodos que hay, si son nodos de dominio nivel superior, sub dominio o de equipo después imprime por pantalla el contador de cada uno.

###### Ejemplo: dada esta estructura



###### Devuelve los siguientes resultados:



## Métodos auxiliares

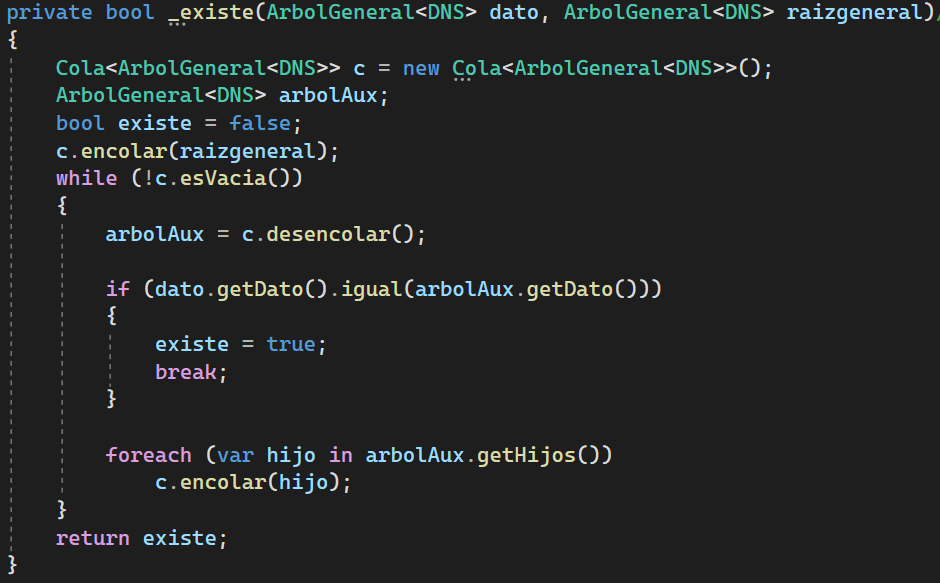
###### Durante el proceso de desarrollo implementé distintos métodos porque era constante la necesidad de comparar el dato almacenado en un ArbolGeneral<DNS> y saber si existen en el ArbolGeneral<DNS>, es decir, en la raíz principal, además, estos métodos, me facilitaron la implementación y tener más control del a la hora de realizar el agregado, la eliminación. También ayudan a tener un código más legible y más fácil de depurar. Todos estos métodos son privados así que no pueden ser utilizados fuera de la clase. Los métodos los siguientes: \_existe(), \_buscar(), igual().

## Métodos auxiliares - \_existe()

###### Cree este método para saber si existe un nodo (ArbolGeneral<DNS>) dentro del nodo raíz, raíz principal (ArbolGeneral<DNS>).

###### La finalidad de este método es condicionar el método Agregarse() y el método Eliminar()

###### El código es el siguiente:



###### Lo que hace este método es realizar un recorrido por niveles y comparar cada nodo de la raiz “raizgeneral” con el nodo “dato”, ambos son de tipo ArbolGeneral<DNS> y se piden como parámetro, y en caso de encontrarlo retorna true y en caso de no encontrarlo retorna false.

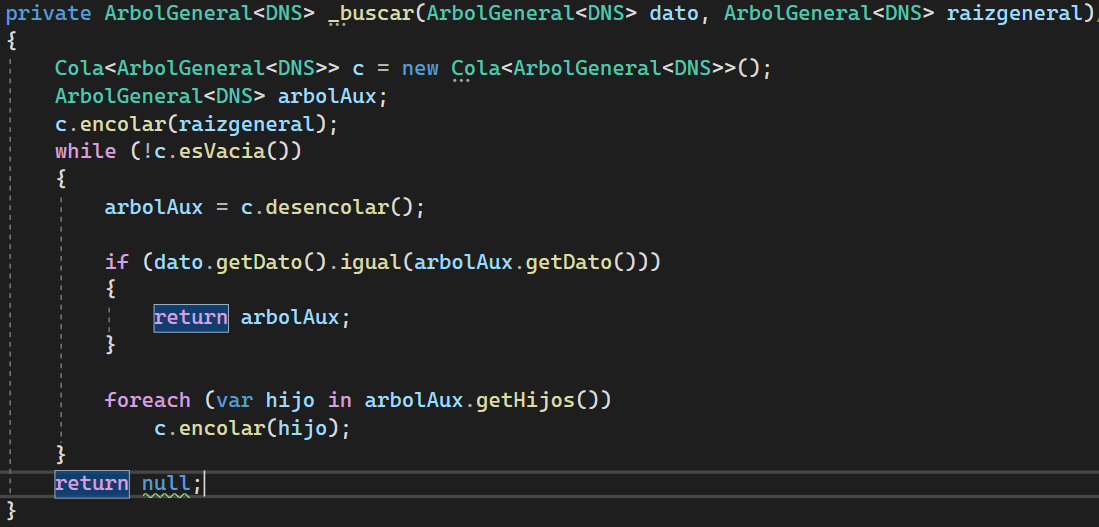
###### Para comprar los nodos, este método utiliza el método igual()

## Métodos auxiliares - \_buscar()

###### Cree este método para que me retorne un nodo (ArbolGeneral<DNS>), en caso que exista si no retorna un null, dentro del nodo raíz.

