Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-0113 Programación II

**Tarea Programada 4**

Documentación externa

Estudiantes:

* Sebastián Cruz Chavarría – B72458
* Roy Andrés Rojas Angulo – B76629

Prof. Edgar Casasola Murillo

I – 2018

Índice

Descripción del problema ……………………………………………………………………. 3

Descripción de la solución …………………………………………………………………….4

Diagrama de clases ……………………………………………………………………………6

Descripción del programa ……………………………………………………………………..7

Compilación del programa ……………………………………………………………………10

Casos de prueba ………………………………………………………………………………11

Lista de archivos entregados …………………………………………………………………14

Descripción del problema

En esta tarea programada se nos planteó como objetivo o problema programar un árbol binario equilibrado, más específicamente un árbol rojo-negro, además, se nos pidió que los pasos de inserción de pares ordenados en el árbol fueran visualizados mediante archivos de formato .svg. Para la solución de este problema fue necesario conocimiento sobre todos los temas vistos en clase, como lo son manejo de memoria, herencia, polimorfismo y plantillas. Este último es el que nos permite diseñar el árbol de manera tal que pueda recibir pares ordenados (llave, valor) de cualquier tipo, eso si la llave es un elemento comparable.

Primeramente, uno de los retos que surgieron para la resolución de esta tarea fue la implementación de un iterador que funcionará sobre el Arbol, la dificultad de este consiste en que se pueda mover entre padres e hijos directos y que no “brinque” a otras ramas.

Hubo también dificultad en la creación del algoritmo de equilibrio del Arbol, cómo se iban a detectar los casos en los que se debía aplicar rotaciones o cambios de color en los nodos y en qué orden.

Se tuvo que realizar ajustes en el diseño del SVG para la visualización del Arbol, ya que en algunas ocasiones ciertos nodos se sobreponían, imposibilitando comprender correctamente cómo se desplegaban las ramas.

Pero sin duda alguna el mayor problema fue poder instanciar el Arbol que se solicitaba en el archivo de texto o como argumento en la consola, ya que desde cualquiera de estas dos fuentes el tipo K de llave y V de valor se recibiría como una cadena de caracteres, la cual no se podría simplemente insertar en los campos correspondientes a K y V en los parámetros de la plantilla.

