Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Sistema para la Edición Gráfica de Árboles con Raíz

Trabajo Especial de Grado
presentado ante la ilustre
Universidad Central de Venezuela por la

Br. Yenny Fung Guan

para optar al título de
Licenciado en Computación

Tutor: **Prof. Héctor Navarro**

Caracas-Venezuela
2016

Universidad Centra de Venezuela
Facultad de Ciencias
Escuela de Computación

ACTA DE VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación de la Universidad Central de Venezuela, para evaluar el Trabajo Especial de Grado titulado **"Sistema para la Edición Gráfica de Árboles con Raíz"**, presentado por la bachiller **Yenny Fung Guan**, cédula de identidad **18.994.977**, como requisito académico para optar al título de Licenciado en Computación, mención Computación Gráfica, dejamos constancia de lo siguiente:

1.- Una vez leído el trabajo por los miembros del jurado, se fijó el día 21 de octubre de 2016 a las 02:00 pm, en la sala 1 del Centro de Computación Gráfica, Facultad de Ciencias de la Universidad Central de Venezuela, para que la aspirante presentara mediante una exposición oral y pública su contenido, respondiendo luego satisfactoriamente a las preguntas formuladas por el jurado.

2.- Finalizada la defensa del Trabajo de Grado, el jurado decidió **APROBARLO** con una calificación de 20 puntos.

3.- Para dar este veredicto, el jurado estimó que el trabajo examinado cumple con los requisitos exigidos para su aprobación en fe de lo cual se levanta la presente acta, el día 21 de octubre de 2016.

Ramirez Esmitt
C.I. 82.147.710
Jurado

Jaime Parada
C.I. 13.194.678
Jurado

Hector Navarro
C.I. 13.113.800
Tutor

Resumen

Los árboles son muy utilizados y estudiados en las ciencias de la computación, sirven como herramientas visuales para facilitar el análisis entre los elementos de un determinado estudio. Por lo general, para comprender los árboles es necesario el uso de figuras que demuestren el estado de estos. La creación de árboles basado en las habilidades propias del usuario o mediante distintos programas de edición gráfica tienden a consumir mayor costo, tiempo y riesgo a errores.

En la actualidad, existen tecnologías web que permiten el despliegue de gráficos directamente desde el navegador sin la instalación de complementos, como el uso de Canvas. Sin embargo, de acuerdo a la investigación realizada no existen aplicaciones disponibles en la Web que contemplen todos los requisitos mínimos y unificados para el proceso de creación y edición de árboles mediante una interfaz gráfica intuitiva. Por esta razón, se plantea como objetivo de la investigación desarrollar una aplicación web, aprovechando las bondades de ésta, que brinde un conjunto de herramientas que intervienen en el proceso de construcción de árboles de manera eficiente.

La solución fue sometida a una serie de pruebas cualitativas con el objetivo de determinar la experiencia de los usuarios en términos de utilidad, complejidad y cumplimiento de los objetivos de la investigación. Los resultados obtenidos indican que la aplicación cumple satisfactoriamente con lo esperado.

Palabras claves: Árboles, Despliegue de árboles, Editor de ádrboles, Aplicación web, Canvas.

Agradecimientos

Agradezco a **Carlos** por enseñarme a programar. No estoy segura si lo hubiera podido lograr sin él, de lo que sí es que definitivamente me ayudó a hacer de la programación mucho más sencilla.

Agradezco a **Ron** por su apoyo constante, por su interés y sobre todo por sus opiniones sinceras y muy exigentes en todos mis proyectos por más pequeños que sean.

Agradezco a **Michelle**, mi mejor amiga, por su apoyo desde tiempos inmemoriales y por tantos buenos momentos que compartimos durante y más allá de la carrera.

Agradezco a **Alejandro** por haberme fortalecido, presionado e incentivado para culminar este reto. Le agradezco todo el cariño.

Agradezco a **Miguel** por su apoyo y cada vez que me recuerda que todo es posible.

Agradezco a **Luiyit** por su confianza y enseñanza.

Agradezco a **Chucho** por haberme contagiado siempre de su alegría en esta larga jornada.

Agradezco a mis **padres** por el apoyo económico que me han brindado durante mis estudios.

Agradezco a los **profesores** y **preparadores** que aportaron significativamente en mi formación y que se han ganado mi admiración. A mi tutor de trabajo, **Hector**, por orientarme en la realización de este trabajo.

Agradezco a la **Universidad Central de Venezuela**, la mejor universidad del país, por la oportunidad y ser responsable de mi formación universitaria.

Finalmente, agradezco a todas aquellas personas, **amigos**, **panas**, **conocidos**, **colegas**, **hermanas** y demás que he olvidado mencionar que de alguna u otra manera me motivaron, acompañaron y ayudaron a culminar este objetivo.

Índice general

Índice de figuras

Índice de cuadros

Introducción

La teoría de grafos es un campo de estudio de las matemáticas y ciencias de la computación que estudia las propiedades de los grafos. Tiene sus inicios en el año 1736 por el matemático suizo Leonhard Euler. La idea principal en que se apoyaba su trabajo surgió del problema de los puentes de Königsberg, que consistía en encontrar un camino que recorriera los siete puentes del río. A partir de la solución del problema, Euler desarrolló algunos de los conceptos fundamentales de la teoría de grafos.

Un grafo, en resumen, es una colección de nodos y de arcos que unen estos nodos. El estudio de este documento se involucra con un tipo especial de grafo llamado árbol. Los árboles fueron utilizados por primera vez en el año 1847 por el físico prusiano Gustav Kirchhoff en su trabajo de redes eléctricas; luego, con la aparición de las computadoras digitales, nuevas aplicaciones fueron surgiendo.

Los árboles son usados como herramientas visuales para facilitar el análisis entre los elementos de un determinado estudio. La idea básica de modelar con árboles consiste en dibujar, analizar y resolver. Sin embargo cuando la tarea de dibujar se basa solo en las habilidades del usuario, el costo tiende a ser mayor y los resultados propensos a errores.

Por esta razón, se plantea como objetivo de la investigación, desarrollar una aplicación web, aprovechando las bondades de ésta, que permita el diseño de árboles de manera ágil con el objetivo de minimizar el costo del proceso y los errores.

Capítulo 1

Generalidades

Este capítulo, como continuación de la introducción, tiene la intención de justificar la investigación y el desarrollo del software. Para ello, se expone la problemática actual y se propone la solución a través de la definición de los objetivos de la investigación, la justificación e importancia del trabajo, el alcance y la delimitación.

1.1. Definición del problema

Los árboles son usados como modelos que permiten expresar de forma visual y sencilla las relaciones entre los elementos de un determinado estudio. Pueden ser utilizados durante el dictado de clases, en redacción de documentos, o en cualquier otra situación que ayude a simplificar la resolución de un problema de naturaleza jerárquica. Existen también, diversos algoritmos que a pesar de no estar relacionados directamente con los árboles, pueden comprenderse mejor al explicarse mediante éstos.

La idea básica de modelar con árboles consiste en dibujar, analizar y resolver. Sin embargo cuando la tarea de dibujar (a mano alzada) se basa en las habilidades y en la percepción del usuario, los árboles complejos suelen verse comprometidos.

Las soluciones encontradas son programas que pueden clasificarse muy sencillos o muy complejos. Muy sencillos porque cumplen con los requisitos básicos mínimos para crear árboles, y muy complejos porque son de propósito general; disponen de funcionalidades (en las que el usuario debe tomar todas las decisiones de coordenadas) para abarcar la construcción de diagramas de todo tipo e innecesarias para dibujar árboles.

Desarrollar una actividad con las soluciones descritas incrementa el riesgo de obtener resultados que tienden a tener un mayor costo y tiempo, de ser propenso a errores y de ser un proceso tedioso, se disminuye la legibilidad del modelo entorpeciendo el análisis del estudio en cuestión.

1.2. Objetivos

1.2.1. Objetivo General

Desarrollar un sistema que permita la representación y manipulación gráfica de árboles con raíz utilizando las herramientas y las tecnologías Web.

1.2.2. Objetivos específicos

- Implementar el conjunto de herramientas necesarias para la elaboración y modificación de árboles.
- Desarrollar un algoritmo eficiente que permita automatizar la posición de los nodos de los árboles.
- Proveer un mecanismo que permita salvaguardar los árboles generados para su posterior despliegue y modificación dentro de la aplicación.
- Permitir la exportación de los árboles a diferentes formatos para su utilización fuera de la aplicación.
- Gestionar el uso de cuentas de usuario para el acceso al sistema.
- Diseñar e implementar una base de datos relacional para el almacenamiento de los árboles y cuentas de usuario.
- Elaborar el diseño de las interfaces gráficas de usuario y definir las interacciones del usuario con el sistema.
- Realizar las pruebas del sistema.

1.3. Justificación

Los árboles, o grafos en general, son importantes porque permiten expresar de forma visualmente sencilla las relaciones entre los elementos de un estudio, facilitando así, de manera práctica y confiable la resolución de problemas o toma de decisiones.

El desarrollo del sistema proveerá una interfaz gráfica de usuario para dibujar y editar árboles de manera fácil e intuitiva, logrando acortar los tiempos de construcción y disminuyendo los riegos a errores.

Además de proveer las herramientas para el dibujado, realizará los cálculos correspondientes para ubicar de forma sistematizada los elementos que lo componen, es decir, ajustará apropiadamente los espacios entre nodos para evitar intersecciones. Al relevar al usuario de esta responsabilidad, puede ahorrarle el tiempo y el costo que le tomaría calcular las medidas por su propia cuenta.

La aplicación dispondrá de gestión de cuentas de usuario, que permitirá el almacenamiento de los grafos creados por cuenta; el salvaguardado de los documentos permitirá que persistan y puedan ser modificados a lo largo del tiempo.

El sistema desarrollado como una aplicación web, ofrecerá la ventaja de requerir únicamente la instalación de un navegador web; un cliente ligero y permite la facilidad para actualizar y mantener aplicaciones sin distribuir e instalar software a muchos usuarios.

1.4. Alcance y delimitación de la investigación

La investigación está dedicada al desarrollo de un sistema para la graficación de árboles con raíz, además de la administración de estos árboles bajo cuentas de usuarios.

El sistema dispondrá de una interfaz gráfica de usuario donde se crearán y modificarán los árboles. Éstos serán imágenes en mapa de bits que deberán graficarse haciendo uso del elemento canvas en un contexto de dos dimensiones.

La interactividad del sistema permitirá a los usuarios manipular los componentes del árbol haciendo uso de los dispositivos de entrada estándar (teclado y/o ratón) en tiempo real.

El sistema deberá proveer al usuario la creación de una cuenta, en la que podrá almacenar y administrar (crear, ver, editar y eliminar) los árboles generados. La creación de cuenta será una herramienta opcional para el usuario.

Para acceder al sistema se deberá solicitar la petición al servidor correspondiente mediante un navegador web. Los navegadores que abarcarán las pruebas de desarrollo serán Firefox versión 36.0.1, Safari versión 7.1.5, Chrome versión 41.0 e Internet Explorer versión 11. Se recomienda utilizar los navegadores a partir de las versiones mencionadas o posteriores.

Capítulo 2

Marco Teórico

En este capítulo se expone las bases teóricas que permiten respaldar el desarrollo del trabajo. Además de exponer la teoría, se presenta una revisión de trabajos anteriores relacionados. La investigación teórica previa a la experimentación ubica al investigador en un correcto encuadre dentro del proceso, y le sugiere cuales son los problemas sin soluciones y que son objetos de estudio.

El capítulo se divide en tres secciones principales, la primera, "Árbol con raíz", trata de un tipo especial de grafo llamado árbol (con raíz). Aquí, se contextualiza las propiedades básicas de grafos y de árboles. Para cerrar se ejemplifica algunas de sus utilidades en la realidad.

La segunda sección, "Aplicaciones disponibles para el dibujado de árboles", está dedicada a la investigación de aplicaciones disponibles para el dibujado de árboles con raíz. Por cada aplicación se puntualiza los lenguajes utilizados para su desarrollo, las opciones con las que cuenta la interfaz y sus principales desventajas. Los detalles de implementación de código serán obviados.

La tercera y última sección,"Contexto tecnológico", hace un enfoque teórico a las herramientas tecnológicas y de los lenguajes requeridos para aplicar en el desarrollo práctico del trabajo.

2.1. Árbol con raíz

2.1.1. Una rápida introducción a la teoría de grafos

2.1.1.1. Grafo

Un **grafo** G se define como un par *(V, E)*, donde V es un conjunto finito no vacío cuyos elementos son denominados vértices o nodos, y E es un conjunto de pares (no ordenados u ordenados) extraídos de la colección de los elementos de V, que reciben el nombre de

aristas o arcos. De manera informal, un grafo es una colección de nodos y de arcos que unen estos nodos [10].

La Figura 2.1 muestra la representación gráfica del grafo G=($\{a,\ b,\ c\}$, $\{\{a,\ c\}$,$\{c,\ b\}\}$).

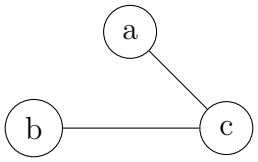


Figura 2.1: Grafo

Sea G un grafo, un **subgrafo** de G es un grafo cuyo conjunto de vértices y aristas son subconjuntos de los vértices y aristas de G, respectivamente [10].

La Figura 2.2 muestra un grafo S=($\{a,\ b,\ c\}$, $\{\{a,\ c\}\}$), que es a su vez un subgrafo del grafo G de la Figura 2.1.

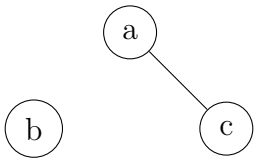


Figura 2.2: Subrafo

2.1.1.2. Grafo dirigido y no dirigido

Sea G=(V, E) un grafo. Un grafo G es **dirigido o digrafo** si E es un conjunto de pares ordenados sobre V. Para cualquier arista (a, b), se dice que la arista es incidente con los vértices a, b; a es adyacente hacia b, mientras que b es adyacente desde a. Además, el vértice a es **origen o fuente** y el b es el **término o vértice terminal** [10].

En un digrafo el **grado de entrada** (ge) de un vértice es el número de vértices adyacentes hacia él y el **grado de salida** (gs) es el número de vértices adyacentes desde él. En el grafo de la Figura 2.3 (a), el grado de entrada del vértice c es 1 y el grado de salida 2.

grado de entrada de c = ge(c) = 1 grado de salida de c = gs(c) = 2

Sea G=(V, E) un grafo. Un grafo G es **no dirigido** si E es un conjunto de pares no ordenados sobre V. Sus arcos no poseen dirección. Una arista como (a, b) representa (a, b), (b, a) [10].

En la Figura 2.3 se muestra un grafo (a) dirigido y un grafo (b) no dirigido. Para cada grafo, la arista (d, d) es un **lazo** por ser una arista que incide con el mismo vértice, y e es un vértice **aislado** por no tener aristas incidentes.

En general, cuando no se especifique si un grafo G es dirigido o no, se supondrá que es un grafo no dirigido.

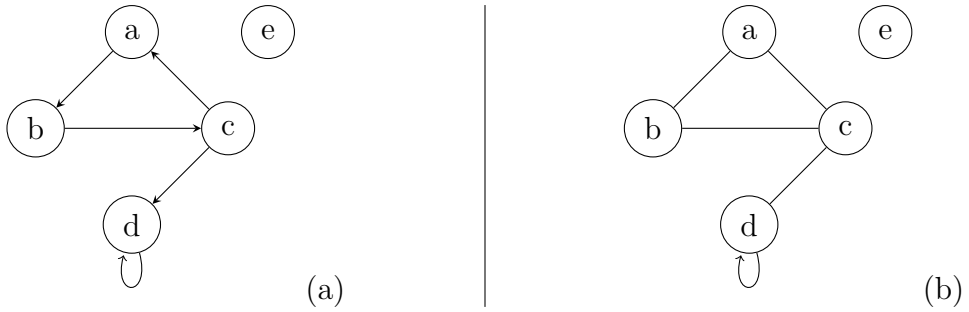


Figura 2.3: Grafo dirigido y no dirigido

2.1.1.3. Camino

Sean x, y vértices (no necesariamente distintos) de un grafo $G=(V, E)$. Un **camino** x-y en G es una sucesión alternada finita (sin lazos) entre n elementos de V y de E de la forma

$$x = v_0,\ e_1,\ v_1,\ e_2,\ v_2,\ \ldots,\ v_{n-1},\ e_n,\ v_n = y$$

donde $v_i \in V$ para $i=0, \ldots, n$ y $e_j \in E$ para $j=1, \ldots, n$. v_{i-1} es adyacente con v_i mediante el arco e_i. Tenga en cuenta que en un camino se puede repetir aristas y vértices (sin lazos) [10].

La cantidad de aristas que compone un camino define la longitud del mismo. Si la longitud es 0, no existen aristas, $x=y$, y el camino se denomina trivial [10].

Para nombrar la ruta del camino se puede enumerar solo las aristas o solo los vértices (si el otro queda determinado claramente). Por ejemplo, $\{a, b\}$, $\{b, c\}$, $\{c, d\}$ es un camino a-d de longitud 3 del grafo de la Figura 2.4.

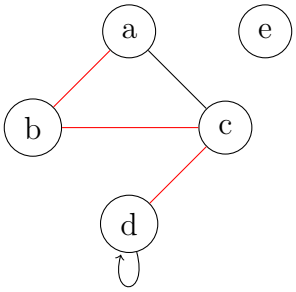


Figura 2.4: Camino en grafo

2.1.1.4. Camino cerrado y abierto

Sea un camino x-y en un grafo $G=(V, E)$. Se dice que el camino x-y es un **camino cerrado**, si $x=y$, es decir, si sus extremos coinciden. En caso contrario, es un **camino abierto** [10].

2.1.1.5. Camino simple

Sea un camino x-y en un grafo $G=(V, E)$. Un camino en G es un **camino simple** o sencillo si todos los vértices que lo componen son distintos [10].

2.1.1.6. Ciclo

Sea un camino x-y en un grafo $G=(V, E)$. Se dice que el camino x-y es un **ciclo**, si es un camino simple y cerrado, es decir, todos los nodos que lo componen son distintos salvo el primero y el último [10].

Un ciclo de longitud n se denomina **n-ciclo** [1].

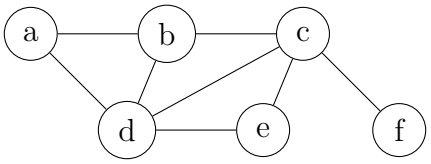


Figura 2.5: Caminos en grafo

Algunos ejemplos de caminos para el grafo de la Figura 2.5 se muestran en el Cuadro 2.1:

	Camino	Longitud	Abierto/ cerrado	Camino simple	Ciclo
1	$\{a, b\}$, $\{b, d\}$, $\{d, c\}$, $\{c, e\}$, $\{e, d\}$, $\{d, b\}$: camino a-b	6	Abierto	No (se repite vértices d y b)	No
2	$b \to c \to d \to e \to c \to f$: camino b-f	5	Abierto	No (se repite vértice c)	No
3	$\{f, c\}$, $\{c, e\}$, $\{e, d\}$, $\{d, a\}$: camino f-a	4	Abierto	Sí	No
4	$\{b, c\}$, $\{c, d\}$, $\{d, b\}$: camino b-b	3	Cerrado	Sí	Sí

Cuadro 2.1: Caminos del grafo de la Figura 2.5

Como el grafo no es dirigido, el camino 1 también es un camino b-a (leyendo las aristas como $\{b, d\}$, $\{d, e\}$, $\{e, c\}$, $\{c, d\}$, $\{d, b\}$, $\{b, a\}$). Similarmente ocurre para los caminos 2, 3 y 4.

2.1.1.7. Grafo conexo y no conexo

Un grafo G es **conexo** si para cualquier par de sus nodos existe un camino simple. En caso contrario, es un grafo **no conexo o disconexo**. Un subgrafo de un grafo disconexo G se dice que es una **componente** de G si es, en sí mismo, un grafo conexo [10].

En la Figura 2.6 se tiene un grafo disconexo y dos componentes conexas.

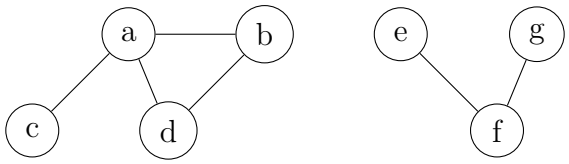


Figura 2.6: Grafo disconexo con dos componentes conexas

2.1.2. Conceptos y fundamentos de árboles con raíz

2.1.2.1. Árbol

Sea $G=(V, E)$ un **grafo no dirigido sin lazos**. Se dice que G es un **árbol**, si es **conexo y no contiene ciclos** [10].

En la Figura 2.7, el grafo (a) es un árbol, pero el grafo (b) no, pues contiene el ciclo $\{a, b\}$, $\{b, c\}$, $\{c, a\}$. El grafo (a) es un árbol, y un subgrafo de (b) que contiene todos sus vértices, en este caso, el grafo (a) es un **árbol recubridor** del grafo (b). Más adelante, se estudiará con mayor detalle sobre los árboles recubridores [10].

El grafo (c) no es conexo, por lo que no puede ser un árbol. Sin embargo, cada componente sí es un árbol, en este caso se tiene un **bosque**; un conjunto de árboles. El grafo (c) es un **bosque recubridor** para el grafo (d) [10].