###### La finalidad de este método es para no agregar nodos repetidos en la “raizgeneral”, de este modo utilizo la referencia que retorna este método como raíz para que el agregado del siguiente nodo se hace sobre este mismo.

###### El código es el siguiente:



###### Lo que hace este método es realizar un recorrido por niveles y comparar cada nodo de la raiz “raizgeneral” con el nodo “dato”, ambos son de tipo ArbolGeneral<DNS> y se piden como parámetro y en caso de encontrarlo retorna una referencia del nodo que está en alguna parte de “raizgeneral” y en caso de no encontrarlo retorna null. Para comprar los nodos, este método utiliza el método igual()

## Métodos auxiliares – igual()

###### Cree este método para tener mayor eficiencia a la hora de comparar datos de tipo ArbolGeneral<DNS> y reemplazar el .Equals() que me estaba fallando.

###### El método .Equals() no me daba los resultados esperados y es porque no es capaz de diferenciar si un nodo es de dominio superior, hoja, etc. Buscaba datos muy iguales y en algunos casos solamente era necesario comparar la etiqueta del dato del ArbolGeneral<DNS>. Antes de crear este método pensé en hacer un override sobre el método .Equals() en la clase DNS para utilizarlo, pero no supe hacerlo así que desarrollé el método igual() que me ayuda a comprar los nodos según el tipo que es.

###### El código es el siguiente:

###### Lo que hace este método es primero averiguar que tipo de DNS se está comparando y luego hace la comparación y retorna true o false según el caso, por ejemplo si un nodo DNS superior se quiere comparar con un nodo DNS hoja, utiliza el último caso base el cual va a ser false, pero en caso de que ambos sean nodos DNS hoja se comparan por IP y siguiendo.

Conclusiones

## Conclusión final

###### La principal idea de este trabajo es usar la clase ArbolGeneral<T>, lo cual está bueno como planteamiento inicial, pero en lo personal a mí me dificultó porque tuve que hacer modificaciones en la clase DNS que pienso que debería haber sido hechas en la clase de ArbolGeneral<T> aunque entiendo perfectamente que la idea es usar la estructura de ArbolGeneral<T> que brindó la cátedra para demostrar lo genérico que puede llegar a ser.

###### De cualquier manera, haya tenido su dificultad o no, el trabajo cumple con lo requerido, las implementaciones finales que hice no son dificiles de hacer y de entender, si me costó pensarlas en un principio pero después de un rato pensarlas salen facilemente y las implementaciones al final salieron bien.

###### La depuración es una herramienta muy útil en la programación, me sirvió muchísimo y nunca antes había visto el potencial que tiene realmente, pienso que si no podría haber depurado no podría haber terminado el trabajo, hasta podría decirse incluso que lo que más aprendí con este trabajo es a depurar entre otras cosas.

## Mejoras

###### \*Para mejorar el trabajo, la primer idea es clara, agregar funcionalidades al módulo de consulta y arreglar el método Agregarse().

###### \*Podría haberse utilizado un patrón strategy para controlar mejor los métodos \_buscar() y \_existe(), en sí es el mismo método que difiere en un poco en el comportamiento.

###### \*También un patrón command para el tema de agregado.

###### \*También podría haberse utilizado un iterator para facilitar un poco los recorridos, aunque no es necesario porque los recorridos ya están implementados por parte de la cátedra.

###### \*Como el tema del agregado es un poco delicado podría hacer un sistema para detectar el largo de la url, por ejemplo, si una url se subdivide en 4 nodos distintos, tener un árbol que solamente haya urls de tamaño 4 sin contar el nivel 0 y en caso de que la url se subdivida en 3 nodos distintos, que se agreguen a un árbol distinto que tenga como mínimo y máximo 3 niveles, sin contar el nivel 0 que es la raíz

## Condiciones

###### Hice muchas pruebas para ver las condiciones de ejecución y el único problema que vi es en el tema de agregar nodos, cuando agrego un DNS [www.google.com](http://www.google.com) y un nodo [www.google.com.ar](http://www.google.com.ar) el programa funciona mal. Así que lo mejor es usar todas URLS de tamaño 4 o todas URLS de tamaño 3.

###### El resto del trabajo funciona bien, no encontré fallos.