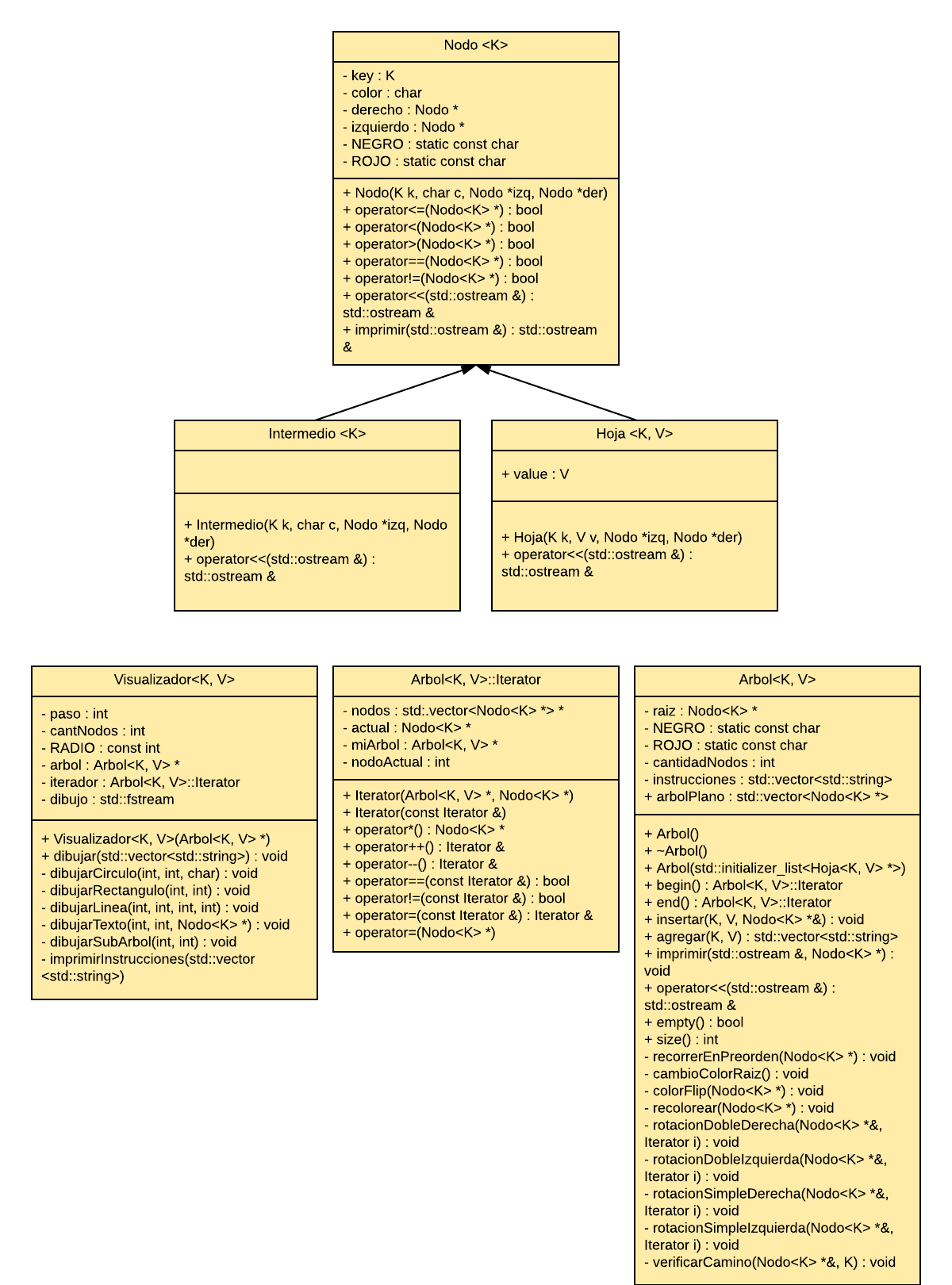
Descripción de la solución

Para la solución de este problema fue necesaria la implementación de cinco clases:

* En la clase Arbol se ubican todos los métodos necesarios para llevar a cabo su equilibrio, por supuesto también podemos encontrar las funciones con las cuales se permite agregar datos a este. Claramente es en este objeto donde convergen y se relacionan las demás clases, excepto tal vez Visualizador la cual más bien existe únicamente como amiga en Arbol para cumplir con sus funciones. Cabe destacar que en Arbol se encuentra anidada la clase Iterator, la cual había sido mencionada anteriormente debido a su forma de funcionar. Iterator es de suma importancia para hacer posible las rotaciones que requiere el Arbol en ciertos momentos.
* Existe la clase emplantillada abstracta Nodo, esta es base de las clases concretas Hoja e Intermedio. A grandes rasgos posee una llave K que puede ser de cualquier tipo (mientras ese tipo sea comparable), esta llave le da un “peso” al Nodo de tal manera que al insertar nodos al Arbol estos se agreguen de manera ordenada. Eso no implica equilibrio, sólo orden en la estructura para cumplir con la parte ordenada del árbol binario. Posee únicamente el método imprimir como virtual puro, pero este es suficiente para hacer de esta clase una de tipo abstracta, o polimórfica. También, implementa varias funciones de operadores relacionales que hereda a sus clases derivadas, dado que tanto un nodo Intermedio como una Hoja tienen un campo para su llave, se comparan por medio de esta variable.
* Intermedio, esta es una clase derivada de Nodo, la cual es también emplantillada, pero no padre de alguna otra. No agrega ningún atributo extra, a diferencia de Hoja. Forma parte vital de la estructura del Arbol, debido a la llave que contiene esta clase, sirve de mediadora para la búsqueda de algún dato en este, en el caso de que fuese necesario, así como para el ordenamiento del árbol.
* La clase Hoja derivada de Nodo alberga además de su llave un valor (V), el cual podemos relacionarlo como un dato. En su implementación del método imprimir se encarga de mostrar ambos, por su parte Intermedio sólo aporta la llave (K). Como su nombre lo indica, estos nodos son el final de sus ramas, y en el caso de esta tarea programada son los que contienen los datos del Arbol.
* Como última pero no menos importante se encuentra la clase Visualizador. Esta se encarga de elaborar un archivo SVG que contiene un dibujo del Arbol, y muestra los pasos que se llevan a cabo para insertar un dato u hoja. Por supuesto contiene todos los métodos necesarios para dibujar los Nodos, tanto Intermedios como Hojas, con sus correspondientes valores y colores.

El diseño de clases se implementó de esta forma dada la similitud entre las clases Intermedio y Hoja, siendo esta última la que contiene el campo para guardar el valor que representa, y una función de inserción de flujo donde mostrará su llave y su valor. Así, se aprovecharon las ventajas del polimorfismo, para tratar a todos los nodos como un Nodo de tipo K (Nodo<K>) genérico y poder manipular fácilmente los nodos del árbol en las operaciones que así lo requieran. En la figura 1 se aprecia mejor el diseño de la jerarquía de clases de Nodos, así como las otras clases de las cuales ya se ha hablado.

Diagrama de clases UML

Figura 1. Diagrama de clases UML.

Descripción del programa

Clase Arbol

Como se mencionó anteriormente la clase Arbol contiene todas las funciones necesarias para mantener tanto el orden, como el equilibrio de la estructura.