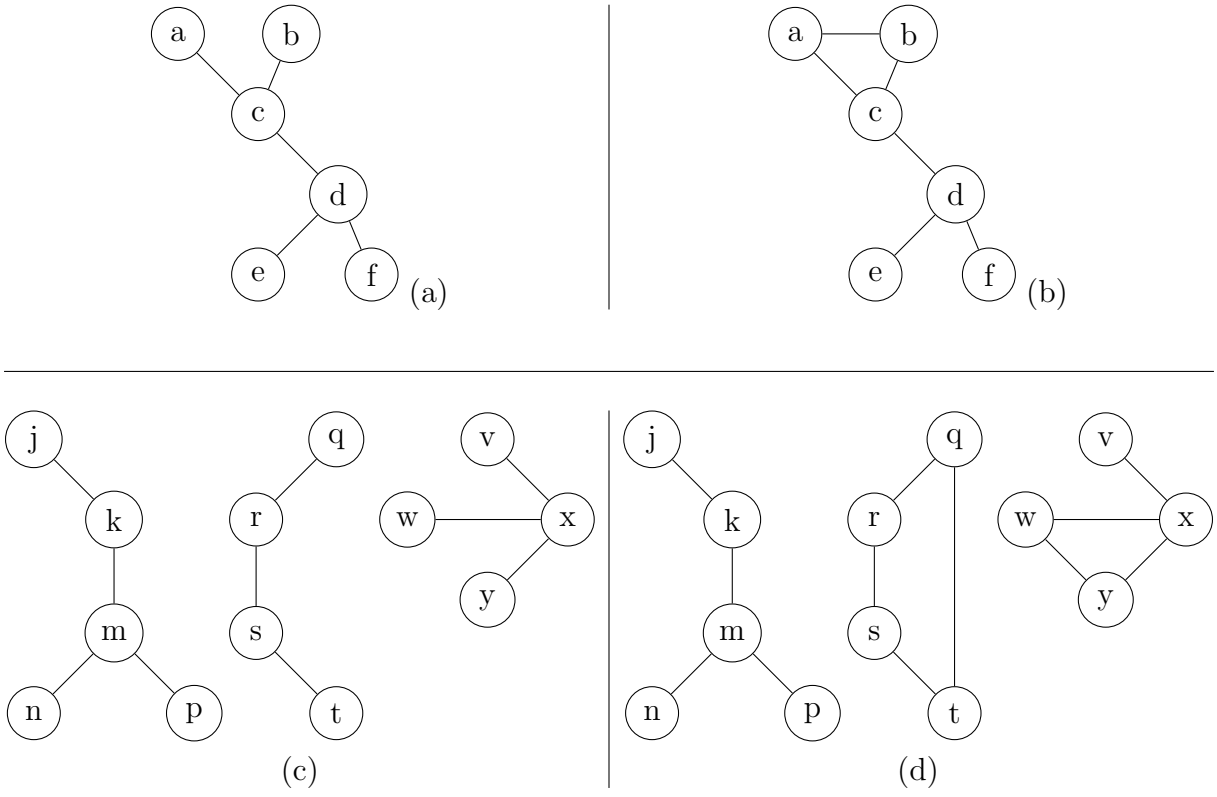


Figura 2.7: Árbol y bosque recubridor

2.1.2.2. Árbol con raíz

Si G es un grafo dirigido, entonces G es un **árbol dirigido** si el grafo no dirigido asociado (**grafo que no se le consideran las direcciones de las aristas**) a él es un árbol [10].

Si G es un árbol dirigido, G es un **árbol con raíz** si existe un único vértice r llamado raíz, tal que el grado de entrada de r=ge(r)=0 y para el resto de los vértices v, ge(v)=1 [1]. En la Figura 2.8, el grafo (a) es un árbol dirigido y el grafo (b) es un árbol que tiene como raíz al vértice r [10].

Un árbol también recibe el nombre de árbol con raíz si un vértice ha sido designado raíz. En este caso las aristas tienen una orientación natural hacia o desde la raíz.

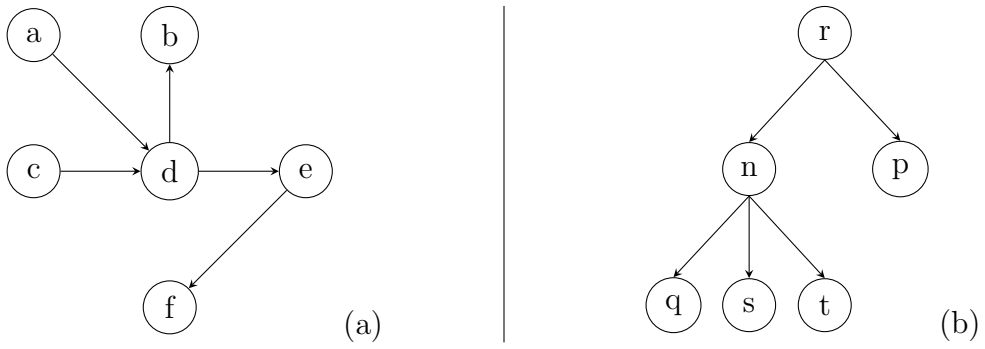


Figura 2.8: Árbol dirigido y árbol con raíz

Se trazará los árboles con raíz como en la Figura 2.8 (b), pero por convenio de que **las direcciones van del nivel superior hacia el inferior, las flechas no serán necesarias**. Además, quedará tácito que **los árboles que se mencionarán de ahora en adelante serán con raíz**.

2.1.2.3. Terminologías según la anatomía de un árbol

Sea un árbol con raíz, con un vértice v

Raíz : nodo con ge(v)=0; que no posee padre.

Padre : nodo con gs(v)>0; que tiene uno o más hijos.

Hijo : nodo con ge(v)>0; que es descendiente de otro nodo.

Hermanos : nodos con el mismo padre.

Hoja o terminal : nodo con gs(v)=0; que no posee hijos.

Interno o ramificación : nodo con gs(v)>0; que no es hoja.

Rama : conexión entre dos nodos del árbol que representa una relación de jerarquía (los aristas o arcos del grafo).

2.1.2.4. Terminologías según la morfología de un árbol

Sea un árbol con raíz, con un vértice v

Ascendientes de un nodo: todos los nodos en el camino desde la raíz del árbol hasta ese nodo.

Descendientes de un nodo: todos los nodos en el camino que parten desde ese nodo.

Nivel de un nodo: la suma del número de arcos que debe ser recorrido entre el nodo y la raíz (o la longitud del camino entre el nodo y la raíz). Por definición la raíz tiene nivel 0.

Nivel de un árbol: el máximo de los niveles de los nodos de un árbol.

Grado de un nodo: número de nodos hijos que tiene dicho nodo (grado de salida).

Grado de un árbol: el máximo de los grados de los nodos de un árbol.

Subárbol: es un subgrafo de un grafo que además es un árbol.

2.1.3. Tipos de árboles

2.1.3.1. Árboles binarios

Un árbol con raíz es **binario** si para cada vértice v, gs(v)=0, 1 o 2, es decir, si tiene a lo sumo dos hijos [10].

Si gs(v)=0 o 2, entonces el árbol con raíz es **binario completo** [10].

En la Figura 2.9, (a) muestra un árbol binario y (b) muestra un árbol binario completo.

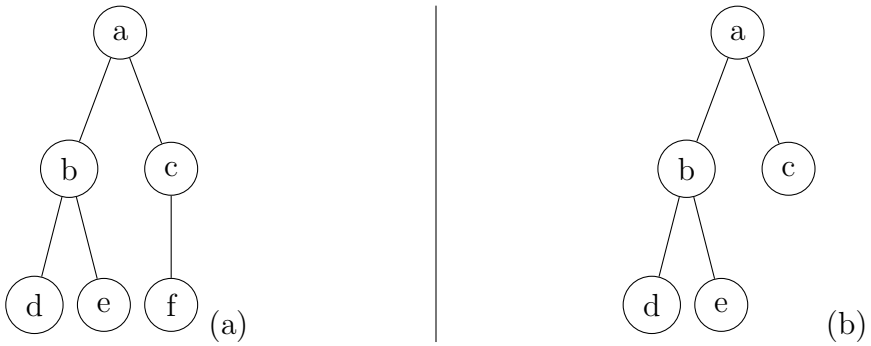


Figura 2.9: Árbol binario y árbol binario completo

2.1.3.2. Árboles n-arios

Un árbol con raíz es **n-ario** si para cada vértice de v, gs(v)<=n, siendo n un número entero positivo, es decir, cada nodo tiene a lo sumo n hijos. En el caso de n=2, resulta también un árbol binario [10].

Si gs(v)=0 o n, entonces es un árbol con raíz **n-ario completo**, si n=2, entonces es un árbol binario completo [10].

En la Figura 2.10, (a) muestra un árbol 3-ario y (b) nuestra un árbol 3-ario completo.

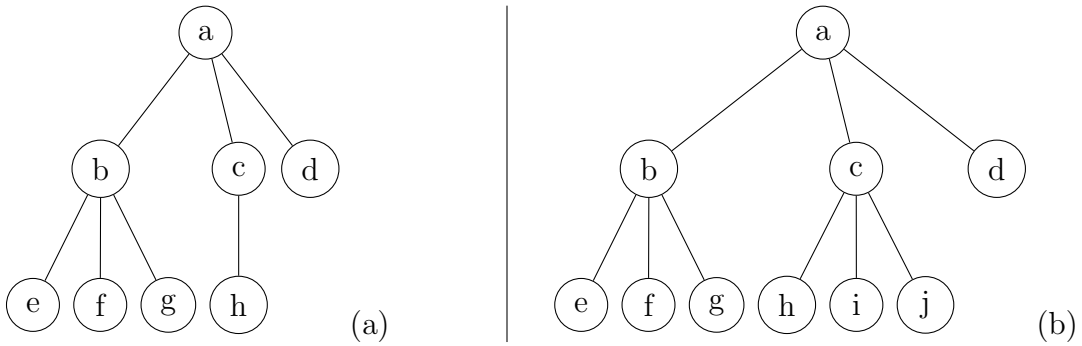


Figura 2.10: Árbol 3-ario y árbol 3-ario completo

2.1.4. Aplicaciones de árboles

Los árboles pueden ser aplicados a diversos campos, desde áreas de informática, estudios de circuitos electrónicos a problemas o situaciones de la vida real. La aplicación consiste en el modelado de la información o la abstracción de los procesos que ocurren en la realidad con la finalidad de reproducirlos en sistemas formales, es decir, a través de estructuras matemáticas (árboles) reconocibles por el ser humano. La información una vez descrita en estructuras matemáticas, permite dar soluciones más efectivas a los problemas planteados.

Algunas de las aplicaciones que se pueden enumerar son las siguientes:

- Búsqueda sistemática de información, una de las aplicaciones más comunes de árboles.
- Almacenamiento de información de naturaleza jerárquica. Por ejemplo, el almacenamiento de información del sistema de archivos de un computador; en este sistema la estructura de directorio normalmente suele ser ramificada, o la taxonomía biológica que clasifica en orden sistemático y jerarquizado la diversidad de los seres vivos.
- Manejo de estructuras de datos en ciencias de la informática. La idea de una estructura de datos árbol es que la colección de los elementos que organiza tenga la apariencia de un árbol de grafo.

- Disposición física en la que se conectan una red de ordenadores para intercambiar datos, es decir la topología de red.

2.2. Aplicaciones disponibles para el dibujado de árboles

2.2.1. Editores web

HTML es un lenguaje de marcas de hipertexto para la elaboración de páginas web. Al código HTML se le puede inscrustar scripts, generalmente Javascript; un lenguaje de programación interpretado utilizado principalmente en el lado del cliente (existe también del lado del servidor) para añadir mejores en la interfaz de usuario y páginas web dinámicas.

Las aplicaciones web tienen la ventaja de solo requerir la instalación de un navegador como cliente ligero para ser interpretadas. No dependen del sistema operativo y tienen la facilidad de poder realizar actualizaciones y mantenimientos sin la necesidad de distribuir e instalar software.

Entre las aplicaciones web más destacadas dedicadas al dibujado de árboles con raíz que se encontraron durante la investigación son:

2.2.1.1. JS Tree Graph

JS Tree Graph es un software de código abierto escrito por Cristhian Fernández y publicado en el año 2012 [8].

Hecho completamente en HTML4 y JavaScript puro, sin incluir bibliotecas de terceros. Básicamente, utiliza los elementos div para componer el árbol.

JS Tree Graph posee una interfaz muy sencilla, ver Figura 2.11, que permite una interacción muy precaria con el usuario. Las opciones con las que cuenta para editar el árbol en tiempo real son las siguientes:

- Agregar un hijo derecho.
- Modificar la orientación del árbol: horizontal o vertical.
- Contraer o expandir los hijos de un nodo.
- Mostrar información del nodo mediante una ventana emergente.

La desventaja principal es que requiere modificar el código de la aplicación para poder añadir la raíz y el valor de los nodos. Aunque para ello solo se necesite un mínimo de conocimiento sobre el código, la aplicación se limita a un grupo de usuarios que manejen el lenguaje.

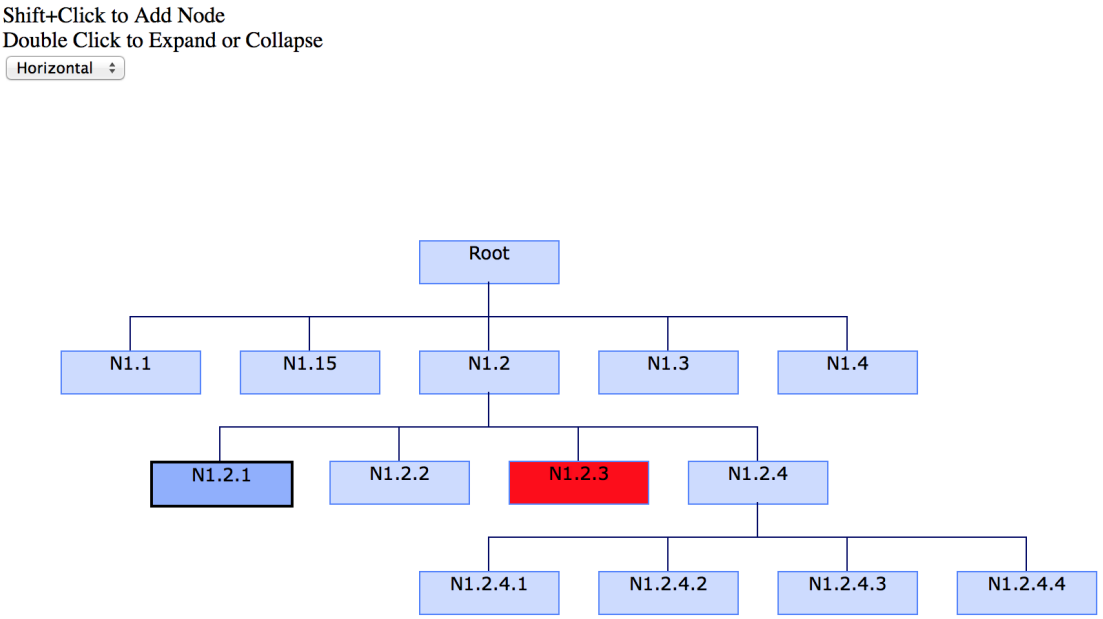


Figura 2.11: Interfaz de JS Tree Graph

2.2.1.2. Tree Graph

Tree Graph es un software de código abierto escrito por Rodrigo Cesar de Freitas Dias y publicado en el año 2012 [9].

Tree Graph está hecho en HTML5 y JavaScript puro, sin ninguna otra dependencia, utiliza el elemento canvas para dibujar el árbol e incluir efectos de animación.

Tree Graph con una interfaz muy sencilla, ver Figura 2.12, permite una interacción muy precaria entre el usuario y la aplicación. Las opciones con las que cuentas para editar el árbol en tiempo real son las siguientes:

- Trasladar el árbol.
- Contraer y expandir los hijos de un nodo
- Mostrar información detallada de un nodo mediante una ventana emergente

La desventaja principal es que requiere modificar el código de la aplicación para añadir nodos y sus valores. Por esta razón, la aplicación se limita a los usuarios que desconocen la implementación del mismo.

2.2.1.3. TreeView

TreeView es un software de código abierto escrito por el profesor Hector Navarro y publicado en el año 2013 [17].

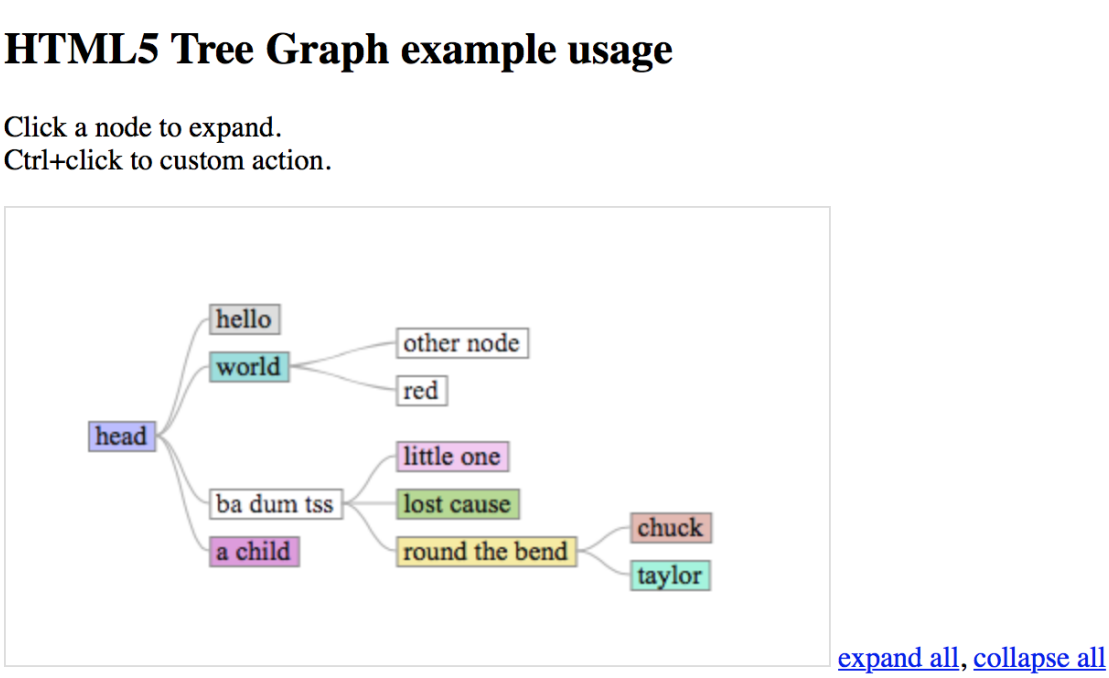


Figura 2.12: Interfaz de Tree Graph

TreeView está escrito en HTML, Javascript, PHP y C++. Para construir el árbol se ejecuta un programa C++ del lado del servidor que genera un documento que contiene el árbol en formato SVG, mediante el uso del lenguaje de progrmación web PHP se envía el documento al lado cliente y se inscrusta el SVG al HTML.

Tree Graph posee una interfaz muy sencilla compuesta principalmente por dos partes, ver Figura 2.12, una de entrada en la parte superior y una de salida en la inferior. En la entrada se tiene un área de texto para definir el árbol y en la salida muestra la figura.

Las opciones con las que cuentas para editar el árbol en tiempo real son las siguientes:

- Definir el árbol mediante un formato sencillo (directamente en la aplicación a diferencia de las anteriores).
- Exportar el árbol en formato SVG.

A pesar que permite construir el árbol en tiempo real, la aplicación carece de funciones para interactuar y modificar el estilo de los nodos. La dependencia de un programa C++ para construir el árbol desde el servidor podría agregar latencia en el redibujado del árbol.

2.2.2. Editores Tex

TeX es un sistema tipográfico, muy popular para la redacción de documentos en el entorno académico. Hay una gran familia de herramientas que ahora se derivan de TeX,

Figura 2.13: Interfaz de Tree View

variantes o extensiones como los motores pdfTeX, LuaTeX y XeTeX, y macros como Plain TeX, LaTeX, ConTeXt, pdfLaTeX (motor pdfTeX con el formato LaTeX), LuaLaTeX (motor LuaTeX con el formato LaTeX) y XeLaTeX (motor XeTeX con el formato LaTeX) [29].

Los distintos compiladores, como LaTeX, además de que se consideran la mejor opción para componer fórmulas matemáticas complejas, pueden construir gráficos de toda clase, así como de árboles.

Para trabajar con los compiladores, se necesita un editor, el cual toma de entrada un documento en texto plano y lo transforma en un documento "bellamente" tipográfico o de alta calidad. Algunos de editores recomendados son:

Sharelatex : un editor gratuito en línea que funciona directamente desde el navegador web [28].

Overleaf : otro editor gratuito en línea que funciona directamente desde el navegador web [23].

Texmaker : uno de los editores más populares de código abierto multiplataforma. Requiere la previa instalación de TeX, como Tex Live, MiKTeX (solo para Windows) o MacTeX (solo para OS X) [32]. En la Figura 2.14 se observa el aspecto del editor.

Eclipse : si es programador, probablemente se haya cruzado con el IDE de Eclipse, de código abierto y multiplataforma, para el proceso de sus desarrollos. Con la adición del plugin TeXlipse obtiene una potente herramienta para la edición de documentos teX desde Eclipse [31].

Un ejemplo de un árbol dibujado usando Tex puede verse en la Figura 2.15, este árbol es el resultado de la compilación de un documento Tex que utiliza el paquete tikz-qtree [4].

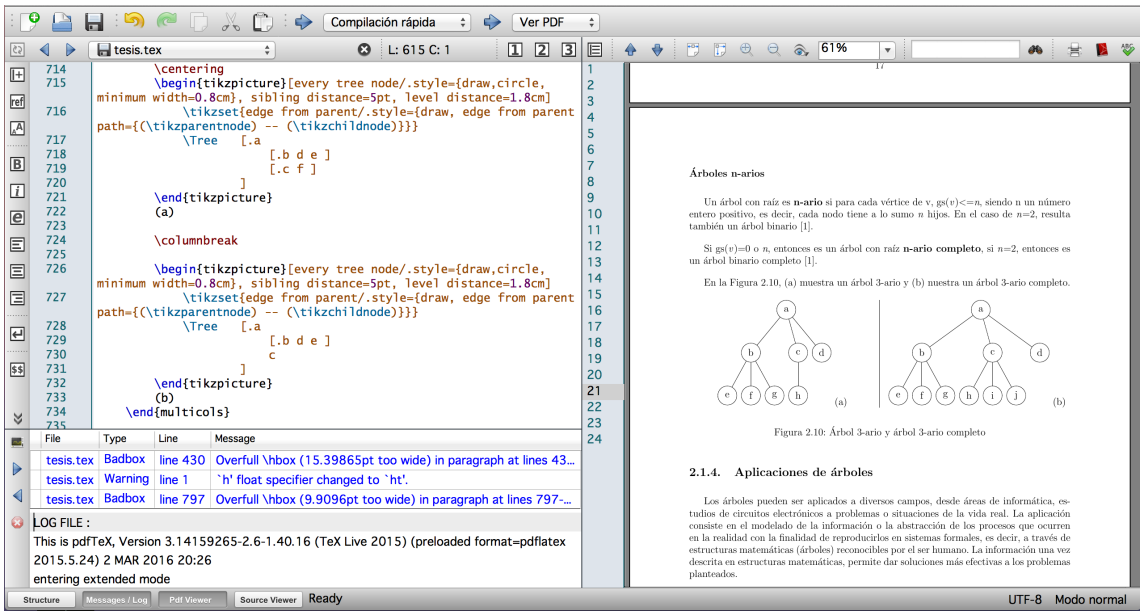


Figura 2.14: Editor Texmaker

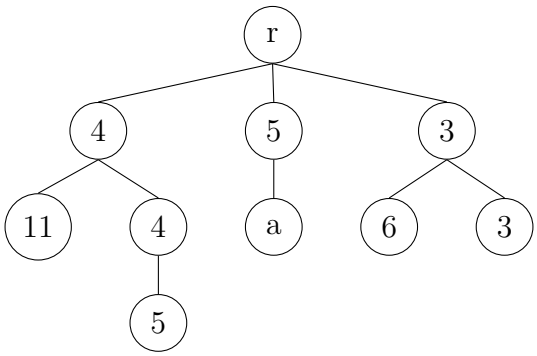


Figura 2.15: Árbol dibujado con el paquete tikz-qtree

2.2.3. Editores de diagramas

Los software de propósito general para la creación de diagramas son también una fuente para dibujar árboles. Algunos de estos son:

Draw.io : un editor de diagramas gratuito en línea desarrollado por la compañía JGraph y liberado en el 2011 [18]. En la Figura 2.16 muestra la interfaz de este editor [6].

Dia Diagram Editor : una aplicación de escritorio para la edición de diagramas, multiplataforma, de código abierto para la creación de diagramas y desarrollado por GNOME con la última versión estable lanzada en el 2011 [7].

Microsoft Visio : una aplicación de escritorio para la edición de diagramas, disponible solo para el sistema operativo Windows y desarrollado por Microsoft con la última versión estable lanzada en el 2016 [14].

SmartDraw : un editor que se presenta como una alternativa a Microsoft Visio, compatible solo con el sistema operativo Windows y desarrollado por SmartDraw con la

versión más estable lanzada en el 2011 [30].

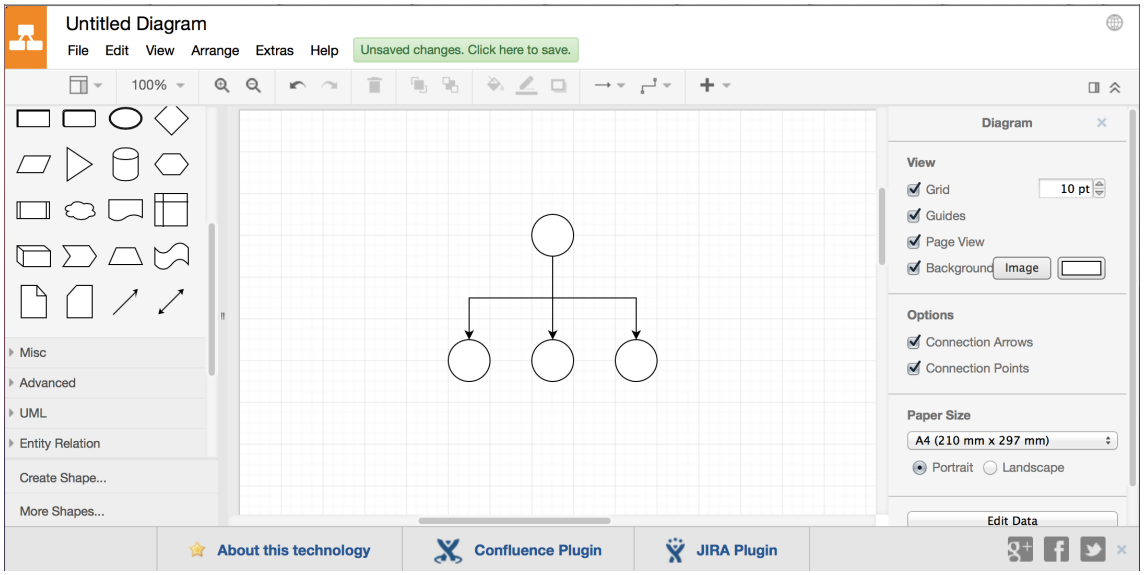


Figura 2.16: Interfaz de Draw.io

Los editores de diagramas mencionados poseen una interfaz muy elaborada e intuitiva, con una amplia gama de herramientas para cubrir en lo posible los requerimientos de los usuarios. Sin embargo, cuando se quiere únicamente dibujar árboles, la extensa lista de herramientas puede llegar a ser un inconveniente.

2.3. Contexto Tecnológico

2.3.1. Arquitectura cliente/servidor

2.3.1.1. Definición

Cliente/servidor es un arquitectura de red en el que cada ordenador o proceso de la red es cliente o servidor. Los servidores están dedicados a proveer recursos o servicios, mientras que los clientes demandan los servicios que provee los servidores [13].

Esta arquitectura establece la relación entre procesos que solicitan servicios (clientes) y los que responden a estos (servidores). Un mismo ordenador puede cumplir las tareas del cliente y del servidor simultáneamente, desde luego, manteniendo la **separación lógica de sus funciones**.

2.3.1.2. Ventajas

- La ventaja importante es que permite modular las funciones, cada función se sitúa en la plataforma más adecuada según su ejecución. El objetivo de separar una gran

pieza de software en módulos es facilitar su proceso de desarrollo, depuración y mantenimiento.

- Los múltiples procesadores en una red permite que se ejecute en partes distribuidas la misma aplicación, logrando la concurrencia de procesos.
- Existe la posibilidad de migrar aplicaciones de un procesador a otro con modificaciones mínimas en los programas.
- Las aplicaciones son escalables. Permite una escalabilidad horizontal o vertical. **Escalabilidad horizontal** al permitir la capacidad de añadir o suprimir estaciones de trabajo que hacen uso de la aplicación (clientes), sin afectar sustancialmente el rendimiento del sistema global. **Escalabilidad vertical** al permitir mejorar las capacidades del servidor, añadir múltiples o migrar a más potentes o de distinta arquitectura sin afectar a los clientes.
- Posibilidad de acceder a los datos independientemente de la ubicación del usuario.
- Ambiente heterogéneo. Las especificaciones de hardware y de sistema operativo entre cliente y servidor no transparentes, se conectan independientemente de sus plataformas.

2.3.1.3. Desventajas

- Cuando una gran cantidad de clientes realizan peticiones simultáneas al mismo servidor, puede congestionarse el tráfico de red.
- Si el servidor se cae, todas las peticiones clientes quedan desatendidas.
- El software y el hardware de un servidor son generalmente muy determinantes. Para atender cierta cantidad de clientes, sobre todo numerosa, un hardware regular de un ordenador personal puede no ser suficiente para suplir el trabajo, normalmente se necesita hardware de mayor potencia, lo que implica mayor coste.

2.3.1.4. Arquitectura multicapas y multiniveles

La **arquitectura multicapa** es una técnica o estilo de programación con el objetivo de separar las funciones lógicas del desarrollo. Cada capa se le confía una misión, que solamente interactúan con sus capas adyacentes lo que le permite abstraerse de las funcionalidades del resto de las capas. De este modo, en caso de que sobrevenga algún cambio, solo se revisa y modifica la capa afectada [11].

La división de tres capas (Figura 2.17) es la separación tradicional mejor conocida:

Lógica de presentación : se encarga de la entrada y salida entre el usuario y la aplicación. Sus principales tareas son: obtener información del usuario, enviarla a la lógica de negocio, recibirla después de su procesamiento y prepararla para retornarla al usuario. Esta capa se comunica únicamente con la lógica de negocio [13].

Lógica de negocios : se encarga del procesamiento de los datos. Sus principales tareas son: comunicarse con la lógica de presentación para recibir las solicitudes y enviar resultados, interactuar con la lógica de datos para ejecutar las reglas de negocio que tiene que cumplir la aplicación, y comunicarse con la lógica de datos para solicitar al gestor de base de datos almacenar o recuperar datos de él [13].

Lógica de datos : se encarga de acceder y gestionar los datos. Sus principales tareas son: comunicarse con la lógica de negocios para recibir solicitudes de almacenamiento o recuperación de datos, también de mantener y asegurar la integridad de los datos [13].

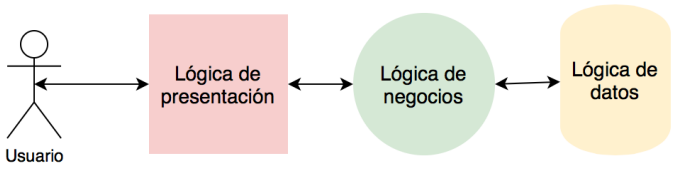


Figura 2.17: Arquitectura de tres capas

La **arquitectura por multinivel** se refiere a la división de las capas de software en piezas físicas de diferentes hardware. Programas de múltiples capas se pueden construirse en un mismo nivel o distribuirse en varios [11].

Es bastante común **confundir los términos de capas y niveles**. Las capas se ocupan de la división lógica, por el contrario, los niveles se ocupan de la distribución de las capas lógicas de forma física.

La **arquitectura cliente/servidor se refieren a veces como arquitecturas de dos capas y dos niveles**. Dos capas primordialmente para separar la lógica de presentación de la lógica de negocios y dos niveles para cada lado, cliente y servidor. Pero pueden existir más capas y niveles dependiendo de los requerimientos del proyecto[13].

2.3.2. Aplicación web

2.3.2.1. Definición

Se denomina **aplicación web** a aquellas herramientas que los usuarios pueden utilizar mediante un navegador para acceder a un servidor web vía Internet o de una intranet. Una aplicación web es un tipo especial de aplicación cliente/servidor, donde tanto el cliente (el navegador) como el servidor (el servidor web) y el protocolo mediante el que se comunican (por lo general http) están estandarizados [1].

2.3.2.2. El cliente web

El **cliente web** es un programa con el que interacciona el usuario para solicitar un servicio remoto a un servidor web [13].

Los **lenguajes del lado del cliente** son ejecutados por el navegador web con el fin de interactuar con el usuario. Entre los lenguajes se tiene HTML, como el formato en que se suele presentar la información al navegador (que conforma la página web), CSS y código script (por lo general Javascript). Además, puede contener pequeños programas o plug-ins que permite enriquecer la aplicación, por ejemplo, en contenidos multimedia (como Flash de Adobe o Silverlight de Microsoft) [33].

2.3.2.3. El servidor web

El **servidor web** es un programa que realiza las conexiones bidireccionales y/o unidireccionales y síncronas o asíncronas con el cliente. Generalmente se utiliza el protocolo HTTP para estas comunicaciones [13].

Los scripts o **lenguajes de programación del lado del servidor** son ejecutados por servidor. La respuesta del programa suele ser una página HTML estándar que se envía al navegador del cliente. Tradicionalmente este programa o script que es ejecutado por el servidor web se encarga de alguna tarea, como componer una páginas, acceder a la base de datos o conexiones en red. Los lenguajes de lado servidor más ampliamente utilizados para el desarrollo de páginas son PHP, ASP, Java, Ruby, PERL, Python o Javascript [33].

En la Figura 2.18 se muestra un ejemplo de una arquitectura cliente/servidor de dos niveles, que se comunica mediante el protocolo HTTP.

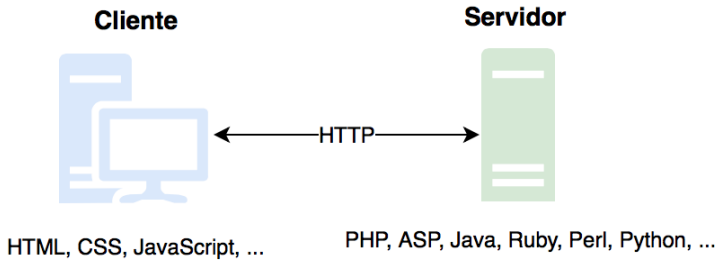


Figura 2.18: Tecnologías empleadas en el cliente y servidor web

2.3.2.4. Arquitecturas de las aplicaciones web

Las aplicaciones web se basan en la arquitectura cliente/servidor: por un lado está el cliente (el agente de usuario) y por otro lado el servidor (el servidor web). Existen diversas variantes de la arquitectura básica según como se implementen las diferentes funcionalidades de la parte servidor.

Algunas de las variantes más comunes son:

1. Un único ordenador aloja el servicio de HTTP, la lógica de negocio, la lógica de datos y los datos. En este caso, el software que ofrece el servicio de HTTP gestiona también la lógica de negocio. [13]. Ver Figura 2.19.

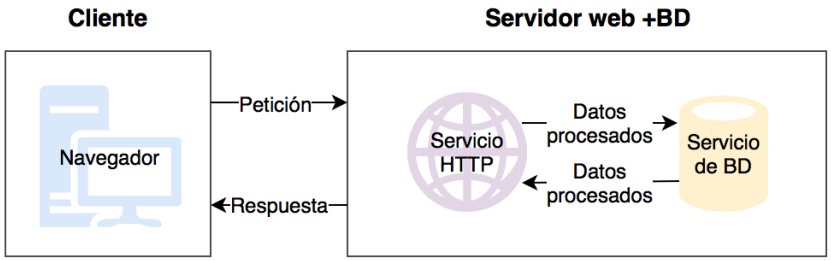


Figura 2.19: Arquitectura de aplicación web: servidor web + BD

2. A partir de la arquitectura anterior se separa la lógica de datos con los datos a un servidor de bases de datos dedicado [13]. Ver Figura 2.20.

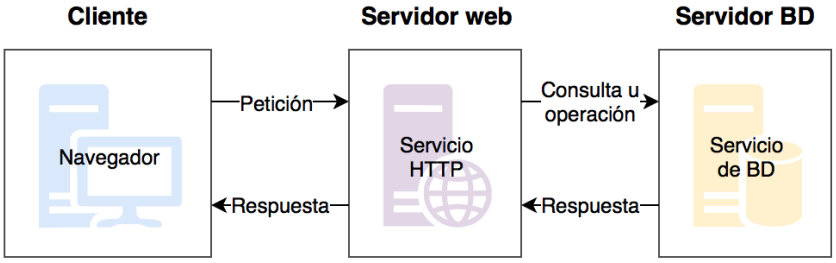


Figura 2.20: Arquitectura de aplicación web: servidor web y servidor BD

3. De la arquitectura número 1 se separa la lógica de negocio del servicio HTTP y se incluye el servicio de aplicaciones para gestionar los procesos que implementan la lógica de negocio [13]. Ver Figura 2.21.

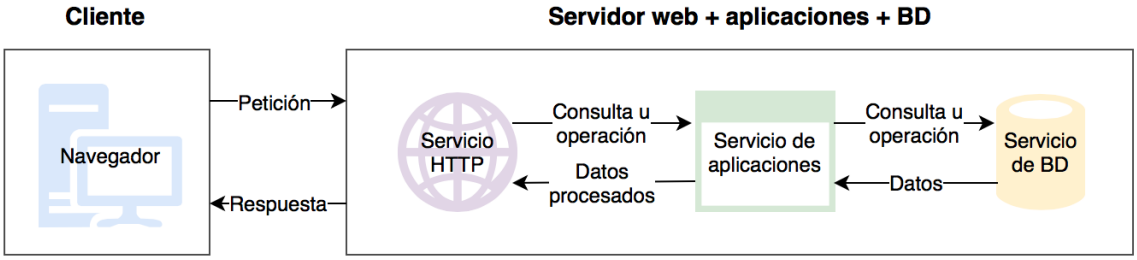


Figura 2.21: Arquitectura de aplicación web: servidor web + aplicaciones + BD

4. De la arquitectura anterior se separa la lógica de datos con los datos a un servidor de base de datos específico [13]. Ver Figura 2.22.

5. Las tres capas de funcionalidades pueden estar separados, cada una en un servidor específico [13]. Ver Figura 2.23.

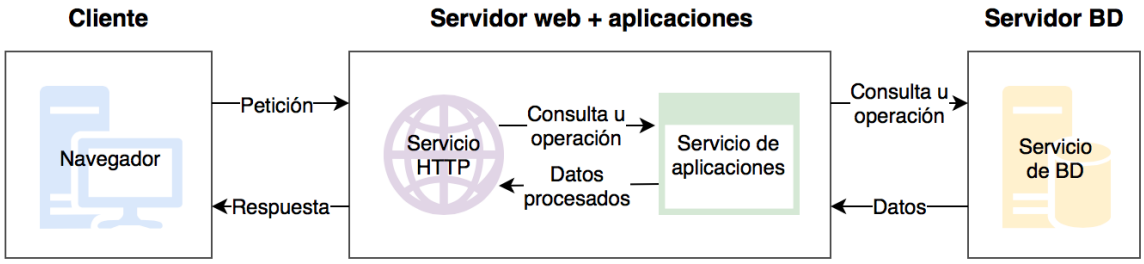


Figura 2.22: Arquitectura de aplicación web: servidor web + aplicaciones y servidor DB

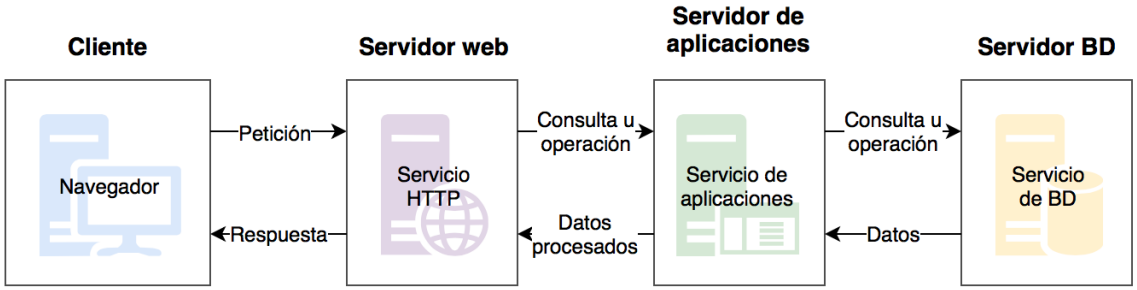


Figura 2.23: Arquitectura de aplicación web: servidor web, servidor de aplicaciones y servidor DB

2.3.3. HTML5

2.3.3.1. Definición HTML

El **lenguaje de marcas de hipertexto** o **HTML** (de las siglas en inglés Hypertext Markup Language) es el lenguaje para describir la estructura de las páginas web. Es un estándar a cargo del World Wide Web Consortium (W3C) o Consorcio WWW [3].

2.3.3.2. Definición HTML5

HTML5 es la más reciente y quinta versión del estándar del lenguaje de marcado HTML de la World Wide Web [35].

Esta versión representa una nueva evolución del estándar que define HTML, con nuevos elementos, atributos y un extenso conjunto de tecnologías que permite a sitios y aplicaciones web soportar un contenido más potente y diverso sin la necesidad de instalar software adicionales como plugins de navegador. La quinta versión ofrece desde una mejora semántica hasta reproducción multimedia y animación de gráficos 2D y 3D [19].

2.3.3.3. Soporte de HTML5 en exploradores web

HTML5 se encuentra aún en proceso de desarrollo [35]. El continuo trabajo de HTML5 suscita probables cambios necesarios en sus especificaciones. Debido a estos cambios, no todas las funcionalidades que provee HTML5 son soportadas por todos los navegadores.

Sin embargo, los navegadores más populares como Chrome, Firefox, Internet Explorer, Safari u Opera, en sus últimas versiones estables hasta la fecha de este documento, tienen un buen soporte y continúan el apoyo hacia más de sus características [19].

La Figura 2.24 muestra el porcentaje de soporte de HTML5 en múltiples navegadores (basado en CanIUse) en distintas versiones. Para obtener mayor información acerca de la compatibilidad del navegador con el estándar HTML5 puede consultar los sitios web CanIUse[1] o Html5Test[2].