La función *recorrerEnPreorden()* se encarga de agregar todos los nodos del Arbol en un contenedor de tipo Vector, requiere de un único parámetro como puntero a Nodo, de la misma forma agrega estos al Vector (como punteros). Esta función trabaja de manera recursiva, primero moviéndose a todos los descendientes izquierdos del Nodo que recibe por argumento y después a los derechos.

Los siguientes son métodos para mantener el equilibrio del Arbol:

Realizan cambios de color:

*CambioColorRaiz()*, como su nombre lo indica se encarga de cambiar el color de la raíz del Arbol en caso de que exista y sea de color rojo.

*ColorFlip()* es el método encargado de identificar si ambos hijos de un Nodo (que es recibido por parámetro) son de color rojo, y por supuesto, realiza el cambio de colores correspondiente.

La función *recolorear()* verifica que el Nodo recibido por parámetro sea de color rojo, también, que los hijos sean de colores distintos (rojo-negro, negro-rojo). Si esto sucede, entonces recolorea de tal manera que el Nodo padre sea negro, y sus hijos rojos.

Realizan rotaciones:

A diferencia de los métodos que modifican colores, los siguientes suponen de antemano que la situación ideal para su ejecución ya está dada, por lo que no realizan verificaciones.

El método *rotacionDobleDerecha()* lleva a cabo una serie de instrucciones en las que realiza algo similar a una rotación simple hacia la izquierda, para que seguidamente se llame a la función *rotaciónSimpleDerecha().*

El método *rotacionDobleIzquierda()* trabaja de manera casi idéntica al anterior, obviamente la diferencia se encuentra en la “dirección” en la que “mueve” los nodos, siendo esta la contraria.

La función *rotacionSimpleDerecha()* recibe como parámetros: un puntero a Nodo y un *Iterator* por referencia. Este método realiza lo que indica su nombre, manipulando los punteros a los nodos hijos, pero además, hace uso del iterador para que este apunte al padre del Nodo que se obtuvo por argumento, y de esta manera, no perder ningún Nodo del Arbol.

Por su parte la función *rotacionSimpleIzquierda()* es homóloga a la que se acaba de explicar, realiza las modificaciones en los punteros con el objetivo de equilibrarlo hacia la izquierda.

Otra característica importante de esta clase es que en casi todos los métodos mencionados se realizan inserciones de strings a un atributo tipo Vector llamado instrucciones, que es utilizado en la clase Visualizador.

Otra función a destacar es *verificarCamino()*. Esta se asegura de que en el camino en el que se pretende insertar un Nodo se den los casos necesarios para llamar a los métodos de rotación. Recibe un Nodo (*raiz*) y una llave (*K*) para irse desplazando por el camino que tomará ese elemento a agregar en el Arbol, ese trabajo es logrado de manera recursiva pasando un nuevo Nodo raíz pero manteniendo la llave K.

El método insertar() agrega una nueva Hoja al Arbol. Lo hace respetando el orden establecido, si la llave de esta es menor a la llave de un Nodo, entonces se moverá a su izquierda, en caso contrario, a la derecha. Continuará así hasta encontrar el final de una rama, donde podrá agregarse. Antes de finalizar el método se invoca a *verificarCamino()* para corregir situaciones no permitidas en este tipo de Arbol.

Existe también la función *agregar()*. Esta se ocupa de sumar un nuevo elemento al Arbol, pasándole por parámetros la llave y el valor de esta nueva Hoja. Cada vez que esto se hace, se actualiza el Vector *arbolPlano* mencionado al inicio, el cual contiene punteros a todos los nodos que conforman el Arbol.

Clase Nodo

Clase abstracta, padre de Intermedio y Hoja. Posee los atributos color, llave (*key*), y un puntero a Nodo izquierdo y derecho.

Contiene varios métodos de comparación (<=, <, >, ==, !=) necesarios para agregar de manera debida cada Nodo a un Arbol. El único método virtual puro es *imprimir().*

Clase Intermedio

Representa a cualquier Nodo que no sea Hoja. Esta clase hereda de Nodo. Implementa la función pura *imprimir(),* en la cual se envía al flujo de salida el valor de la llave entre paréntesis redondos.

Clase Hoja

Esta clase también hereda de Nodo. Agrega un nuevo atributo llamado *value* que representa el dato que contiene esa Hoja. Al implementar *imprimir()* se envía al flujo de salida tanto la llave (que ya la tenía por herencia) como el valor.

Clase Visualizador

Como se mencionó anteriormente esta clase es la que se ocupa de crear un archivo SVG con el objetivo de visualizar el estado del Arbol cuando se le insertan nuevos Nodos. Posee varios métodos de dibujo:

El método *dibujarCirculo()* se encarga de dibujar los nodos tanto rojos como negros. Recibe por parámetros dos coordenadas (x, y) y un carácter que representa el color del cual será el Nodo.