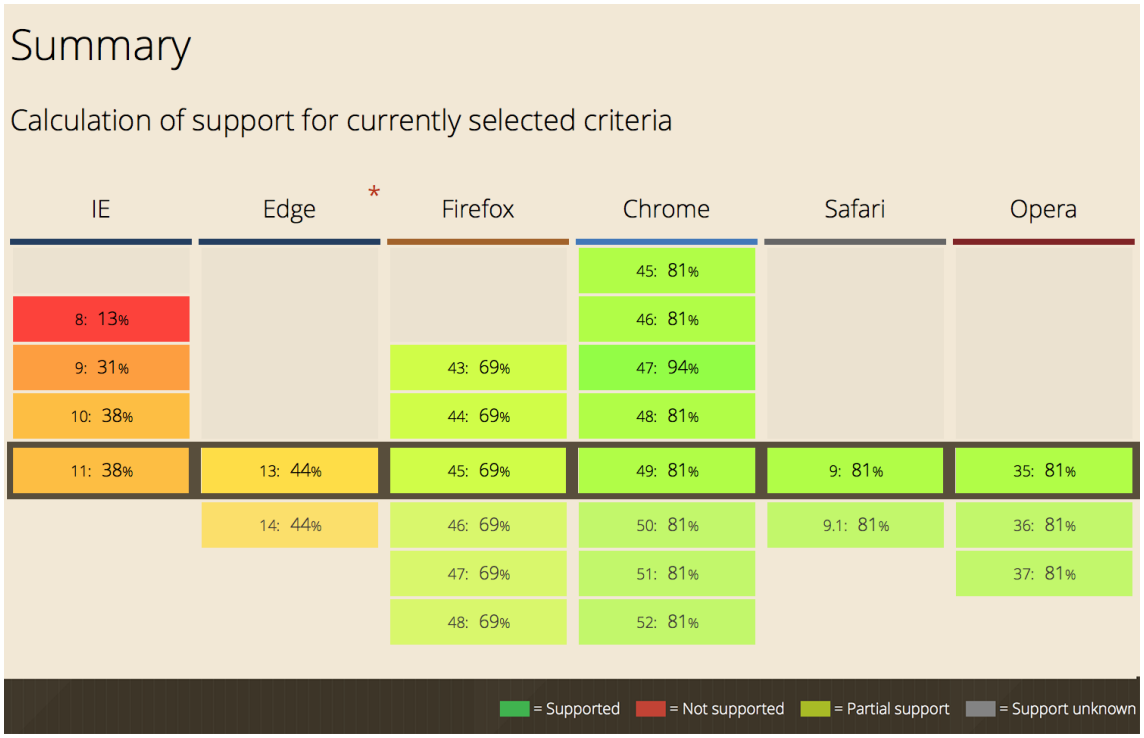


Figura 2.24: Cálculo de soporte entre los navegadores para los criterios de formulario, SVG, video, Server-sent, almacenamiento, canvas y nuevas semántica html5. Revisión 20 de marzo de 2016

A efecto de que cada navegador puede soportar sólo algunas de las características o funcionalidades de HTML5, los desarrolladores web deben ser cautelosos al escribir contenido para esta especificación. Antes de escribir un sitio web basado en HTML5, es preciso asegurarse que cada uno de los navegadores, al menos para los más populares, cuenten con las funcionalidades que se van a utilizar.

2.3.4. El elemento canvas

2.3.4.1. Definición

El elemento **canvas** es un elemento HTML que permite dibujar gráficos mediante scripts (normalmente JavaScript) [36]. La traducción de canvas al español es lienzo, y

[1] http://caniuse.com
[2] https://html5test.com

básicamente es eso, una superficie en la que se puede dibujar como si fuera un lienzo.

canvas es un elemento que se incluye en la quinta versión de HTML. Debido que permite a los navegadores compatibles representar gráficos de dos y tres dimensiones sin la instalación de software de terceros o plug-ins es una de las innovaciones más atractivas para la versión.

El contenido del elemento canvas es contenido embebido, es decir, contenido que importa otro recurso o inserta contenido de otro lenguaje de marcado al documento (como el caso del elemento img, svg, iframe, audio o video) [34]. Para dibujar sobre el canvas se necesita acceder a su contexto de renderización a través del API de un script. El contexto de un lienzo se estudiará más adelante.

El siguiente código HTML muestra un ejemplo de cómo usar las etiquetas del elemento canvas:

```
<canvas id="canvas" width="200" height="500">No soporta canvas
   </canvas>
```

2.3.4.2. El contexto de renderización

Por defecto, el lienzo es completamente transparente. Para dibujar dentro de un canvas se necesita referenciar al contexto del canvas. Es posible confundirse con los conceptos de **elemento y contexto canvas**. El elemento canvas es el nodo DOM del documento HTML, mientras que el contexto es un objeto (**CanvasRenderingContext2D o WebGLRenderingContext**) con propiedades y métodos que permite renderizar y manipular gráficos sobre del elemento canvas [37].

Para hallar el contexto de un elemento canvas, mediante uso de scripts, primero se busca el nodo DOM del elemento, una vez identificado se accede a su contexto.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Canvas from scratch</title>
    <meta charset="utf-8">

    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1/
       jquery.min.js"></script>

    <script>
      $(document).ready(function(){
        var canvas=document.getElementById('canvas');
        var ctx=canvas.getContext('2d');
       });
    </script>
  </head>

  <body>
    <canvas id="canvas" width="200" height="200">
      <span class="rojo">Elemento<b>canvas</b> no soportado.</span>
    </canvas>
  </body>
```

```
22 </html>
```

En la línea 11 del código script, la instrucción `document.getElementById()` retorna el nodo DOM para el elemento canvas (HTMLCanvasElement). En la línea 12, la instrucción `getContext()` accede al contexto del objeto canvas y debe recibir un argumento que especifique el tipo de contexto (CanvasRenderingContext2D o WebGLRenderingContext), para este caso el contexto es `2D` (CanvasRenderingContext2D).

Se puede tener más de un elemento canvas en la misma página y cada lienzo mantiene su propio estado. Si se aplicase el método `getContext()` (con los mismos argumentos) en múltiples ocasiones a un mismo elemento, retornaría la referencia al mismo contexto todas las veces [37].

Los contextos de renderización pueden ser:

CanvasRenderingContext2D : la interfaz CanvasRenderingContext2D proporciona propiedades y métodos para dibujar sobre la superficie de dibujo de un elemento canvas de contexto de renderización de **dos dimensiones**. Para obtener un objeto de esta interfaz, se llama al método `getContext()` de un canvas y se suministra el argumento `2d` como tipo de contexto.

WebGLRenderingContext : la interfaz WebGLRenderingContext proporciona propiedades y métodos para dibujar sobre la superficie de dibujo de un elemento canvas de contexto de renderización de **tres dimensiones**. Para obtener un objeto de esta interfaz, se llama al método `getContext()` de un canvas con el argumento `webgl` como tipo de contexto.

2.3.5. Hojas de Estilo en Cascada

2.3.5.1. Definición

Hojas de Estilo en Cascada o **CSS** (por sus siglas en inglés Cascading Style Sheets), es un lenguaje utilizado para describir el aspecto de los documentos HTML y XML. Este mecanismo permite el control sobre estilo y formato de los documentos separandolo de la presentación [2].

CSS permite el control de estilo y formato de múltiples elementos y páginas web al mismo tiempo. Cualquier cambio en el estilo marcado para un elemento en la CSS afectará a todas las páginas vinculadas a esa CSS en las que aparezca ese elemento [2].

Algunos de los aspectos que se puede controlar con CSS sobre los elementos del documento son: color, tamaño y tipo de letra del texto, separación horizontal/vertical entre elementos o posición dentro de la página.

2.3.6. Javascript

2.3.6.1. Definición

JavaScript (abreviado comúnmente **JS**) es un lenguaje ligero e interpretado, dialecto del estándar ECMAScript, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador [21].

Surgió con el objetivo de programar ciertos comportamientos sobre las páginas web, respondiendo a las interacciones del usuario, realizando acciones automáticas sencillas o animaciones. Sin embargo, las necesidades de las aplicaciones web modernas y el HTML5 ha provocado que el uso de Javascript haya llegado al nivel de complejidad y prestaciones tan alto como otros lenguajes de primer nivel [12, 5].

Javascript ya no solo es un lenguaje que se encuentra en Internet y la Web del lado del cliente, también existe del lado del servidor, en programas de videojuegos y en base de datos. JavaScript ya no es uso exlusivo para el ámbito de páginas web [12, 5].

2.3.6.2. Programación basada en prototipos

JavaScript dispone de fuertes capacidades de programación orientada a objetos, a pesar de que han tenido lugar algunos debates respecto a sus diferencias de su capacidades en comparación con otros lenguajes.

JavaScript es un **lenguaje basado en prototipos**, un estilo de programación orientado a objetos en la que las clases no están presentes[3] (como en C++ o Java). Un prototipo es un objeto abstracto, capaz de contener otros objetos dentro, los cuales pueden ser distintos tipos de datos: variables, vectores, funciones e inclusive otros grupos de objetos [20].

Entonces, en lugar de declarar clases, JavaScript utiliza funciones como clases (define prototipos). Las variables dentro de este serán las propiedades, y las funciones serán los métodos.

[3]Las clases de javascript esta introducidas en **ECMAScript 6**. La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos a JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia [18]. Sin embargo esta tecnología parte del estándar de ECMAScript 2015 (ES6) no es compatible para la fecha con todos los navegadores más usados

2.3.7. MySql

2.3.7.1. Definición

MySql es un sistema de gestión de base de datos de código abierto [4] es desarrollado, distribuido, y soportado por Oracle Corporation [16]. Utiliza el lenguaje de consultas SQL.

Una base de datos es una colección estructurada de datos. Podría ser una colección de imágenes o una gran cantidad información en una red corpotativa. Para acceder, agregar, eliminar o modificar los datos almacenados en una base de datos se necesita de un **sistema de gestión de base de datos** como MySQL Server [16].

2.3.7.2. Base de datos relacional

Las bases de datos de MySql son relacionales. Una **base de datos relacional** almacena datos en tablas separadas en lugar de un solo gran almacén. La base de datos organiza los archivos para optimizar la velocidad de acceso[16].

Ofrece un modelo lógico con objetos tales como bases de datos, tablas, filas y columnas para un entorno de programación flexible. Se configura reglas que rigen las relaciones entre los diferentes campos de datos, como por ejemplo uno-a-uno, uno-a-muchos, único, obligatorio u opcional, y punteros entre las diferentes tablas [16].

2.3.7.3. Ventajas

- Escalable, MySQL Server puede funcionar perfectamente en un escritorio o portátil, junto con sus otras aplicaciones, servidores web, y así sucesivamente. Si se dedica una máquina completa con MySQL, se puede ajustar la configuración para aprovechar toda la memoria, potencia de la CPU, y la capacidad de E/S disponibles. MySQL también puede escalar hasta grupos de máquinas, conectados en red [16].
- Muy rápida, MySQL Server se desarrolló originalmente para manejar grandes bases de datos mucho más rápido que las soluciones existentes. Su conectividad, velocidad y seguridad hacen de MySQL Server **altamente apropiado para acceder bases de datos en Internet** [16].
- Soporte multi-hilos, múltiples clientes tienen acceso concurrente [16].

[4] Cualquiera puede descargar el software MySQL a través de Internet, usarlo y modificarlo sin pagar. El software MySQL usa la GPL (Licencia Pública General de GNU), para definir lo que puede y no puede hacer con el software en diferentes situaciones. Si se necesita añadir código MySQL en una aplicación comercial, se puede comprar una versión con licencia comercial

2.3.8. Ruby on Rails

2.3.8.1. Definición

Ruby on Rails o **RoR** es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby [27]. Fue desarrollado por el danés David Heinemeier y liberado en 2014.

Rails tiene conjunto de librerías, automatismos y convenciones destinados a resolver los problemas más comunes a la hora de desarrollar una aplicación web, para que el programador pueda concentrarse en los aspectos únicos y diferenciales de su proyecto en lugar de los problemas recurrentes [26].

Rails fue creado en 2003 por David Heinemeier Hansson y desde entonces ha sido extendido por el Rails core team, más de 2.100 colaboradores y soportado por una extensa y activa comunidad [26].

La fisolofía de Rails se rige por dos grandes principios:

No te repitas a ti mismo (Dont Repeat Yourself, DRY) : principio de desarrollo de software que estable no repetir información a lo largo del proyecto, para obtener un programa más fácil de mantener y entender, más extensible y menos propenso a errores [27].

Convención sobre configuración : el programador solo necesita definir aquella configuración que no es convencional, dismunuyendo el número de desiciones, simplicidad [27].

2.3.8.2. Patrón de arquitectura MVC

Ruby on Rails sigue el paradigma del modelo Modelo-Vista-Controlador (ver Figura 2.25):

Modelo : es la capa responsable de la lógica de negocios y de la lógica de datos. Encargada de la recuperación de datos convirtiéndolos en conceptos significativos para la aplicación, así como su procesamiento, validación, asociación y cualquier otra tarea relativa a la manipulación de dichos datos [24].

A primera vista los objetos del Modelo puede ser considerados como la primera capa de la interacción con cualquier base de datos usado por la aplicación.

Vista : es la capa responsable de la lógica de presentación (junto con el Controlador). Encargada del uso de la información de la cual dispone para producir cualquier interfaz de presentación de cualquier petición que se presente [24].

Por ejemplo, la capa Modelo devuelve un conjunto de datos, la tarea de la vista es hacer la página HTML (XML u otro formato) que los contenga y presentarla al usuario.

Controlador : es la capa responsable de la lógica de presentación (junto con la Vista). Se encarga de gestionar las peticiones de los usuarios y responder la información solicitada con la ayuda tanto del Modelo como de la Vista [24].

Los controladores pueden ser vistos como los administradores encargados de delegar las tareas a los trabajadores adecuados. Espera las peticiones de los clientes, comprueba su validez y asigna la tarea de procesamiento al Modelo correspondiente, selecciona el tipo de respuesta adecuada, y finalmente delega el proceso de presentación a la capa de la Vista.

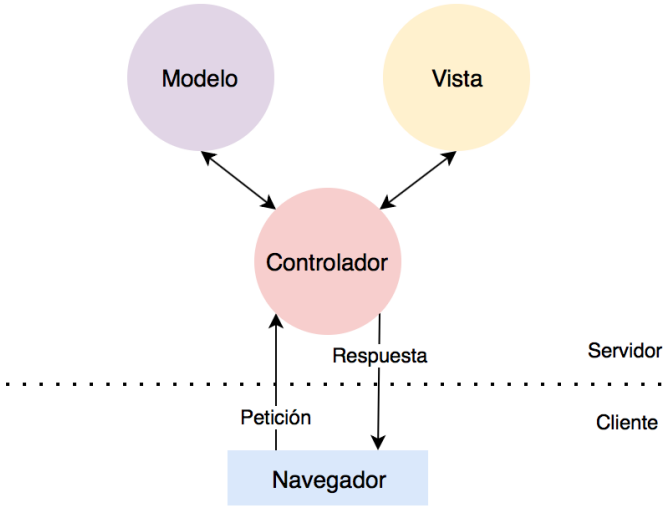


Figura 2.25: Patrón de arquitectura MVC

Capítulo 3

Análisis

En este capítulo se determinan los requisitos funcionales como los no funcionales del sistema. Estos requisitos determinan de forma precisa las propiedades o restricciones que debe satisfacer el sistema. Los requerimientos que se dictan son el resultado de la investigación de las especificaciones de soluciones actuales y de las necesidades sin cubrir.

Una vez identificados los requerimientos se procede a realizar el modelo del sistema a través de la construcción de los diagramas de casos de uso.

Por último se presenta las entidades relevantes del sistema, sus relaciones y propiedades a través del modelo entidad-relación.

3.1. Identificación de los requerimientos

3.1.1. Requerimientos funcionales

3.1.1.1. Basados en la edición del nodo

1. Nombrar el valor del nodo.
2. Escalar el tamaño del nodo.
3. Modificar el estilo del nodo:
   - Establecer la forma: circular y rectangular.
   - Establecer el color de relleno.
   - Establecer el color de borde.
   - Establecer el tipo de línea de borde: recto, punteado o sin borde.
   - Establecer el grosor de borde.
4. Expandir o colapsar la descendencia del nodo.

3.1.1.2. Basados en la edición de la arista

1. Nombrar el valor de la arista.
2. Modificar el estilo de la arista:
   - Establecer el color.
   - Establecer el tipo de línea de borde: recto o punteado.
   - Establecer el grosor.

3.1.1.3. Basados en la edición del árbol

1. Crear árbol n-ario o binario.
2. Crear varios árboles en un mismo espacio de trabajo.
3. Modificar la cantidad de nodos del árbol mediante:
   - Adición de nodo.
   - Sustracción de nodo.
4. Escalar el tamaño del árbol.
5. Trasladar la posición del árbol.
6. Modificar el estilo del árbol:
   - Establecer la longitud entre niveles.
   - Establecer la longitud entre los nodos por nivel.
7. Copiar un àrbol.
8. Cotar un àrbol.
9. Pegar un àrbol sobre otros nodos o sobre ninguno.
10. Seleccionar nodos del árbol.

3.1.1.4. Generales

1. Exportar el árbol a:
   - Formato PNG.
   - Formato LaTeX (pst-tree) (requisito deseable, no obligatorio).
2. Salvaguardar los árboles como un documento.
3. Gestión de los documentos (de los árboles): crear (nuevo documento), leer (abrir documento), actualizar y remover.

4. Opción de ayuda en el editor de árboles que contenga un breve tutorial de uso del editor.
5. Manejo de cuentas de usuario:
    - Registro.
    - Autenticación.
    - Gestión de cuentas: crear, leer, actualizar y remover.
6. Interfaz ajustada a los roles de usuario.

3.1.2. Requerimientos no funcionales

- **Confiabilidad**: el sistema debe comportarse de acuerdo a los que los usuarios esperan de él, ejecutando las acciones deseadas en el tiempo preciso. Sin embargo, cuando la renderización es muy compleja (una cantidad excesiva de nodos) este tiempo podría estar comprometido.
- **Usabilidad**: proporcionar una interfaz gráfica de usuario intuitiva, sencilla, poco propenso a errores y con metáforas establecidas y estandarizadas para un más fácil aprendizaje y uso del mismo.
- **Seguridad**: garantizar que toda la información contenida en el sistema debe estar protegida contra accesos no autorizados, mediante mecanismos de autenticación y sesiones que no permitan la fuga de datos.
- **Consistencia**: no deben existir contradicciones entre las funcionalidades del sistema.
- **Accesibilidad**: es accesible en los navegadores de escritorio: Chrome, Firefox, Opera y Safari, mediante los eventos de usuario accionados por los dispositivos de E/S estándar: ratón y teclado.

3.2. Diagrama de casos de uso

Los actores que se comunican con el sistema son los siguientes:

Administrador : es un tipo de usuario que tiene los siguientes permisos:

- Gestionar lista de usuarios registrados (sin acceso a los documentos asociados a los usuarios).
- Gestionar lista de documentos creados desde su cuenta.
- Acceder al editor de árboles.

Usuario autenticado : es un tipo de usuario que tiene los siguientes permisos:

- Gestionar lista de documentos creados desde su cuenta.

- Acceder al editor de árboles.

Usuario no autenticado :es un tipo de usuario que tiene los siguientes permisos:

- Acceder al editor de árboles sin la posibilidad de guardar ni abrir documentos.

En la Figura 3.1 se ilustra el diagrama de casos de uso en el nivel 0. En este nivel se describe únicamente los actores que intervienen en el sistema.

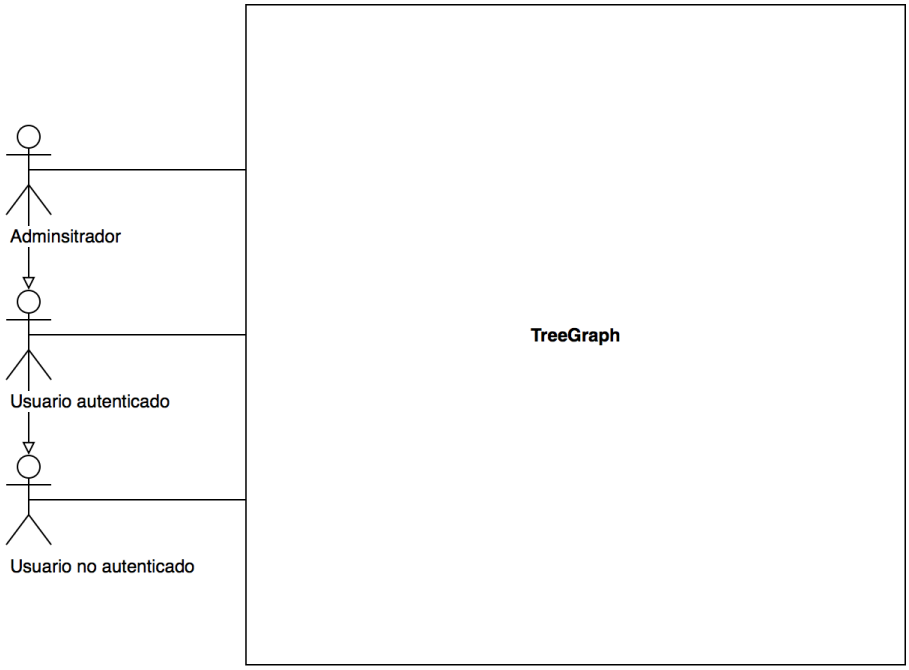


Figura 3.1: Diagrama de casos de uso nivel 0

En la Figura 3.2 se muestra el diagrama de casos de uso en el nivel 1. En este nivel se desglosa las funcionalidades del sistema y se indica la asociación de estos con los actores.

En las figuras 3.3, 3.4, 3.5 y 3.6 se muestra el diagrama de casos de uso en los niveles 2, 3, 4 y 5 respectivamente. En cada nivel se muestra a mayor detalle las relaciones de extensión, inclusión y generalización entre los casos de uso.

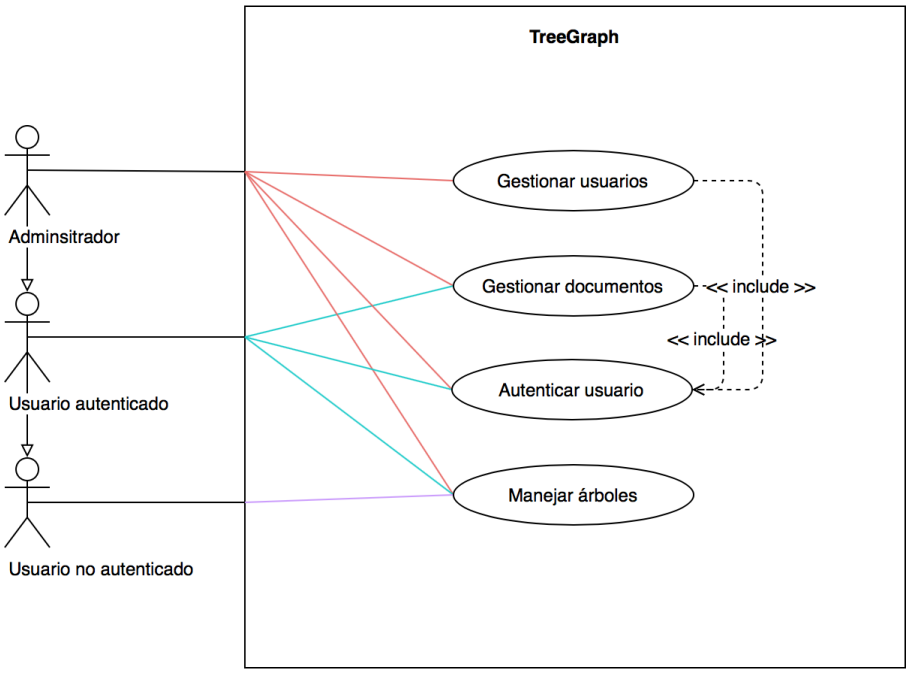


Figura 3.2: Diagrama de casos de uso nivel 1

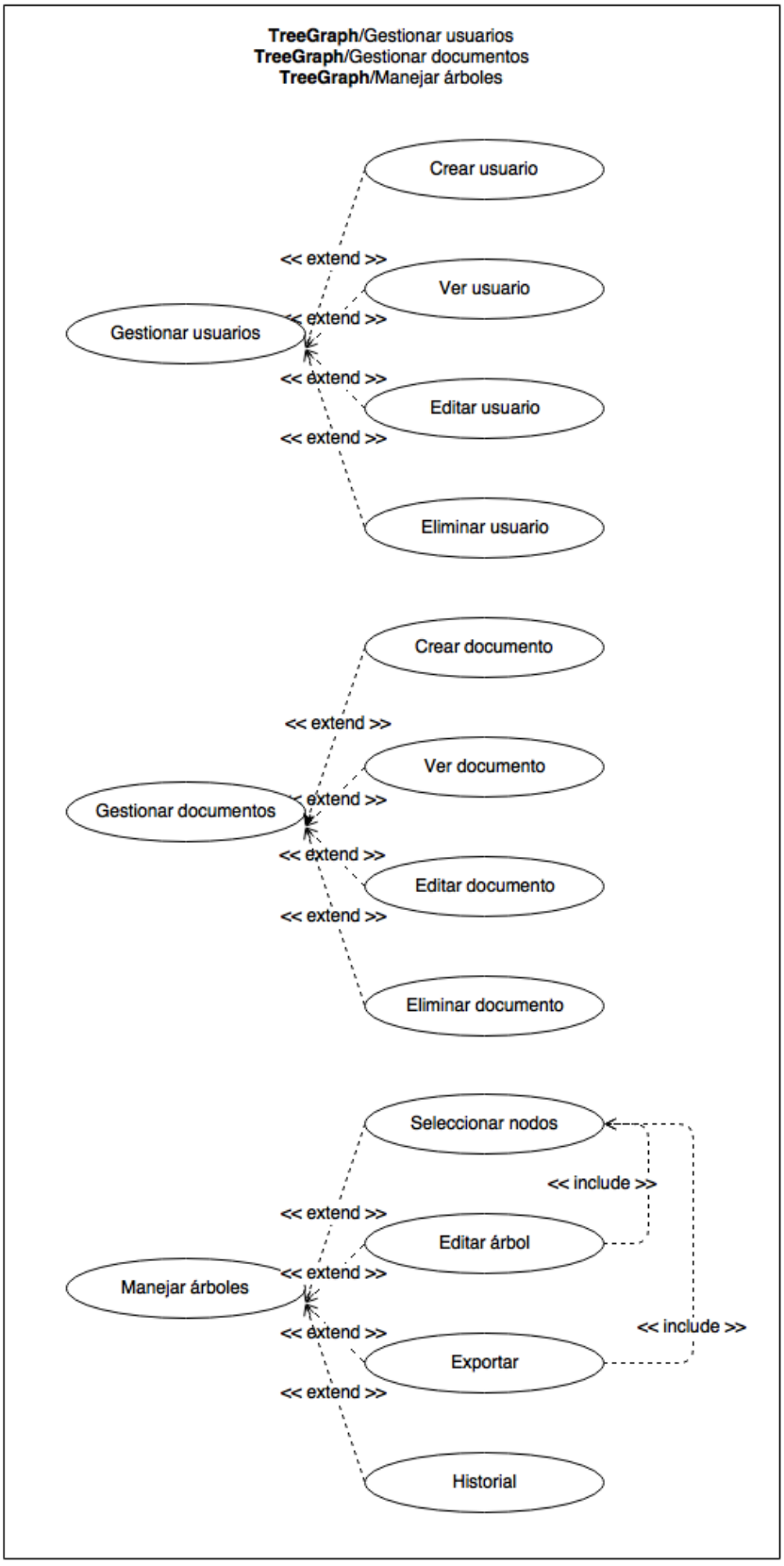


Figura 3.3: Diagrama de casos de uso nivel 2

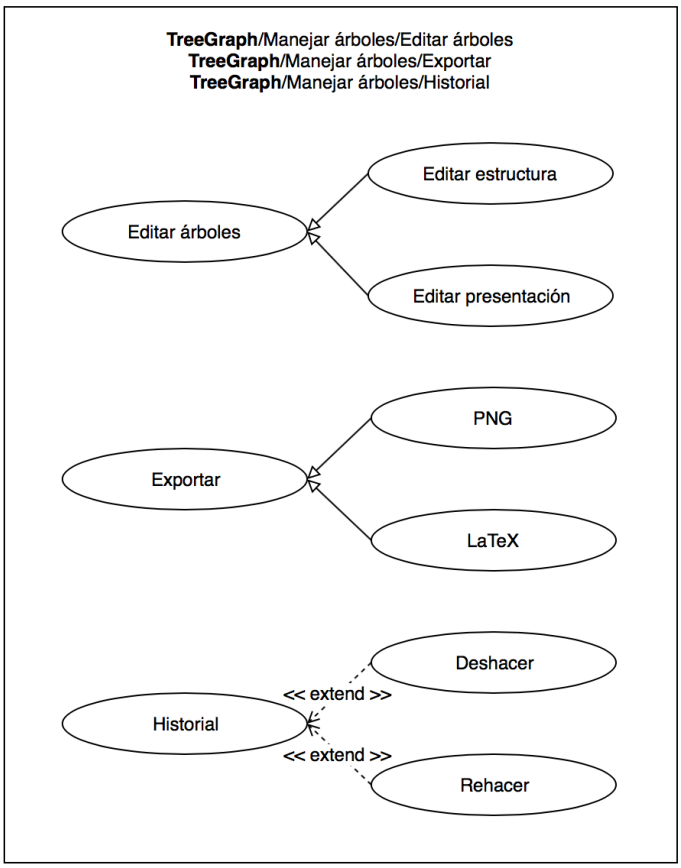


Figura 3.4: Diagrama de casos de uso nivel 3

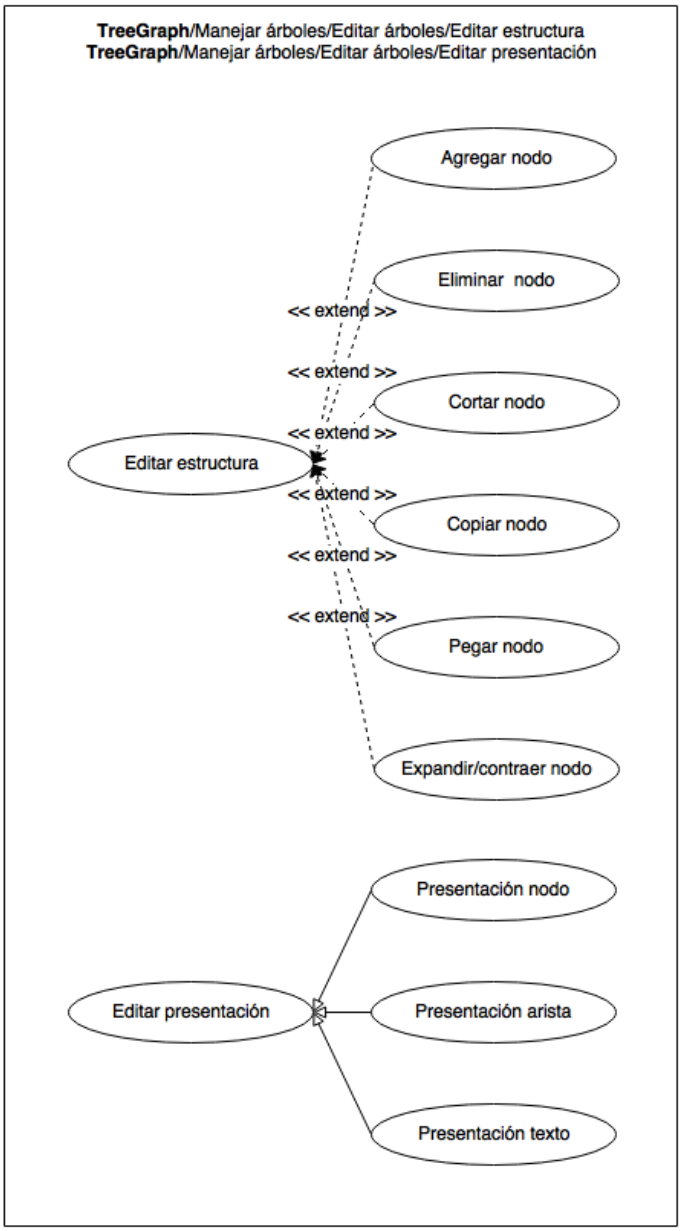


Figura 3.5: Diagrama de casos de uso nivel 4

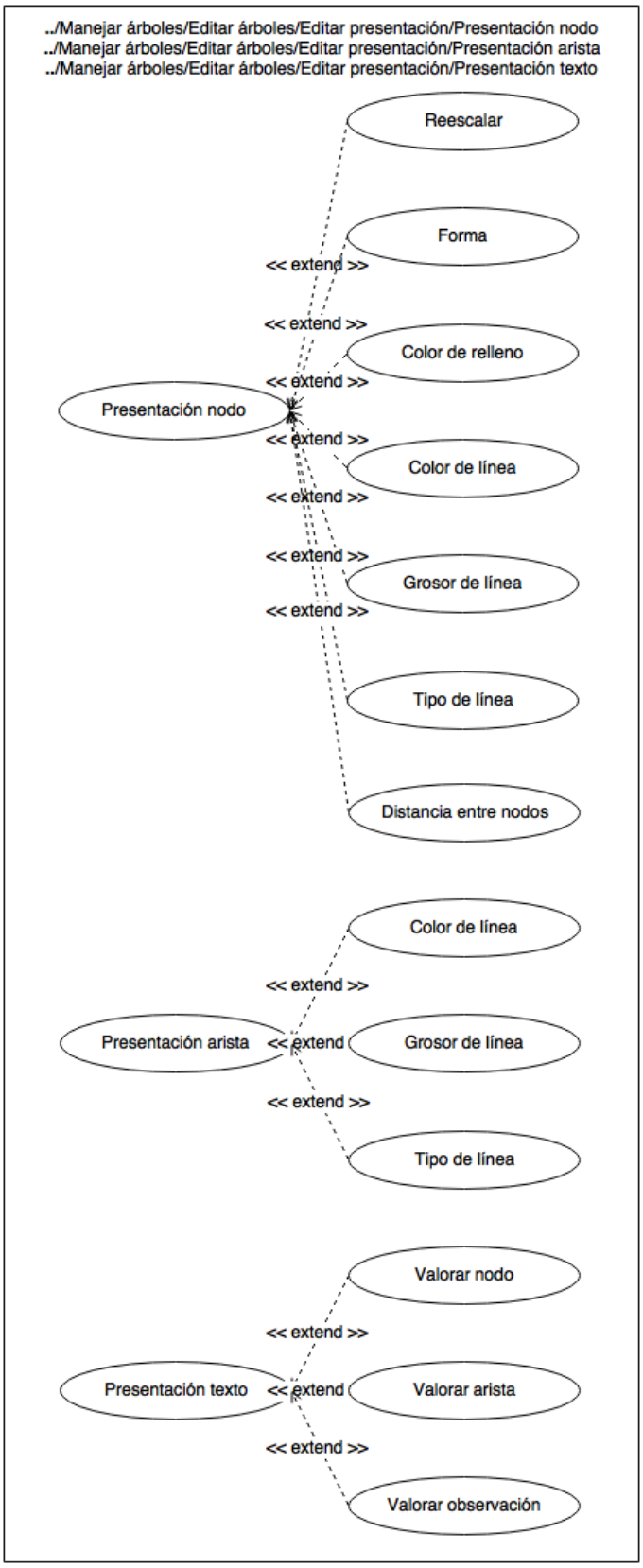


Figura 3.6: Diagrama de casos de uso nivel 5

3.3. Modelo de entidad-relación

En el modelo entidad-relación de la Figura 3.7 se especifica la estructura de base de datos que debe implementar sistema. Cada entidad corresponde a una tabla de la base de datos:

users entidad que almacena las propiedades de los usuarios relevantes para identificar sus cuentas.

documents entidad utilizada para almacenar los documentos de trabajo creados por los usuarios registrados. Estos documentos corresponden a los árboles.

Un usuario puede o no tener documentos creados, entonces un registro de la entidad *users* puede tener de ninguno a varios registros de la entidad *documents*, pero un registro *documetns* solo puede pertenecer a un autor o registro *users*.

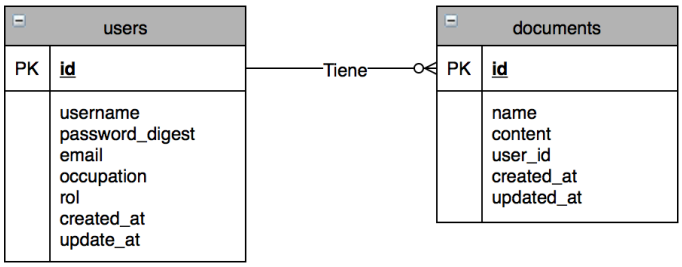


Figura 3.7: Modelo entidad relación

Capítulo 4

Diseño e Implementación

Este capítulo trata sobre el proceso de traducir los requisitos del capítulo anterior en una representación de software, que se asemeja al código fuente. Es decir, implementa todos los requisitos explícitos en la etapa de análisis.

Se muestra en detalle la arquitectura del sistema con la finalidad de ser una guía para que los desarrolladores puedan entender, probar y mantener el software. La arquitectura es la estructura jerarquizada de los módulos del programa, la interacción entre ellos y las interfaces usadas por estos módulos.

Para proporcionar de manera sencilla una idea completa del software, el capítulo separa el diseño en dos módulos, identificando a groso modo las dos funcionalidades principales. Estas corresponden a las secciones de "Módulo de editor gráfico de árboles" y "Módulo de gestión de cuentas de usuario y documentos".

Por último, se anexa las interfaces gráficas de usuario.

4.1. Arquitectura de la solución

La solución se diseña como una aplicación web, que consecuentemente es un tipo especial de aplicación de la **arquitectura cliente/servidor**.

Para apoyar el desarrollo, aliviando el exceso de carga asociado con actividades comunes usadas en desarrollos web, se utiliza un framework para aplicaciones web, en este caso Ruby on Rails versión 4.2. El framework se basa en el patrón de **arquitectura MVC**.

La solución basada en MVC, está formada por tres módulos principales: Modelo, Vista y Controlador. El objetivo de dividir la solución en módulos es hacerla más legible, manejable y fácil de mantener. Cada uno de los módulos del sistema representa una capa lógica con tareas específicas. Dichos módulos se pueden comunicar entre ellos a través de interfaces bien definidas, tales como clases, métodos y atributos.

La arquitectura general del sistema se define como arquitectura MVC. Sin embargo

para estudiar con mayor detalle la solución, es conveniente separar el diseño de manejo de editor de árboles del diseño de manejo de almacenamiento de documentos y cuentas de usuarios.

4.2. Módulo de editor gráfico de árboles

Este módulo se encarga de proveer las operaciones que requiere el editor de árboles para su correcto funcionamiento. En esta sección se estudia la arquitectura en la que se basa el módulo para contener dichas operaciones, que a su vez están encapsuladas en interfaces que coleccionan propiedades y métodos para cumplir el propósito.

A continuación, se describe la arquitectura del sistema de edición de árboles y las interfaces con sus propiedades y métodos de mayor relevancia.

4.2.1. Arquitectura

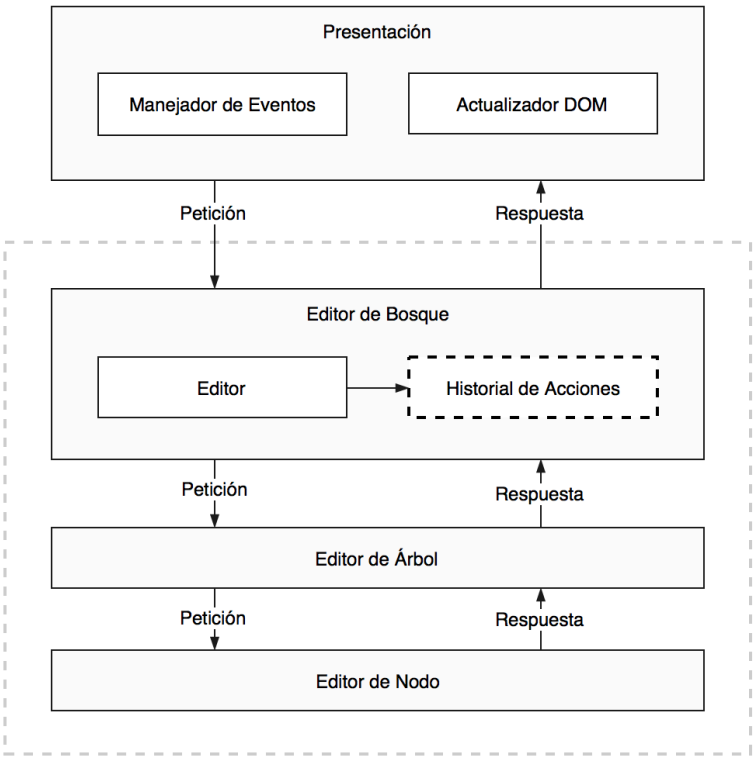


Figura 4.1: Arquitectura del módulo de editor de árboles

Presentación es la capa responsable de definir la lógica de presentación e interfaz gráfica de usuario del editor para la comunicación entre usuario y sistema. Posee dos submódulos:

Manejador de eventos se encarga de la entrada entre el usuario y la aplicación. Obtiene información del usuario mediante manipulación de detección de eventos. Esta información en enviada a la capa de Editor de Bosque.

Actualizador DOM se encarga de la salida entre la aplicación y el usuario. La respuesta recibida del Editor de Bosque es utilizada para actualizar la interfaz de usuario.

Interfaces que utiliza: Interfaz UpdateDom.

Editor de bosque es la capa responsable de procesar la edición a nivel de bosque. Compuesto por dos submódulos:

Editor es el núcleo de la capa, se encarga propiamente de la edición del bosque. Mantiene una lista de árboles a los que modifica haciendo la solicitud al Editor de Árbol.

Interfaces que utiliza: Interfaz Forest, Interfaz SubTree.

Historial de Acciones su labor consiste en almacenar el estado de los árboles por cada cambio a petición del usuario. El historial permite deshacer y rehacer las acciones ejecutadas sobre los árboles. Este módulo encapsula las operaciones de "grabado de acciones" de manera que funcione como una extensión de la capa que pudiera ser eliminado sin afectar significativamente la funcionalidad de esta y demás capas.

Interfaces que utiliza: Interfaz Record.

Editor de Árbol es la capa responsable de procesar la edición a nivel de árbol. Mantiene una lista de nodos a los que modifica haciendo la solicitud al Editor de Nodo. Además se encarga de renderizar el árbol sobre el canvas de context 2D.

Interfaces que utiliza: Interfaz Tree, Interfaz ExporToLatex.

Editor de Nodo es la capa responsable de procesar la edición a nivel de nodo. Es la última etapa de modificación, pues trabaja con la unidad más pequeña del árbol.

Interfaces que utiliza: Interfaz Node, Interfaz Binary.

4.2.1.1. Contexto tecnológico

Para la implementación de las interfaces (o de las capas Editor de Bosque hacia abajo) se utiliza el lenguaje de programación interpretado **JavaScript** sin la adición de librerías de terceros.

Para la lógica de presentación e interfaz gráfica de usuario (o capa de Presentación) se utiliza **HTML5**, **CSS3**, **JavaScript**.

4.2.1.2. Escalabilidad de la arquitectura

La implementación de la capa de Presentación esta ajustada a los valores de requerimientos de la interfaz gráfica de usuario (como distintos tipos de elementos de formulario), de los tipos de eventos de entrada (como ratón o pantalla táctil), del navegador y del dispositivo (como *smartphones* o computadoras de escritorio) en el que se ejecuta la aplicación. Es una capa susceptible a modificaciones.

Para el resto de las capas inferiores, los valores descritos son transparentes o al menos no dependen significativamente.

Se pudiera desarrollar la aplicación para **diferentes entornos o múltiples pestañas de trabajo**, alterando el código en la capa de Presentación mientras se reutiliza las demás.

4.2.2. Interfaz Node

La interfaz Node proporciona un conjunto de propiedades y métodos para manipular las características propias de un nodo; su estilo y su relación con otros. El tipo de nodo que maneja pertenece a los árboles n-ario.

4.2.2.1. Formato *raw* de Node

El formato `raw` de Node es un formato que permite representar un objeto Node como un archivo. Este formato está basado en una variable JSON que contiene las propiedades necesarias del objeto Node por valor y no por referencia, que permite volver a traducirlo como objeto.

Este formato es usado por algunos métodos del programa para realizar copias de los nodos o para almacenarlos en base de datos. En JavaScript, una copia a un objeto es una copia al apuntador en memoria que lo contiene y no la copia en sí del objeto, para lograr una copia independiente se genera una copia del nodo en formato *raw*.

4.2.2.2. Cálculo de las posiciones de los nodos

Los valores de las posiciones de los nodos y aristas se **precalculan y almacenan** en una estructura de memoria antes de ser desplegadas en el canvas. El programa **recalcula** los valores cada vez que se solicita una modificación en el árbol que altere las posiciones de sus nodos, por ejemplo, cuando se añade o se cambia el tamaño del nodo. No todas las modificaciones requieren un refrescamiento de los valores, por ejemplo, cuando se cambia el color o el tipo de línea de los nodos.

Para el cálculo de las posiciones se toma en cuenta la distancia mínima de **separación entre niveles** del árbol, la distancia mínima de **separación entre nodos del mismo nivel** y los **límites** de cada nodo. Para conocer los límites se considera el tamaño del nodo y el grosor de línea. En la Figura 4.2 indica el tamaño de nodo con borde.

Cálculo de las posiciones en la coordenada x La solución se implementa de manera recursiva **recorriendo los nodos en sentido postorden**, se visita primero los hijos de izquierda a derecha para luego llegar a la raíz de cada subárbol. En la 4.3 se muestra un ejemplo del conteo de orden de visita de un árbol.

Para calcular la posición x de un nodo es necesario saber la posición de sus hijos. Una

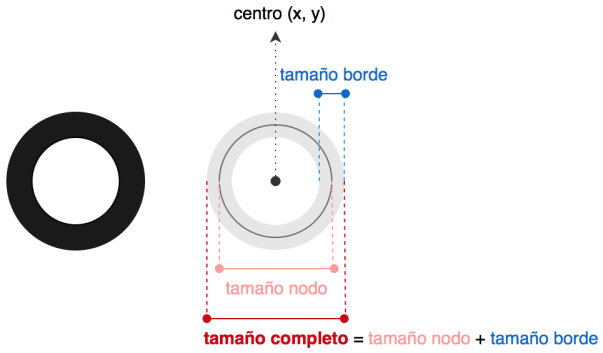


Figura 4.2: Tamaño del nodo

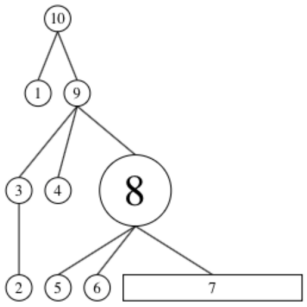


Figura 4.3: Ejemplo del conteo del recorrido postorden

vez ubicados los hijos se centra el nodo respecto a ellos. La distancia tomada para centrar puede ser medida:

1. Desde el límite izquierdo del primer hijo hasta el límite derecho del último, como se muestra en el árbol izquierdo de la Figura 4.4.
2. Desde el centro del nodo del primer hijo hasta el centro del último, como se muestra en el árbol derecho de la Figura 4.4. En este caso el grado de inclinación de las arista de entrada del primer y último hijo son siempre simétricas, razón por la cual es la **alternativa implementada en la solución**.

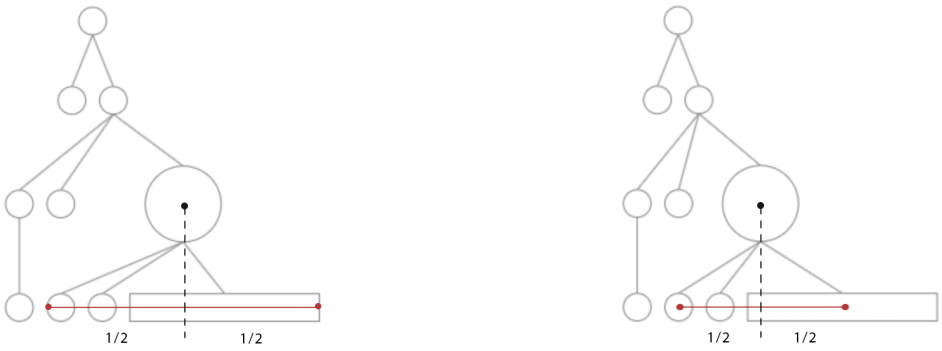


Figura 4.4: Distribución horizontal

Ahora bien, para posicionar a cada hijo se toma en cuenta que estos **no deben solaparse y cumplir una distancia mínima de separación respecto a su nodo de la**

izquierda, en caso de solaparse deben ser trasladados junto con sus descendientes hacia la derecha.

En casos de **árboles binarios** hay que hacer distinción entre el nodo izquierdo y derecho. Si el nodo es hijo único, se debe posicionar éste como si su hermano existiera (y con el mismo tamaño) para ocupar el espacio que obliga al nodo único moverse a la izquierda o derecha dependiendo del caso. En la Figura 4.5 se ejemplifica un caso de posicionamiento de árbol binario.

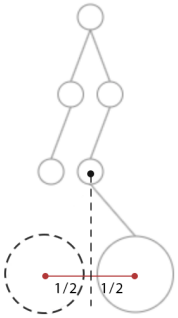


Figura 4.5: Distribución horizontal de árbol binario

Cálculo de las posiciones en la coordenada *y* La solución se implementa de manera iterativa, **recorriendo los nodos del árbol por nivel**, primero se visita la raíz y luego los nodos del siguiente nivel y así sucesivamente.

Cada nivel tiene una coordenada de y constante que se calcula tomando en cuenta la altura máxima del nodo en el nivel anterior y la del nivel actual para que no lleguen a solaparse. En la Figura 4.6 se puede apreciar la distribución por nivel.

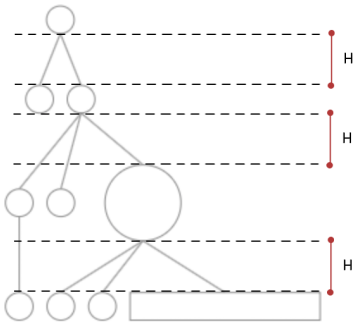


Figura 4.6: Distribución entre los nodos por nivel

4.2.2.3. Propiedades

tree Referencia hacia atrás, apunta al árbol que lo contiene (a la instancia Tree).

Sintaxis