Una función *dibujarRectangulo()* para representar exclusivamente nodos de tipo Hoja. Al igual que la anterior, recibe los mismos tres tipos de parámetro.

El método *dibujarLinea()* se encarga de trazar una línea desde un determinado punto de inicio hasta uno final. En el Arbol son las que unen los nodos.

*DibujarTexto()* introduce los valores correspondientes a un Nodo en su figura. Por ejemplo: En el caso de una Hoja, este método “escribiría” su llave y valor dentro del rectángulo que le pertenezca.

También se encuentra la función *imprimirInstrucciones()* que coloca al lado del Arbol resultante los pasos que se llevaron a cabo para conseguir ese estado del mismo.

La función llamada *dibujarSubArbol()* es recursiva, y se enfoca en plasmar en el archivo SVG un subArbol, que es entendido (en este método) como un Nodo padre con ambos hijos. Es la encargada de llamar a los demás métodos de dibujo de los cuales ya se ha hecho mención.

Por último la función principal nombrada *dibujar().* Se le ha asignado como tarea moverse por todo el Arbol (mediante el uso de *Iterator*) para así dibujarlo junto con las instrucciones ligadas, le corresponde también generar el archivo SVG.

Clase Cargador

Clase de suma importancia ya que identifica e instancia el tipo de Arbol que el usuario del programa requiere, más específicamente, define los tipos integrales en la plantilla de Arbol.

Para su funcionamiento hace uso de un único método denominado *generarArbol()* el cual se encarga de identificar los tipos (*tipoLlave y tipoValor*) que le han sido enviados por parámetro a Cargador, y generar ese Arbol en concreto. Recibe un objeto *fstream* con los pares ordenados que conforman cada Hoja, y una cadena de caracteres que representa el nombre del archivo HTML con todos los dibujos del árbol.

Compilación y ejecución del programa

La compilación en sí de nuestro trabajo no requiere de algún procedimiento especial, únicamente compilar el archivo ***main.cpp***.

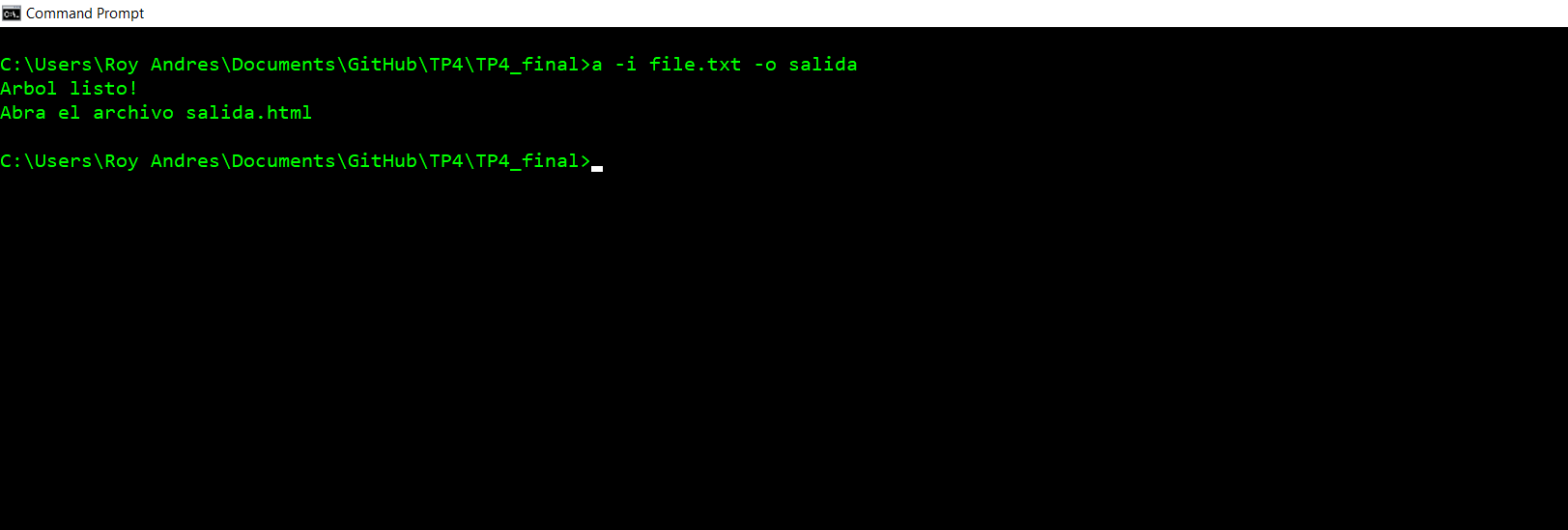
En el caso de nuestro programa decidimos que tomaría los nodos desde un archivo de texto, por lo que los argumentos a la hora de ejecutar tendrían el siguiente formato:

nombreProgramaCompilado **-i** nombreArchivo.txt **-o** nombreArchivoSalidaHTML

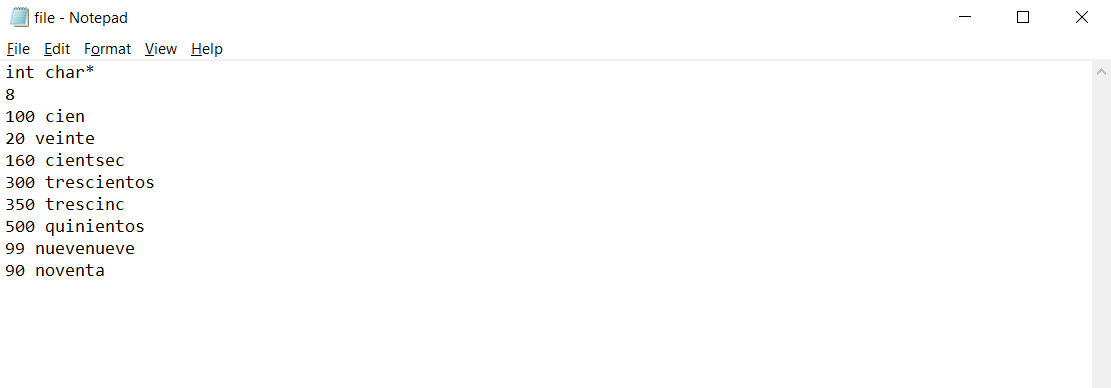
Lo necesario para ejecutar sería entonces el nombre del archivo (de extensión .txt) que contendrá los pares (llave, valor) a ingresar en el árbol binario rojo-negro, y el nombre que se le desee asignar a la salida HTML con la visualización completa.

Casos de prueba (salida esperada y obtenida)

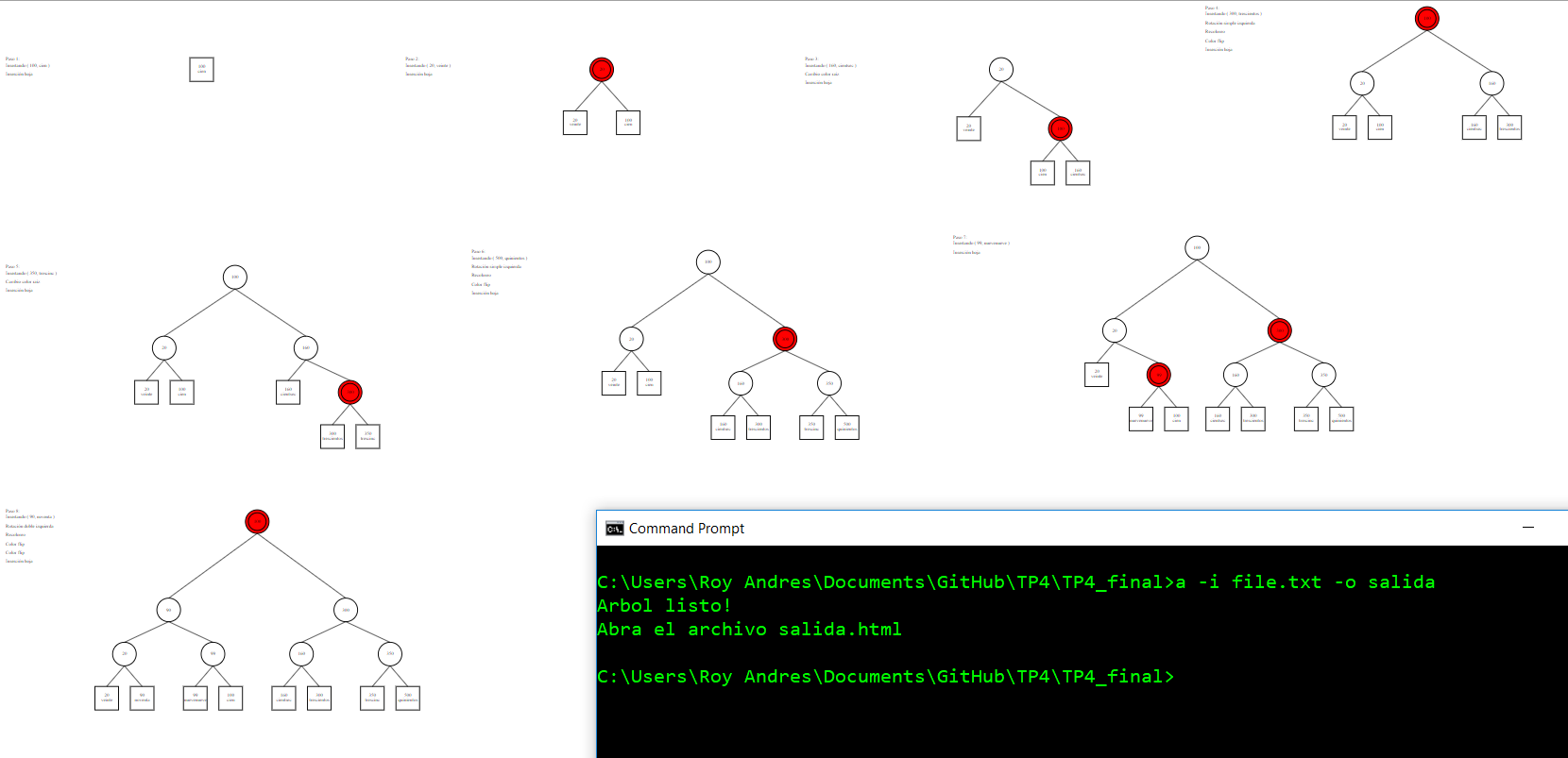
En la siguiente captura se realizó la ejecución del programa con los argumentos debidamente colocados.