```
1  tree;
```

depth Nivel de profundidad del nodo.

Sintaxis

```
Node.depth;
```

parent Apuntador al nodo padre.

Sintaxis

```
Node.parent;
```

siblingLeft Apuntador al nodo hermano izquierdo más cercano, de no tener conserva su valor predeterminado `null`.

Sintaxis

```
Node.siblingLeft;
```

siblingRight Apuntador al nodo hermano izquierdo más cercano, de no tener conserva su valor predeterminado `null`.

Sintaxis

```
Node.siblingRight;
```

limitLeft Apuntador al nodo izquierdo más cercano (en el mismo nivel), que podría ser a su vez hermano o un nodo de diferente padre, de no tener conserva su valor predeterminado `null`.

Sintaxis

```
Node.limitLeft;
```

limitRight Apuntador al nodo derecho más cercano (en el mismo nivel) que podría ser a su vez hermano o un nodo de diferente padre, de no tener conserva su valor predeterminado `null`.

Sintaxis

```
Node.limitRight;
```

children Arreglo de apuntadores a los nodos hijos, de no tener conserva su valor predeterminado vacío (arreglo de longitud cero).

Sintaxis

```
Node.children;
```

expand booleano que indica si el nodo expande o no sus hijos, su valor por defecto es `true`. Los descendientes de un nodo contraído son ignorados para la renderización. Para indicar que el nodo está contraído se dibuja un símbolo o viñeta debajo del nodo.

Sintaxis

```
Node.expand;
```

render booleano que indica si el nodo se renderiza o no, su valor por defecto es `true`. Los nodos deben renderizarse cuando no poseen un ascendiente contraído, es decir un node ascendiente con valor `expand` igual a `true`.

Sintaxis

```
Node.render;
```

text JSON que indicar los valores del nodo y su arista (y un valor de observación).

Sintaxis

```
Node.text;
```

Implementación

```
this.text = {
  nameNode: undefined, //valor del nodo
  nameEdge: undefined,  //valor de arista
  opcional: undefined, //valor de comentario
};
```

bB JSON que contiene los valores del cuadro delimitador o *boundingbox* del nodo y su arista.

Sintaxis

```
Node.bB;
```

Implementación

```
this.bB = {
  bNode: { //valores para el nodo
    x: undefined, y: undefined, //posicion central

    width: 20, height: 20, //tamano sin borde
    widthHalf: undefined, heightHalf: undefined,

    widthOut: undefined, heightOut: undefined, //tamano exterior
        = width + grosor de borde/2
    widthOutHalf: undefined, heightOutHalf: undefined,

    bText: { // valores para el valor del nodo
      width: undefined, height: undefined // tamano del espacio
          dentro del nodo
    }
  },

  bExpand: { //valores de la vineta expansor
    width: undefined, height: undefined, //tamano
    widthHalf: undefined,

    marginTop: undefined, //separacion superior con el nodo
  },
};
```

style JSON que contiene los valores del estilo del nodo y su arista.

Sintaxis

```
Node.style;
```

Implementación

```
this.style = {
  sNode: { //estilos de nodo
    shape: 'arc', //forma
    lineWidth: 1, //grosor de linea
    fillStyle: 'rgb(255, 255, 255)', //color de relleno
    strokeStyle: 'rgb(0, 0, 0)', //color de linea
    lineDash: [], //tipo de linea
    sText: { //estilos de texto del nodo
      fillStyle: '#000', //color
      fontSize: undefined, //tamano ex px
      fontSizeMax: undefined, //tamano para una letra
      fontFace: 'serif'
    },
  },
```

```
15
16    sExpand: { //estilos del expansor
17      fillStyle: '#000' //color de relleno
18    },
19
20    sEdge: { //estilos de arista
21      strokeStyle: 'rgb(0, 0, 0)', //color
22      lineWidth: 1, //grosor
23      lineDash: [], //tipo
24      sText: { //estilos del texto de arista
25        fillStyle: '#000', //color
26        fontFace: 'serif'
27      }
28    },
29  };
```

4.2.2.4. Métodos

addChild Añade un hijo al nodo.

Sintaxis

```
1   Node.addChild(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo de tipo Node.

Implementación

```
1   this.addChild = function(data){
2     //crea el nuevo nodo
3     var node = new Node(data);
4
5     //inicializa las propiedades
6     node.depth = this.depth + 1;
7     node.setTree(this.getTree());
8
9     //si el padre no es visible o no expande sus hijos entonces
          el nuevo nodo tampoco
10    if(!this.render || !this.expand)
11      node.render = false;
12
13    //se actualiza la relacion entre padre e hijo
```

```
14    node.parent = this;
15    this.children.push(node);
16
17    //si el padre tiene mas hijos entonces necesariamente tiene un
          hermano y limite izquierdo
18    var sibling = this.children[this.children.length-2];
19    if(sibling){
20      //se actualizan las relaciones entre hermanos y limites
21      node.siblingLeft = node.limitLeft = sibling;
22      sibling.siblingRight = sibling.limitRight = node;
23
24    //sino solo se identfica su nodo a la izquierda
25    }else{
26      //se actualizan las relaciones entre limites
27      var cousin = node.getCousinLeft();
28      if(cousin){
29        node.limitLeft = cousin;
30        cousin.limitRight = node;
31      }
32    }
33
34    //identifica su nodo a la derecha
35    var cousin = node.getCousinRight();
36    if(cousin){
37      //se actualizan las relaciones entre limites
38      node.limitRight = cousin;
39      cousin.limitLeft = node;
40    }
41
42    return node;
43  };
```

addLeft Añade un hermano izquierdo al nodo.

Sintaxis

```
1   Node.addLeft(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo tipo Node en caso exitoso.

Retorna `null` en caso contrario.

addRight Añade un hermano derecho al nodo.

Sintaxis

```
Node.addRight(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo tipo Node en caso exitoso.

Retorna `null` en caso contrario.

remove Remueve el nodo.

Sintaxis

```
Node.remove();
```

Implementación

```
this.remove = function(data){
  // si no es un nodo raiz
  if(this.parent){

    //reestablece la conexion entre el hermano izquierdo  y
       derecho del nodo a eliminar
    var siblingLeft = this.siblingLeft,
        siblingRight = this.siblingRight;
    if(siblingLeft)
      siblingLeft.siblingRight = siblingRight;
    if(siblingRight)
      siblingRight.siblingLeft = siblingLeft;

    //reestablece la conexion entre los nodos que rodean los
       extremos de cada nivel del subarbol que se elimina
    var limitLeft = this.limitLeft,
        limitRight = this.limitRight;
    while(mientras haya extremos){
      ...
      se actualizan las relaciones entre limites
      ...
    }

    //remueve el apuntador hijo del padre del nodo a eliminar
```

```
    for(var i = 0; i < this.parent.children.length; ++i){
      if(this === this.parent.children[i]){
        delete this.parent.children[i];
        this.parent.children.splice(i, 1);
          break;
        }
      }

  // si es un nodo raiz
  }else{
    // vacia los valores de las propiedades
    tree = undefined;
    data = undefined;
    delete this.depth;
    delete this.parent;
    delete this.siblingLeft;
    ...
    demas propiedades
    ...
  }
};
```

fitText Reescala el tamaño de la fuente del valor del nodo para que éste no exceda los límites. El tamaño se calcula en base de la longitud del valor y del máximo tamaño establecido.

Sintaxis

```
Node.fitText(context);
```

Parámetros

`context`: contexto de renderización.

export Traduce el nodo a formato *raw*.

Sintaxis

```
Node.export();
```

Retorno

Retorna la traducción en formato *raw*.

Implementación

```
this.export = function(){
  //realiza backtracking para obtener las propiedades de los
     nodos descendientes
  function exportDescendent(node){
    var nodeRaws = [];
    for(var i = 0; i < node.children.length; ++i){
      var child = node.children[i];
      nodeRaws.push({
        expand: child.expand,
        text: JSON.parse(JSON.stringify(child.text)),
        style: JSON.parse(JSON.stringify(child.style)),
        bB: JSON.parse(JSON.stringify(child.bB)),
        children: exportDescendent(child)
      });
    }
    return nodeRaws;
  }

  //obtiene las propiedades del nodo
  var nodeRaw = {
    instance: 'Tree',
    expand: this.expand,
    text: JSON.parse(JSON.stringify(this.text)),
    style: JSON.parse(JSON.stringify(this.style)),
    bB: JSON.parse(JSON.stringify(this.bB)),
    children: exportDescendent(this)
  };
  return nodeRaw;
};
```

import Reemplaza el nodo por `nodeRaw`.

Sintaxis

```
  Node.import(nodeRaw);
```

Parámetros

`nodeRaw`: nodo en formato *raw*.

Retorno

Retorna el nuevo nodo concatenado tipo Node en caso exitoso.

Retorna `null` en caso contrario.

Implementación

```
this.import = function(nodeRaw){
    //realiza backtracking para establecer las propiedades de
       los nodos descendientes
        function importDescendent(node, nodeRaw){
            for(var i = 0; i < nodeRaw.children.length; ++i){
                var childPlane = nodeRaw.children[i];
                var child = node.addChild({
                    expand: childPlane.expand,
                    text: JSON.parse(JSON.stringify(
                       childPlane.text)),
                    style: JSON.parse(JSON.stringify(
                       childPlane.style)),
                    bB: JSON.parse(JSON.stringify(childPlane.bB)
                       )
                });
                importDescendent(child, childPlane);
            }
        }

        //establece las propiedades del nodo
        if(nodeRaw.instance === 'Tree'){
            this.expand = nodeRaw.expand;
            this.text = JSON.parse(JSON.stringify(nodeRaw.text))
               ;
            this.style = JSON.parse(JSON.stringify(nodeRaw.style
               ));
            this.bB = JSON.parse(JSON.stringify(nodeRaw.bB));

            while(this.children.length > 0)
                this.children[0].remove();

            importDescendent(this, nodeRaw);
            return this;
        }
        return null;
    };
```

importAddChild Añade el nodo `nodeRaw` como hijo de `node` de la posición `index`.

Sintaxis

```
  Node.importAddChildByIndex(index, nodeRaw);
```

Parámetros

`index`: entero de la posición del hijo.

`nodeRaw`: nodo en formato *raw*.

Retorno

Retorna el nuevo nodo concatenado tipo Node en caso exitoso.

Retorna `null` en caso contrario.

toogleExpand Habilita o deshabilita la expansión de los nodos hijos.

resizeWidth Reescala el tamaño de la anchura del nodo.

Sintaxis

```
Node.resizeWidth(width);
```

Parámetros

`width`: número de anchura en px.

Implementación

```
this.resizeWidth = function(width){
  //establece la anchura sin borde
  this.bB.bNode.width = width;

  //precalcula la mitad de la anchura
  this.bB.bNode.widthHalf =  width / 2;

  //establece la anchura exterior
  this.bB.bNode.widthOut = width + this.style.sNode.lineWidth;

  //precalcula la mitad de la anchura extrior
  this.bB.bNode.widthOutHalf = this.bB.bNode.widthOut / 2;

  //calcula mediante proporcion el nuevo tamano para el texto
  this.bB.bNode.bText.width = (Node.source.sNode.sText.width * (
     width - this.style.sNode.lineWidth)) / (
     Node.source.sNode.width - Node.source.sNode.lineWidth);
   };
```

resizeHeight Reescala el tamaño de la altura del nodo.

Sintaxis

```
Node.resizeHeight(height);
```

Parámetros

`height`: número de altura en px.

resizeFontSizeMax Reescala el tamaño máximo de la fuente del valor del nodo para que éste no exceda los límites. El tamaño está calculado en base a un valor de longitud 1 .

Sintaxis

```
Node.resizeFontSizeMax();
```

resizeExpand Reescala el tamaño de la viñeta dibujada cuando el nodo está contraído, es decir, cuando el valor de `expand` es igual a `true`.

Sintaxis

```
Node.resizeExpand();
```

resizeLineWidthNode Reescala el tamaño de línea del nodo.

Sintaxis

```
Node.resizeLineWidthNode(lineWidth);
```

Parámetros

`lineWidth`: número del grosor en px.

resizeLineWidthEdge Reescala el tamaño de línea de la arista.

Sintaxis

```
Node.resizeLineWidthEdge(lineWidth);
```

Parámetros

`lineWidth`: número del grosor en px.

4.2.3. Interfaz Binary

La interfaz Binary proporciona un conjunto de propiedades y métodos para manipular las características propias de un nodo binario; su estilo y su relación con otros.

Un nodo binario es un nodo que pertenece a un árbol binario, como un árbol binario es a su vez n-ario (de grado 2), la interfaz Binary hereda las propiedades y los métodos de la Interfaz Node (ciertos métodos son sobreescritos para adaptarse a las necesidades de los nodos binarios).

4.2.3.1. Propiedades

side Indica si en un hijo izquierdo o derecho, `left` y `right` respectivamente, para el caso de la raíz mantiene su valor indefinido.

Sintaxis

```
Node.side;
```

4.2.3.2. Métodos

addChild Añade un hijo al nodo. Por defecto intenta añadir un hijo izquierdo, si ya existe, intenta por el derecho.

Sintaxis

```
Node.addChild(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo de tipo Node.

Implementación

```
this.addChild = function(data){
  //agrega el hijo izquierdo
  var node = this.addLeft(data);
  if(node)
    return node;

  //si el nodo izquierdo ya existe intenta con el lado derecho
  return this.addRight(data);
};
```

addLeft Añade un hijo izquierdo al nodo.

Sintaxis

```
Node.addLeft(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo tipo Node en caso exitoso.

Retorna `null` en caso contrario.

addRight Añade un hijo derecho al nodo.

Sintaxis

```
1  Node.addRight(data);
```

Parámetros

`data`: propiedades para inicializar el nuevo nodo.

Retorno

Retorna el nuevo nodo tipo Node en caso exitoso.

Retorna `null` en caso contrario.

4.2.4. Interfaz Tree

La interfaz Tree proporciona un conjunto de propiedades y métodos que permiten gestionar la estructura, presentación y renderización de uno o más nodos.

A fin de facilitar el estudio de la interfaz Tree se describe el proceso de renderización multicapas que implementa la aplicación. De esta manera, se comprenderá mejor el estudio de ciertas propiedades y métodos que operan en esta interfaz y la siguiente (Forest).

4.2.4.1. Renderización con múltiples elementos canvas

Para optimizar la renderización simultánea de varios árboles en una misma área, se implementa la superposición de múltiples elementos canvas. La ventaja de trabajar con varias capas de elementos canvas independientes es que se consigue un aumento significativo del rendimiento al actualizar solo la capa que se necesita sin modificar el resto.

Las clases Node, Tree y Forest como objetos desde el punto de vista del proceso de renderizado multicapas son identificados de la siguiente manera:

Node se considera la unidad más pequeña del escalón.

Tree representa una capa de elemento canvas compuesta por uno o más instancias Node que deben estar relacionadas entre sí para formar un único árbol. En el canvas se renderiza el árbol.

Forest colecciona todos los elementos canvas o instancias Tree. Los canvas están apilados unos encima de otros, para manejar el orden en profundidad de cada canvas se trabaja con la propiedad de estilo `z-index`[1].

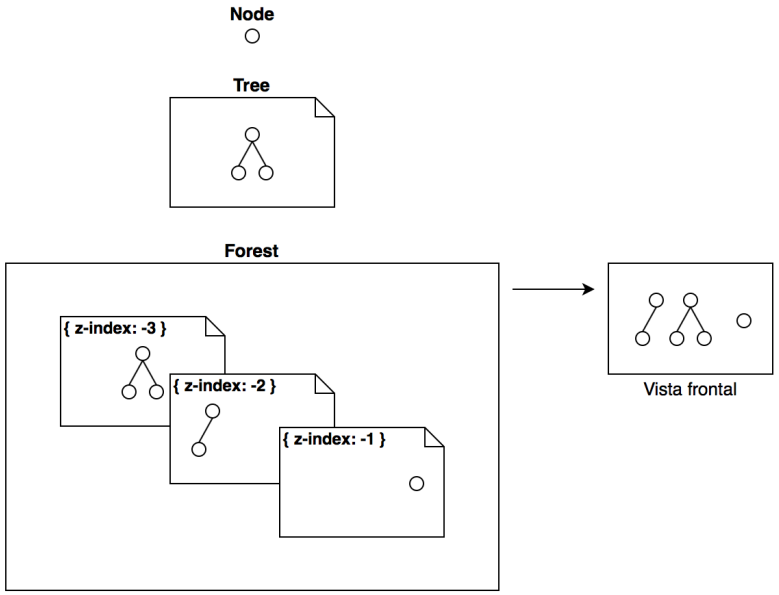


Figura 4.7: Renderización con múltiples capas canvas

El proceso de renderización multicapas sigue los siguientes pasos:

1. El usuario **solicita** modificar uno o más nodos.
2. Se **identifican los nodos por capas**.
3. Una vez identificadas las capas, cada una de manera independiente debe **ejecutar los cambios** solicitados. Los cambios solicitados pueden demandar el refrescamiento del cálculo de las posiciones de los nodos y despliegue sobre el canvas.
4. Dependiendo de las operaciones ejecutadas en el paso anterior, **nuevas capas pueden ser agregadas, eliminadas o reposicionadas en profundidad** en Forest.

[1]La propiedad CSS z-index indica el z-order de un elemento y sus descendientes. Cuando varios elementos se superponen, z-order determina cuales cubren a otros. Un elemento con mayor z-index generalmente cubre a otro menor [22].

4.2.4.2. Propiedades

eWorkspace Elemento que representa el espacio de dibujo del editor, éste es el contenedor del `canvas`.

Sintaxis

```
eWorkspace;
```

canvas Elemento canvas donde se renderiza el árbol.

Sintaxis

```
canvas;
```

context Contexto de renderización 2D de `canvas`.

Sintaxis

```
context;
```

rootNode Node para la raíz del árbol.

Sintaxis

```
rootNode;
```

rootNodePos JSON de las coordenadas bidimensionales de la raíz del árbol.

Sintaxis

```
rootNodePos;
```

visibleMatrix Matriz que contiene todos los nodos del árbol, cada fila le corresponde almacenar los nodos del nivel con la posición de la fila, es decir, la fila 0 almacena el nodo del nivel 0, la fila 1 almacena los nodos del nivel 1 y así sucesivamente. La intención de esta matriz consiste en mantener una propiedad que facilite y acorte el tiempo de acceso a todos los nodos, en especial a los nodos de un nivel.

Sintaxis

```
visibleMatrix;
```

bBTree JSON con los valores de *boundingbox* del árbol.

Sintaxis

```
Tree.bBTree;
```

style JSON con el estilo del árbol. Se refiere al estilo del árbol aquellos valores de estilo que afectan a todos los nodos del árbol por igual, por ejemplo: la distancia horizontal y vertical entre ellos.

Sintaxis

```
Tree.style;
```

4.2.4.3. Métodos

resize Reescala el `canvas`.

Sintaxis

```
resize(pos);
```

Add.root Añade el nodo raíz del árbol en la posición `pos`. Utilizado para especificar un árbol n-ario.

Sintaxis

```
Tree.Add.root(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Retorno

Retorna el nuevo nodo de tipo Node.

Add.rootBinary Añade el nodo raíz del árbol en la posición `pos`. Utiliza para especificar un árbol binario.

Sintaxis

```
Tree.Add.rootBinary(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: }x: 10, y: 15}.

Retorno

Retorna el nuevo nodo de tipo Node.

Add.children Añade hijos a los nodos `nodes` del árbol.

Sintaxis

```
Tree.Add.children(nodes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

Retorno

Retorna un arreglo que recolecta las variables de retorno de cada operación aplicada a `nodes` en el mismo orden. El arreglo puede contener resultados los siguientes valores:

Si la operación es exitosa el resultado es el nuevo nodo de tipo Node.

De lo contrario el resultado es `null`.

Add.lefts Añade un hermano izquierdo a los nodos `nodes` en árboles n-arios o un hijo izquierdo en árboles binario.

Sintaxis

```
Tree.Add.lefts(nodes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

Retorno

Retorna un arreglo que recolecta las variables de retorno de cada operación aplicada a `nodes` en el mismo orden. El arreglo puede contener resultados los siguientes valores:

Si la operación es exitosa el resultado es el nuevo nodo de tipo Node.

De lo contrario el resultado es `null`.

Add.rights Añade un hermano derecho a los nodos `nodes` en árboles n-arios o un hijo derecho en árboles binario.

Sintaxis

```
1  Tree.Add.rights(nodes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

Retorno

Retorna un arreglo que recolecta las variables de retorno de cada operación aplicada a `nodes` en el mismo orden. El arreglo puede contener resultados los siguientes valores:

Si la operación es exitosa el resultado es el nuevo nodo de tipo Node.

De lo contrario el resultado es `null`.

removes Remueve a partir de los nodos `nodes` del árbol.

Sintaxis

```
1  Tree.removes(nodes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

remove Remueve a partir del nodo `node` del árbol.

Sintaxis

```
Tree.remove(node);
```

Parámetros

`node`: nodo de tipo Node.

expand Expande o contrae la renderización de la descendencia a partir del nodo `node` del árbol.

Sintaxis

```
Tree.expand(node);
```

Parámetros

`node`: nodo de tipo Node.

Retorno

Si el nodo tiene hijos para expandir o contraer retorna el nodo Node.

De lo contrario retorna `null`.

cut Corta el árbol a partir del nodo `node`. Cuando se corta, a parte de removerse del árbol, se genera una copia en formato *raw* del subárbol cortado.

Sintaxis

```
Tree.cutPlane(node);
```

Parámetros

`node`: nodo de tipo Node.

Retorno

Retorna el subárbol cortado en formato *raw*.

Export.original Traduce el árbol a partir del nodo `node` a formato *raw*.

Sintaxis

```
Tree.Export.original(node);
```

Parámetros

`node`: nodo de tipo Node.

Retorno

Retorna la traducción en formato *raw*.

Export.png Traduce el árbol desde la raíz a formato PNG.

Sintaxis

```
Tree.Export.png();
```

Retorno

Retorna un texto de la traducción en formato PNG.

Implementación

```
png: function(){
  //copia el estado original
  var widthBackup = context.canvas.width;
  var heightBackup = context.canvas.height;
  var rootNodePosBackup = {x: rootNodePos.x, y: rootNodePos.y};
     //respaldo de posicion de la raiz
  var tempExpanded = []; //respaldo de nodos no expandidos
  var tempRender = []; //respaldo de nodos no renderizados

  //expande los nodos antes de tomarle la fotografia
  for(var i = 0; i<Object.keys(visibleMatrix).length; ++i)
    for(var j = 0; j<visibleMatrix[i].length; ++j){
      var node = visibleMatrix[i][j];
      if(!node.expand){
        node.expand = true;
        tempExpanded.push(node);
      }
      if(!node.render){
        node.render = true;
        tempRender.push(node);
```

```
      }
    }

  //ordena la posicion de los nodos sin contracciones
  CalcPosition.vertical();

  //posiciona el arbol en la esquina superior izquierda
  rootNodePos = {x: rootNode.bB.bNode.x + 5, y:
     rootNode.bB.bNode.y + 5};
  if(self.bBTree.xMin < 0)
    rootNodePos.x -= self.bBTree.xMin;
  if(self.bBTree.yMin < 0)
    rootNodePos.y -= self.bBTree.yMin;
  self.Translate.toRootNodePos();

  //recorta el canvas
  context.canvas.width = self.bBTree.width + 10;
  context.canvas.height = self.bBTree.height + 10;

  // agrega fondo blanco
  context.fillStyle = 'rgb(255, 255, 255)';
  context.fillRect(0, 0, canvas.width, canvas.height);

  //dibuja
  Render.draw();

  //produce la imagen en PNG del canvas
  var img = context.canvas.toDataURL("image/png");

  // elimina fondo blanco
  context.clearRect(0, 0, canvas.width, canvas.height);

  // devuelve al estado original
  for(var i = 0; i<tempExpanded.length; ++i)
    tempExpanded[i].expand = false;
  for(var i = 0; i<tempRender.length; ++i)
    tempRender[i].render = false;

  rootNodePos = {x: rootNodePosBackup.x, y: rootNodePosBackup.y
     };
  context.canvas.width = widthBackup;
  context.canvas.height = heightBackup;
  CalcPosition.vertical();
  self.Translate.toRootNodePos();
  Render.draw();

  return img;
},
```

Export.pstTree Traduce el árbol desde la raíz a formato LaTeX con paquete pst-tree.

Sintaxis

```
Tree.Export.pstTree();
```

Retorno

Retorna un texto de la traducción en formato LaTeX con paquete pst-tree.

Import.root Establece el árbol `treeRaw` como el nuevo árbol de la intancia.

Sintaxis

```
Tree.Import.root(treeRaw);
```

Parámetros

`treeRaw`: árbol en formato *raw*.

Retorno

Retorna el nuevo nodo tipo Node.

Import.addSelfs Reemplaza los nodo de `nodes` por los árboles en `treeRaws`, en una correspondencia de uno a uno.

Sintaxis

```
Tree.Import.addSelfs(nodes, treeRaws);
```

Parámetros

`nodes`: arreglo de nodos de tipo Nodes.

`treeRaws`: arreglo de árboles en formato *raw*.

Retorno

Retorna un arreglo que recolecta las variables de retorno de cada operación aplicada a `nodes` en el mismo orden. El arreglo puede contener resultados los siguientes valores:

Si la operación es exitosa el resultado es el nuevo nodo de tipo Node.

De lo contrario el resultado es `null`.

Import.addChild Añade el árbol `treeRaw` como hijo de `node` de la posición `index`.

Sintaxis

```
Tree.Import.addChildByIndex(index, node, treeRaw);
```

Parámetros

`index`: entero que indica la posición del nuevo hijo.

`node`: nodo de tipo Node.

`treeRaw`: árbol en formato *raw*.

Retorno

Retorna el nuevo nodo concatenado tipo Node en caso exitoso.

Retorna `null` en caso contrario.

ResizeNode.widths Reescala la anchura de los nodos `nodes` por `widths`, en una correspondencia uno a uno.

Sintaxis

```
Tree.ResizeNode.widths(nodes, widths);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`widths`: arreglo de números de anchura en px.

ResizeNode.heights Reescala la anchura de los nodos `nodes` por `heights`, en una correspondencia uno a uno.

Sintaxis

```
Tree.ResizeNode.heights(nodes, heights);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`heights`: arreglo de números de altura en px.

ResizeNode.lineWidthNodes Reescala el tamaño de línea de los nodos `nodes` por `lineWidths`, en una correspondencia uno a uno.

Sintaxis

```
Tree.ResizeNode.lineWidthNodes(nodes, lineWidths);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`lineWidths`: arreglo de números de tamaños de línea en px.

ResizeNode.lineWidthEdges Reescala el tamaño de línea de las aristas de los nodos `nodes` por `lineWidths`, en una correspondencia uno a uno.

Sintaxis

```
Tree.ResizeNode.lineWidthEdges(nodes, lineWidths);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`lineWidths`: arreglo de números de tamaños de línea en px.

ResizeTree.resize Reescala el tamaño del árbol.

Sintaxis

```
Tree.ResizeTree.resize(ref, newRef);
```

ResizeMargin.top Reescala la distancia vertical entre los nodos del árbol.

Sintaxis

```
Tree.ResizeMargin.top(margin);
```

Parámetros

`margin`: número de distancia en px.

ResizeMargin.left Reescala la distancia horizontal entre los nodos del árbol.

Sintaxis

```
Tree.ResizeMargin.left(margin);
```

Parámetros

`margin`: número de distancia en px.

Style.shapes Establece la forma de los nodos `nodes` por `shapes`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.shapes(nodes, shapes, widths, heights);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`shapes`: arreglo de string que indica la forma.

`widths`: arreglo de números de anchura en px.

`heights`: arreglo de números de altura en px.

Style.Color.fillStyleNodes Establece el color de relleno de los nodos `nodes` por `fillStyles`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Color.fillStyleNodes(nodes, fillStyles);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`fillStyles`: arreglo de string que indica los colores en RGB.

Style.Color.strokeStyleNodes Establece el color de línea de los nodos `nodes` por `strokeStyles`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Color.strokeStyleNodes(nodes, strokeStyles);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`strokeStyles`: arreglo de string que indica los colores en RGB.

Style.Color.strokeStyleEdges Establece el color de línea de las aristas de los nodos `nodes` por `strokeStyles`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Color.strokeStyleEdges(nodes, strokeStyles);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`strokeStyles`: arreglo de string que indica los colores en RGB.

Style.LineDash.lineDashNodes Establece el tipo de línea de los nodos `nodes` por `lineDashNodes`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.LineDash.lineDashNodes(nodes, lineDashes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`lineDashes`: Un arreglo. Una lista de números que especifica las distancias para dibujar una línea alterna y una brecha. Por ejemplo, [5, 15].

Style.LineDash.lineDashEdges Establece el tipo de línea de las aristas de los nodos `nodes` por `lineDashEdges`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.LineDash.lineDashEdges(nodes, lineDashes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`lineDashes`: Un arreglo. Una lista de números que especifica las distancias para dibujar una línea alterna y una brecha. Por ejemplo, [5, 15].

Style.Text.nameNodes Establece el valor de los nodos `nodes` por `nameNodes`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Text.nameNodes(nodes, nameNodes);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`nameNodes`: arreglo de string.

Style.Text.nameEdges Establece el valor de las aristas de los nodos `nodes` por `nameEdges`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Text.nameEdges(nodes, nameEdges);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`nameEdges`: arreglo de string.

Style.Text.opcionals Establece el valor observación de los nodos `nodes` por `opcionals`, en una correspondencia uno a uno.

Sintaxis

```
Tree.Style.Text.opcionals(nodes, values);
```

Parámetros

`nodes`: arreglo de nodos de tipo Node.

`opcionals`: arreglo de string.

Translate.toRootNodePos Traslada la posición del árbol hasta la posición de la propiedad `rootNodePos`.

Sintaxis

```
Tree.Translate.toRootNodePos();
```

Translate.byPos Traslada la posición del árbol asignándole a la propiedad `rootNodePos` la posición `pos`.

Sintaxis

```
Tree.Translate.byPos(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

CalcPosition.vertical Calcula las posiciones que debe tomar los nodos del árbol para que se desplieguen en sentido vertical y en dirección hacia abajo.

Sintaxis

```
CalcPosition.vertical();
```

Implementación

```
vertical: function(){
  var first = true;

  //mueve hacia la derecha los descendientes de un nodo
  function moveNodesDescendents(node, diff){
    node.bB.bNode.x += diff;
    for(var i = 0; i < node.children.length; ++i)
      moveNodesDescendents(node.children[i], diff);
  }

  //calculo de la coordenada x
  function performLayoutX(node){
    var limitLeftRendered = node.getLimitLeftRendered();

    //recursivamente recorre todos los nodos visibles (los hijos de nodos que no han sido colapsados)
    if(node.expand && node.children.length > 0)
      for (var i = 0; i < node.children.length; ++i)
        performLayout(node.children[i]);

      //centra el nodo respecto a sus hijos
      //identifica si es nodo binario o no
      if(node instanceof Binary && node.children.length === 1){
        var childrenWidth = node.children[0].bB.bNode.width + self.style.marginLeft;
        if(node.children[0].side === "left")
          node.bB.bNode.x = node.children[0].bB.bNode.x + childrenWidth / 2;
      else
        node.bB.bNode.x = node.children[0].bB.bNode.x - childrenWidth / 2;
      }else{
        node.bB.bNode.x = node.children[0].bB.bNode.x + ((node.children[node.children.length-1].bB.bNode.x - node.children[0].bB.bNode.x) / 2);
      }

      //si tiene nodos a la izquierda
      if(limitLeftRendered){

        //calcula la distancia de separacion con el nodo de la izquierda
        var inter = node.bB.bNode.x - node.bB.bNode.widthOutHalf - limitLeftRendered.bB.bNode.x - limitLeftRendered.bB.bNode.widthOutHalf;

        //si es menor
        if(inter < self.style.marginLeft)
          //mueve el nodo y sus descendientes a la derecha para que cumpla con el minimo de separacion
          moveNodesDescendents(node, self.style.marginLeft - inter);
      }

    //si el nodo no tiene hijos pero si limite izquierdo
    }else if(limitLeftRendered){

      //se posiciona al lado derecho de su limite izquierdo
```

```
48        node.bB.bNode.x = limitLeftRendered.bB.bNode.x +
              limitLeftRendered.bB.bNode.widthOutHalf +
              self.style.marginLeft + node.bB.bNode.widthOutHalf;
49
50      //si no posee hijos ni limite izquierdo
51      }else{
52
53        //si es el primer nodo del recorrido entonces x=0
54        if(first){
55          node.bB.bNode.x = 0;
56          first = false;
57
58        //si no x se posiciona muy a la izquierda para que solape
              algun nodo del nivel anterior y luego pueda ser
              removido justo lo necesario cuando se atiende al padre
59        }else{
60          node.bB.bNode.x = -999999;
61        }
62      }
63    };
64
65    //calculo de la coordenada y
66    function performLayoutY(){
67
68      //recorre los nodos por nivel
69      visibleMatrix[0][0].bB.bNode.y = 0;
70      for(var i = 1; i < Object.keys(visibleMatrix).length; ++i){
71
72        //busca el nodo mas grande en el nivel superior
73        var maxLast = 0;
74        for(var j = 0; j<visibleMatrix[i-1].length; ++j)
75          if(maxLast < visibleMatrix[i-1][j].
                bB.bNode.heightOutHalf)
76          maxLast = visibleMatrix[i-1][j].bB.bNode.heightOutHalf;
77
78        //busca el nodo mas grande en el nivel del recorrido
              actual
79        var max = 0;
80        for(var j = 0; j<visibleMatrix[i].length; ++j)
81          if(max < visibleMatrix[i][j].bB.bNode.heightOutHalf)
82          max = visibleMatrix[i][j].bB.bNode.heightOutHalf;
83
84        //establece la coordanada y
85        var y = visibleMatrix[i-1][0].bB.bNode.y + maxLast +
              self.style.marginTop + max;
86        for(var j = 0; j<visibleMatrix[i].length; ++j)
87          visibleMatrix[i][j].bB.bNode.y = y;
88      }
89    }
90
91    performLayoutX(rootNode);
92    performLayoutY();
93    boundingBoxLoad();  \\carga el boundingbox
94  }
```

Render.draw Renderiza el árbol en el canvas.

Sintaxis

```
Render.drawTree();
```

Render.clear Borra el árbol en el `canvas`.

Sintaxis

```
Render.clearTree();
```

getNodeByPos Obtiene el nodo del árbol que ocupa la posición `pos`.

Sintaxis

```
Tree.getNodeByPos(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Retorno

Retorna el nodo en caso de ser existir.

Retorna `null` en caso contrario.

4.2.5. Interfaz ExportToLatex

La interfaz ExportToLatex proporciona un conjunto de propiedades y métodos que permiten la traducción del árbol de tipo Tree a LaTeX. Esta clase funciona como una extensión de la clase Tree, por lo tanto la clase Tree debe heredarla para poder usar sus operaciones.

4.2.5.1. Métodos

exportPstTree Traduce el árbol a LaTeX utilizando el paquete ps-tree.

Sintaxis

```
ExportToLatex.exportPstTree();
```

Retorno

Retorna un texto en LaTeX.

4.2.6. Interfaz SubTree

La interfaz SubTree proporciona un conjunto de propiedades y métodos que permiten almacenar un subconjunto de nodos de un árbol. Este subconjunto no permite elementos duplicados.

4.2.6.1. Selección de nodos

La selección de nodos permite indicar qué elementos se desea modificar simultáneamente. Esta selección se organiza por grupos de nodos que pertenecen al mismo árbol, la razón se debe a que, hacer los cálculos primero a cada uno de los objetos relacionados y después redibujarlos todos de una vez, es más eficiente que operar y redibujar cada vez por nodo[2].

Con un pequeño número de nodos seleccionados por árbol quizás no se aprecie un aumento significativo del rendimiento, pero a mayor número de selección se reduce la cantidad de refrescamiento en el canvas.

La clase Forest contiene el conjunto de todos los árboles, un subconjunto de estos árboles conformarían un subbosque.

4.2.6.2. Propiedades

tree Apuntador a la instancia Tree para identificar el árbol al que pertenece la lista de `nodes`.

Sintaxis

```
tree;
```

nodes Arreglo de apuntadores de los nodos seleccionados que pertenece la propiedad tree.

Sintaxis

[2]Razón por la cual la mayoría de los métodos de la Interfaz Tree reciben como parámetro un arreglo de nodos

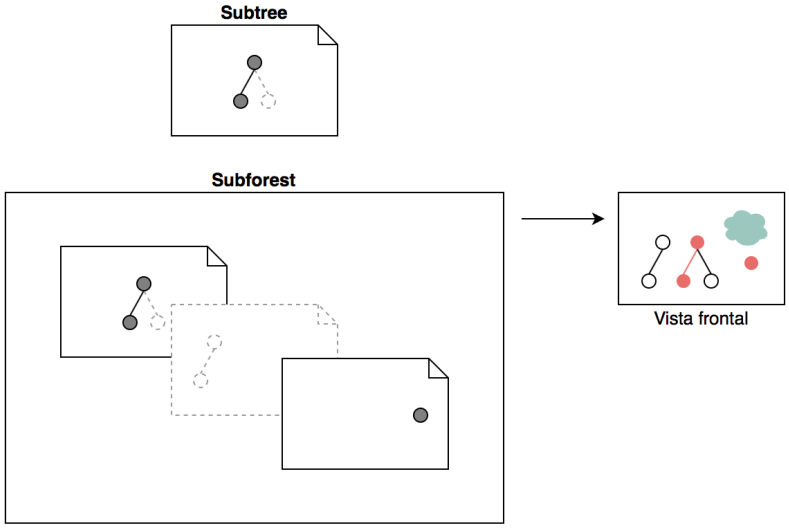


Figura 4.8: Selección de nodos por árbol

```
nodes;
```

4.2.6.3. Métodos

addNode Añade el nodo `node` si éste no se repite.