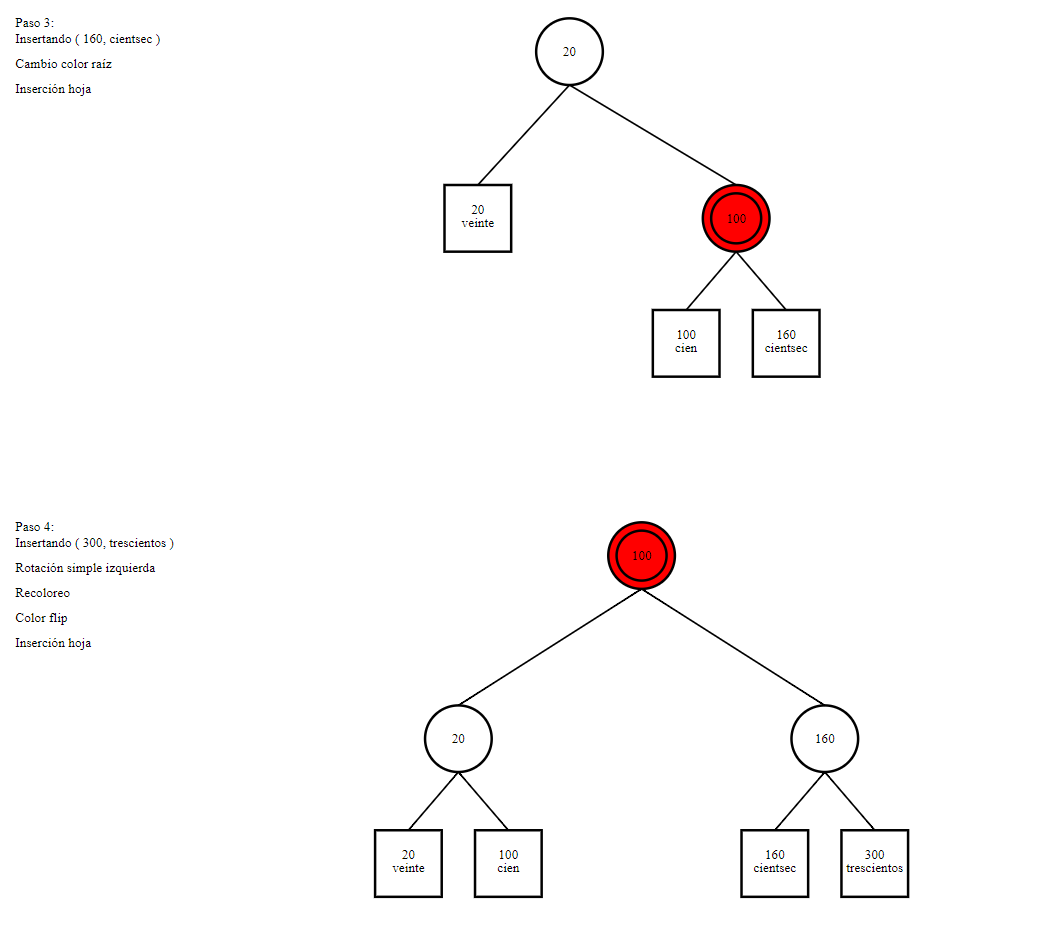
El archivo del que se recibieron los datos:



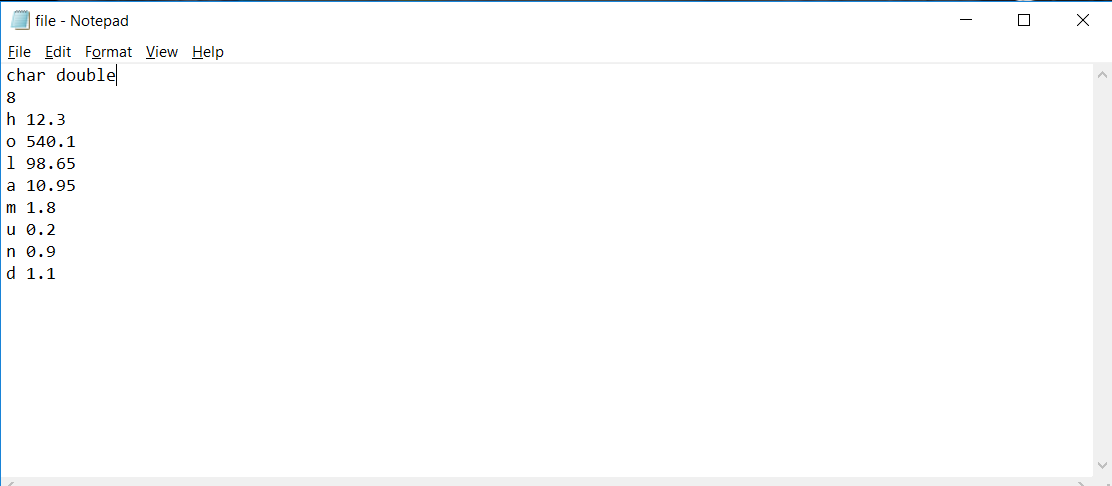
Se obtuvo la salida esperada, el Arbol terminó ordenado y equilibrado. Además, no hubo problema con la visualización SVG.



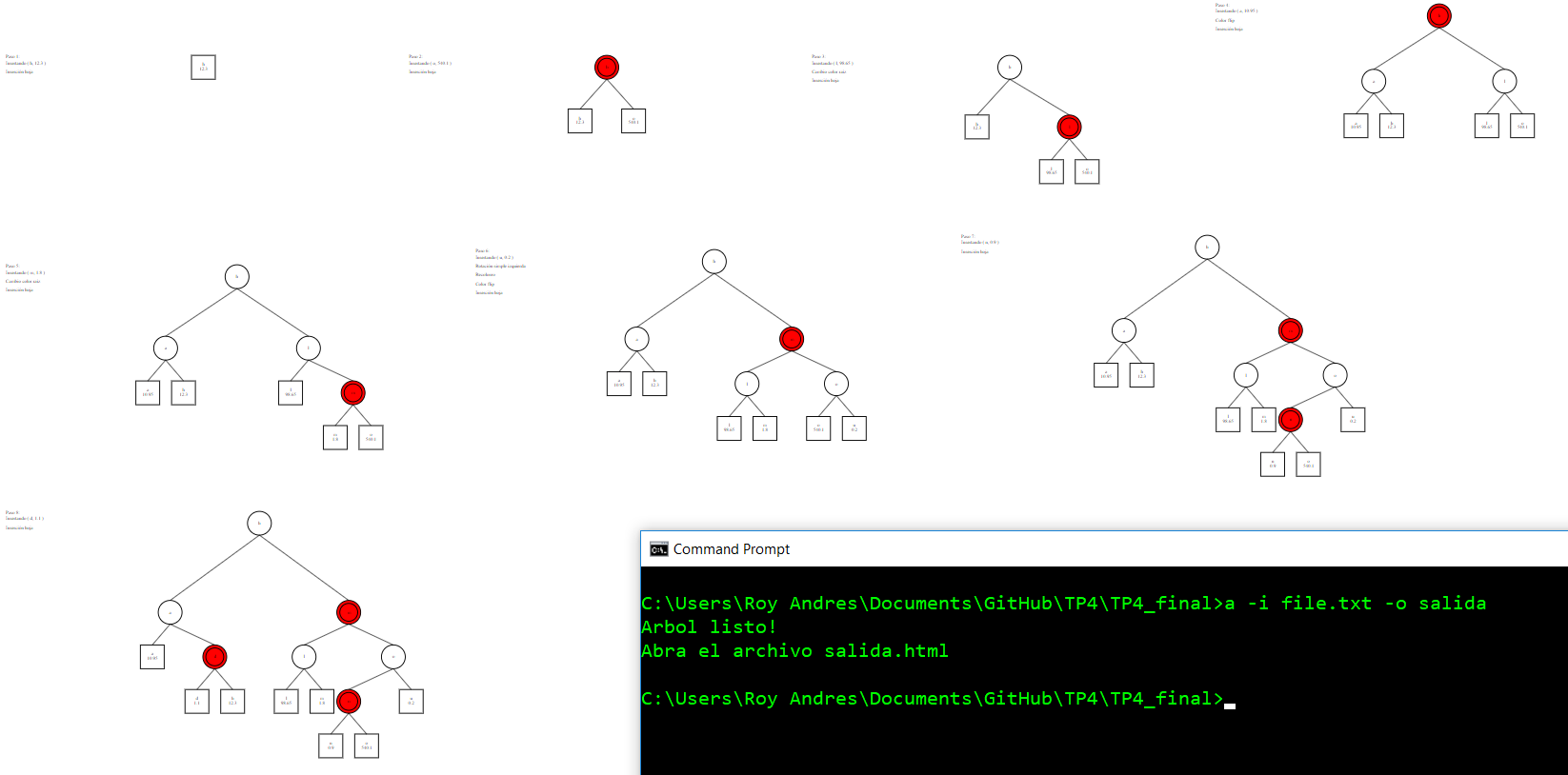
Aquí una vista más cercana de un par de los pasos realizados.



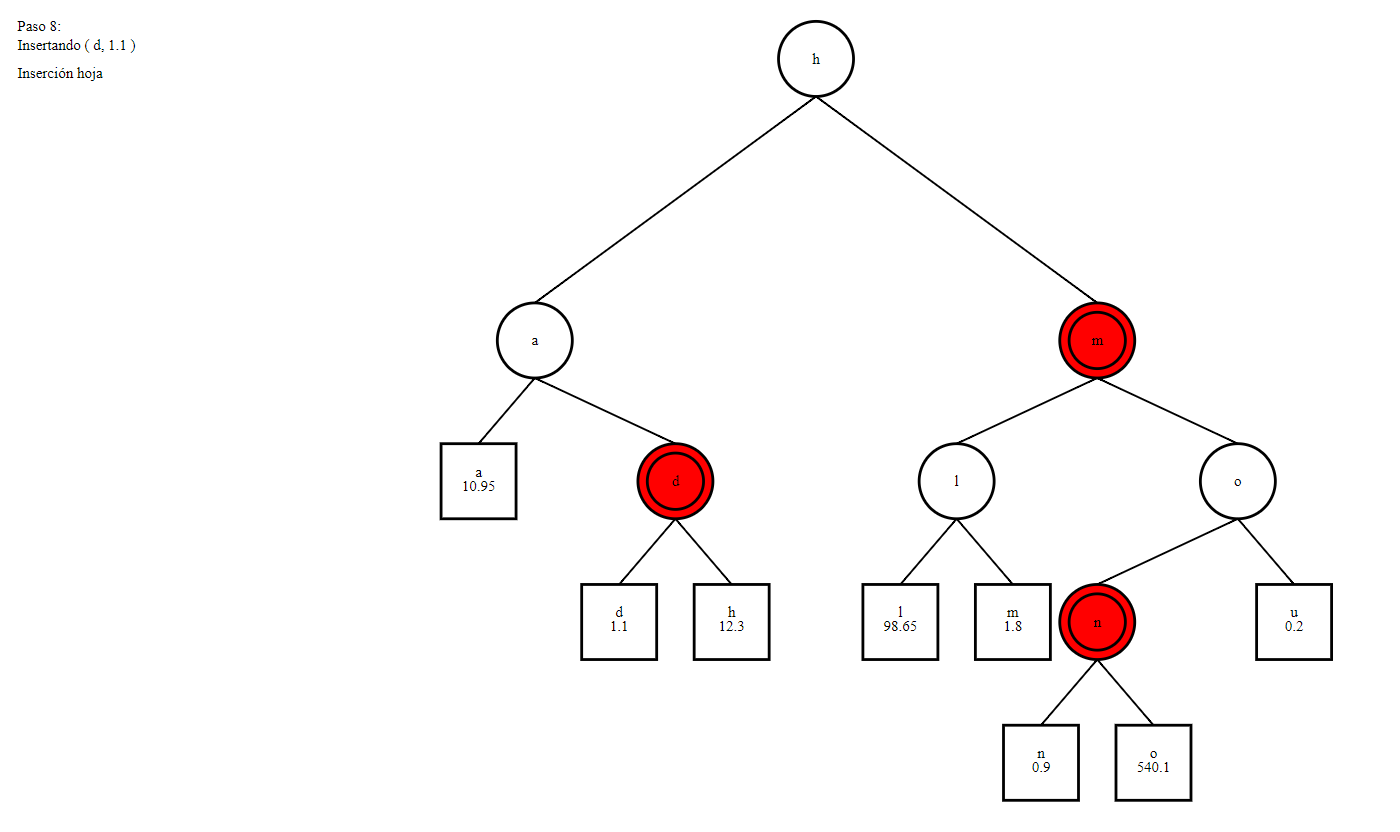
En el siguiente caso el archivo contendrá llaves de tipo *char* y valores *double*.



Este es el resultado obtenido, mismo que el esperado:



Este es el resultado del Arbol al insertar el último Nodo:



Lista de archivos entregados

-main.cpp

-Arbol.h

-Arbol.cpp

-Nodo.h

-Intermedio.h

-Hoja.h

-Cargador.h

-Visualizador.h

-Visualizador.cpp

-Documentacion\_Externa.pdf

-Diagrama TP4.png