Sintaxis

```
SubTree.addNode(node);
```

Parámetro

`node`: nodo tipo de Node.

Retorno

Retorna `true` si el nodo se añade.

Retorna `false` en caso contrario.

removeNode Remueve el nodo `node` si éste existe.

Sintaxis

```
SubTree.removeNode(node);
```

Parámetro

`node`: nodo de tipo Node.

Retorno

Retorna `true` si el nodo se remueve.

Retorna `false` en caso contrario.

4.2.7. Interfaz Forest

La interfaz Forest proporciona un conjunto de propiedades y métodos que permiten gestionar la estructura, presentación y renderización de uno o más árboles.

Forest administra los árboles como un arreglo de capas superpuestas, como anteriormente se ha descrito. Adicionalmente a estas capas posee una capa frontal.

4.2.7.1. Capa frontal

La capa frontal sirve como una especie de borrador para dibujar anotaciones, selecciones o cualquier otra cosa sin afectar el dibujo de las capas árboles. Por ejemplo: renderizar el subconjunto de nodos almacenados en una variable SubTree para simular la selección de nodos.

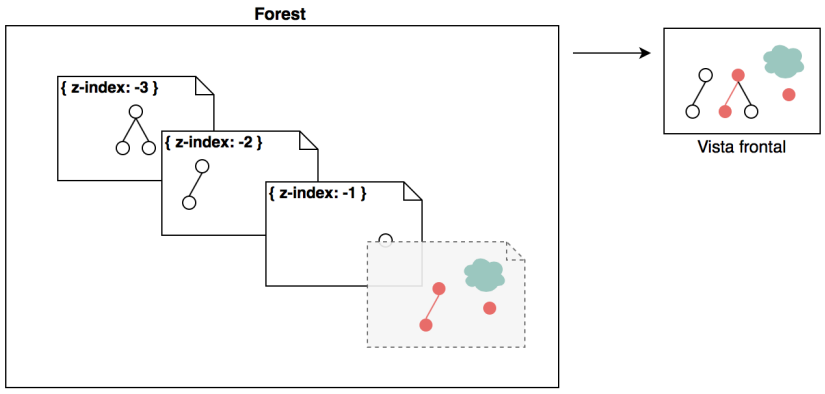


Figura 4.9: Capa frontal canvas en Forest

4.2.7.2. Propiedades

eDrawspace Elemento que representa el espacio de dibujo del editor, éste es el contenedor de toda las capas `canvas`.

Sintaxis

```
eWorkspace;
```

canvas Elemento canvas frontal.

Sintaxis

```
canvas;
```

context Contexto de renderización 2D del `canvas` frontal.

Sintaxis

```
context;
```

trees Arreglo de Tree para indicar los árboles que componen el bosque.

Sintaxis

```
trees;
```

subTrees Arreglo de Subtree para indicar los nodos seleccionados por árbol.

Sintaxis

```
subTrees;
```

bBPhoto JSON para indicar el *boundingbox* o recorte de pantalla para realizar una captura sobre el espacio de dibujo del editor.

Sintaxis

```
bBPhoto;
```

treeRawCopy Copia de un árbol en formato JSON.

Sintaxis

```
treeRawCopy;
```

isScaling booleano que indica si la opción de escalar esta habilitada.

Sintaxis

```
isScaling;
```

record instancia de Record para manipular el historial del editor.

Sintaxis

```
record;
```

4.2.7.3. Métodos

resize Reescala el `canvas` frontal.

Sintaxis

```
resize(pos);
```

pushTree añade otro árbol de tipo Tree al bosque, que consecuentemente apila una capa de elemento canvas en el editor.

Sintaxis

```
Forest.pushTree();
```

Implementación

```
this.pushTree = function(canvasId){

  //genera el atributo id del canvas. Obtiene el numero mas bajo
      que no este siendo usado por una capa
  var generateCanvasId = function(){
    var ids = [];
    if(trees.length > 0){

      //lista todos los id
      for(var i = 0; i < trees.length; ++i)
        ids[trees[i].getCanvasId()] = true;

      //en orden ascendente comenzando desde 0 retorna el primer
          numero sin usar
      for(var i = 0; i < trees.length; ++i)
        if(ids[i] === undefined)
```

```
            return i;
        }
        return trees.length;
    }

    //crea una nueva intancia Tree, si el id es pasado por
        parametro lo uitliza, de lo contrario genera uno
    var tree = canvasId ? new Tree(canvasId, eDrawspace) : new
        Tree(generateCanvasId(), eDrawspace);

    //inserta el canvas en el espacio de dibujo
    if(trees.length === 0){
        eDrawspace.appendChild(tree.getCanvas());
    }else{
        eDrawspace.insertBefore(tree.getCanvas(), trees[trees.length
            - 1].getCanvas());
        for(var i = 0; i < trees.length; ++i)
            trees[i].getCanvas().setAttribute("style", "z-index: " + (
                i - trees.length - 1));
    }

    //apila la instancia en la propiedad trees del Forest
    trees.push(tree);
    return tree;
};
```

removeTree remueve un árbol del bosque, por lo tanto retira una capa canvas del editor.

Sintaxis

```
Forest.removeTree();
```

getNodeByPos Obtiene el nodo del bosque que ocupa la posición `pos`.

Sintaxis

```
Forest.getNodeByPos(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Retorno

Retorna el nodo en caso de ser existir.

Retorna `null` en caso contrario.

Render.draw Renderiza el bosque, es decir, dibuja todos los árboles sus respectivas capas `canvas`.

Sintaxis

```
Forest.draw();
```

Render.clear Limpia el bosque, es decir, borra todos los árboles de sus respectivas capas `canvas`.

Sintaxis

```
Forest.clear();
```

SubForest.addNode Añade el nodo `node` a `subTrees`. El método se encarga de añadir el nodo al subárbol de `subTrees` que le corresponde sin repeticiones.

Sintaxis

```
Forest.SubForest.addNode();
```

SubForest.removeNode Remueve el nodo `node` de `subTrees`.

Sintaxis

```
Forest.SubForest.removeNode();
```

Selection.nodeToogle añade o remueve de la selección de nodos, el nodo que ocupa la posición `pos`. Los nodos seleccionados se almacenan en la propiedad `subTrees`.

Sintaxis

```
Forest.Selection.nodeToogle();
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Selection.nodeSimple mantiene seleccionado como único nodo, el nodo que ocupa la posición `pos`. El nodo seleccionado se almacena en la propiedad `subTrees`.

Sintaxis

```
Forest.Selection.nodeSimple(pos);
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Add.root Añade un nodo raíz al bosque en la posición `pos`. Utilizado para especificar un árbol n-ario.

Sintaxis

```
Forest.Add.root();
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Add.rootBinary Añade un nodo raíz al bosque en la posición `pos`. Utilizado para especificar un árbol binario.

Sintaxis

```
Forest.Add.rootBinary();
```

Parámetros

`pos`: JSON con los números de las coordenadas bidimensionales. Por ejemplo: {x: 10, y: 15}.

Add.child Añade hijos a los nodos del bosque de `subTrees`.

Sintaxis

```
Forest.Add.child();
```

Add.left Añade un hermano izquierdo a los nodos del bosque de `subTrees` en árboles n-arios o un hijo izquierdo en árboles binario.

Sintaxis

```
Forest.Add.left();
```

Add.right Añade un hermano derecho a los nodos del bosque de `subTrees` en árboles n-arios o un hijo derecho en árboles binario.

Sintaxis

```
Forest.Add.right();
```

Add.remove Remueve los nodos del bosque de `subTrees`.

Sintaxis

```
Forest.Add.remove();
```

expand Expande o contrae la renderización de la descendencia a partir de los nodos de `subTrees`.

Sintaxis

```
Forest.expand();
```

copy Copia un árbol del bosque a partir del primer nodo de `subTrees`. La copa se genera en formato `raw` y es asignada a la propiedad `treeRawCopy`.

Sintaxis

```
Forest.copy();
```

cut Corta un árbol del bosque a partir del primer nodo de `subTrees`. Cuando se corta, a parte de removerse del árbol, se genera una copia en formato *raw* del subárbol cortado. La copia es asignada a la propiedad `treeRawCopy`.

Sintaxis

```
Forest.cut();
```

paste Pega el árbol `treeRawCopy` a los nodos de `subTrees`.

Sintaxis

```
Forest.paste();
```

Export.original Traduce el bosque en formato *raw*.

Sintaxis

```
Forest.Export.original();
```

Export.png Traduce el primer árbol de `subTrees` a formato PNG.

Sintaxis

```
Forest.Export.png();
```

Export.pstTree Traduce el primer árbol de `subTrees` a formato LaTeX con paquete pst-tree.

Sintaxis

```
Forest.Export.pstTree();
```

import Importa un bosque en formato `raw`.

Sintaxis

```
Forest.import(doc);
```

translate Traslada la posición de los árboles de `subTrees`.

Sintaxis

```
Forest.translate();
```

ResizeNode.width Reescala la anchura de los nodos de `subTrees` por `width`.

Sintaxis

```
Forest.ResizeNode.width(width);
```

Parámetros

`width`: número de anchura en px.

ResizeNode.height Reescala la altrua de los nodos de `subTrees` por `height`.

Sintaxis

```
Forest.ResizeNode.height(height);
```

Parámetros

`height`: número de altura en px.

ResizeNode.lineWidthNode Reescala el grosor de línea de los nodos de `subTrees` por `lineWidth`.

Sintaxis

```
Forest.ResizeNode.lineWidthNode(lineWidth);
```

Parámetros

`lineWidths`: número de tamaño de línea en px.

ResizeNode.lineWidthEdge Reescala el grosor de línea de las aristas de los nodos de `subTrees` por `lineWidth`.

Sintaxis

```
Forest.ResizeNode.lineWidthEdge(lineWidth);
```

Parámetros

`lineWidths`: número de tamaño de línea en px.

ResizeMargin.top Reescala la distancia vertical entre los nodos de los árboles de `subTrees` por `margin`.

Sintaxis

```
Forest.ResizeMargin.top(margin);
```

Parámetros

`margin`: número de distancia en px.

ResizeMargin.left Reescala la distancia horizontal entre los nodos de los árboles de `subTrees` por `margin`.

Sintaxis

```
Forest.ResizeMargin.left(margin);
```

Parámetros

`margin`: número de distancia en px.

Style.shape Establece la forma de los nodos de `subTrees` por `shape`.

Sintaxis

```
Forest.Style.shape(shape);
```

Parámetros

`shape`: string de la forma: "arc" para nodos circulares o "rect" para nodos rectangulares.

Style.Color.fillStyleNode Establece el color de relleno de los nodos de `subTrees` por `fillStyle`.

Sintaxis

```
Forest.Style.Color.fillStyleNode(fillStyle);
```

Parámetros

`fillStyle`: string que indica los colores en RGB.

Style.Color.strokeStyleNode Establece el color de línea de los nodos de `subTrees` por `strokeStyle`.

Sintaxis

```
Forest.Style.Color.strokeStyleNode(strokeStyle);
```

Parámetros

`fillStyle`: string que indica los colores en RGB.

Style.Color.strokeStyleEdge Establece el color de línea de las aristas de los nodos de `subTrees` por `strokeStyle`.

Sintaxis

```
Forest.Style.Color.strokeStyleEdge(strokeStyle);
```

Parámetros

`fillStyle`: string que indica los colores en RGB.

Style.LineDash.lineDashNode Establece el tipo de línea de los nodos de `subTrees` por `strokeStyle`.

Sintaxis

```
Forest.Style.LineDash.lineDashNode(lineDash);
```

Parámetros

`lineDash`: Un arreglo. Una lista de números que especifica las distancias para dibujar una línea alterna y una brecha. Por ejemplo, [5, 15].

Style.LineDash.lineDashEdge Establece el tipo de línea de las aristas de los nodos de `subTrees` por `strokeStyle`.

Sintaxis

```
Forest.Style.LineDash.lineDashEdge(lineDash);
```

Parámetros

`lineDash`: Un arreglo. Una lista de números que especifica las distancias para dibujar una línea alterna y una brecha. Por ejemplo, [5, 15].

Text.nameNode Establece el valor de los nodos de `subTrees` por `nameNode`.

Sintaxis

```
Forest.Text.nameNode(nameNode);
```

Parámetros

`nameNode`: string.

Text.nameEdge Establece el valor de las aristas de los nodos de `subTrees` por `nameNode`.

Sintaxis

```
Forest.Text.nameEdge(nameEdge);
```

Parámetros

`nameEdge`: string.

Text.opcional Establece el valor de observación o comentario de los nodos de `subTrees` por `nameNode`.

Sintaxis

```
Forest.Text.opcional(opcional);
```

Parámetros

`opcional`: string.

Record.undo Deshace una acción del usuario, retrocediendo al estado del bosque justo antes de ejecutar la acción.

Sintaxis

```
Forest.Record.undo();
```

Record.redo Rehace una acción del usuario, avanzando al estado del bosque después de ejecutar la acción.

Sintaxis

```
Forest.Record.redo();
```

Photo.capture realiza una captura en formato PNG sobre el espacio de dibujo.

Sintaxis

```
Forest.Photo.Photo();
```

4.2.8. Interfaz Record

La interfaz UpdateDom proporciona un conjunto de propiedades y métodos que permiten gestionar el historial del editor. En ella se almacenan los datos mínimos necesarios para recrear un estado previo o posterior del bosque.

4.2.8.1. Propiedades

records arreglo de los estados.

Sintaxis

```
records();
```

iCurrent indica la posición actual en el arreglo de estados `records`.

Sintaxis

```
iCurrent();
```

iCheckpoint indica el indice del punto de control en el arreglo de estados `records`. El punto de control marca el ultimo estado guardado.

Sintaxis

```
iCheckpoint();
```

4.2.8.2. Métodos

rec almacena los estados en el arreglo`records`.

Sintaxis

```
1  Record.rec(command, dataNodes);
```

Parámetros

`command`: tipo de acción

`dataNodes`: datos necesarios para recrear la acción.

UndoRecord retrocede un estado.

Sintaxis

```
1  Record.UndoRecord();
```

Retorno

Retorna un arreglo de nodos de tipo Node que deberían estar seleccionados después de rehacer un estado.

Retorna null en caso contrario

RedoRecord avanza un estado.

Sintaxis

```
1  Record.RedoRecord();
```

Retorno

Retorna un arreglo de nodos de tipo Node que deberían estar seleccionados después de rehacer un estado.

Retorna null en caso contrario

4.2.9. Diagrama de clases TreeGraph

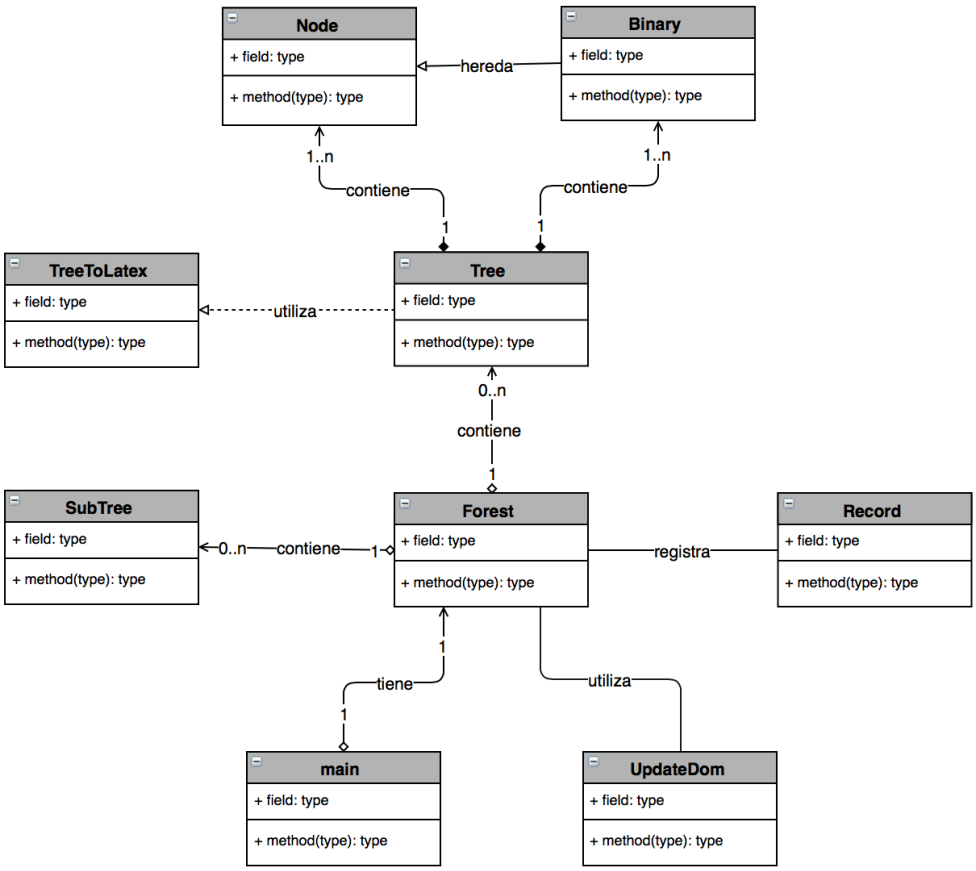


Figura 4.10: Diagrama de clases TreeGrpah simplificado

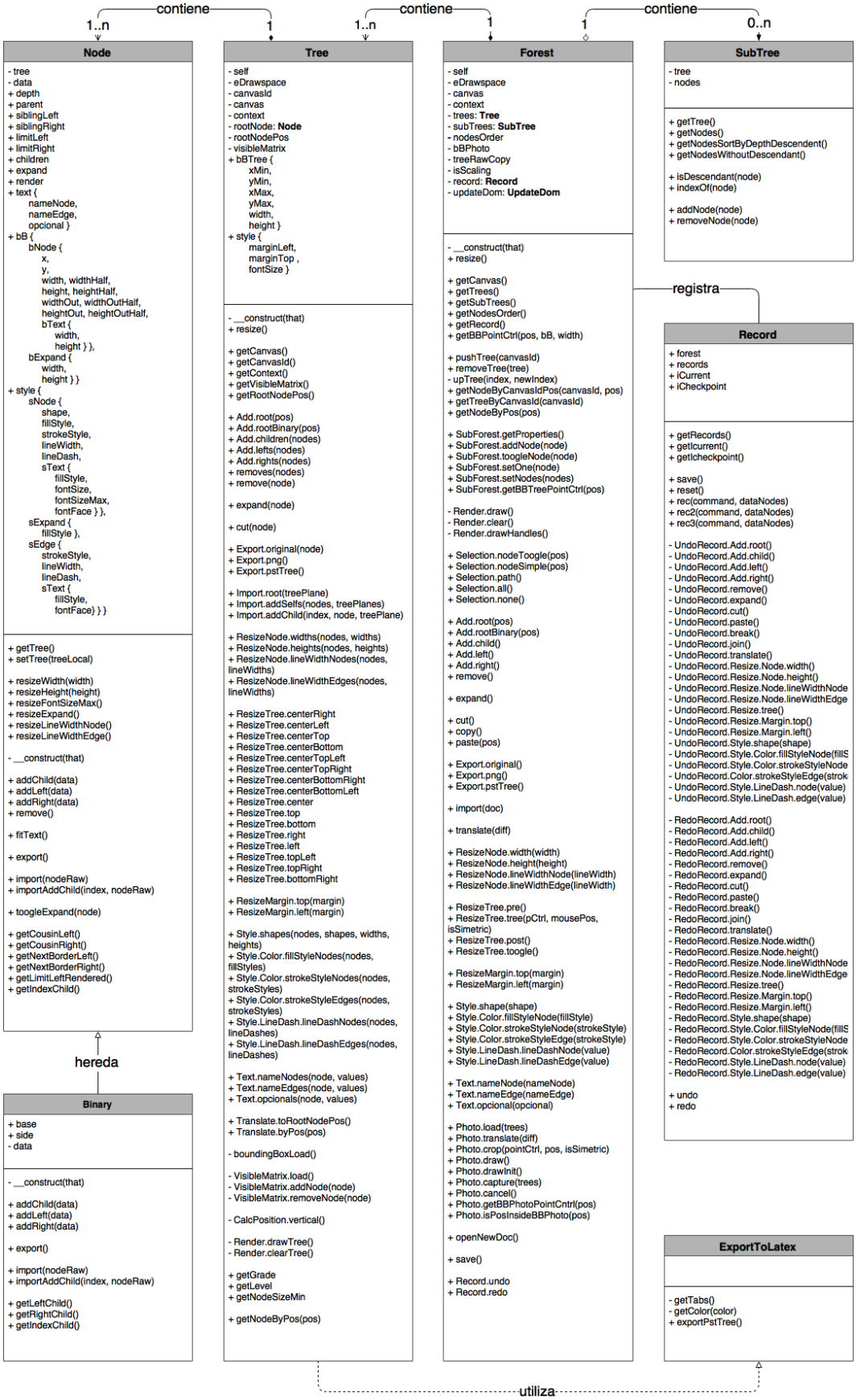


Figura 4.11: Diagrama de clases TreeGrpah

4.3. Módulo de gestión de cuentas de usuario y documentos

Este módulo se encarga de proveer las operaciones que se requiere para el acceso mediante cuentas de usuario y el almacenamiento de los árboles dentro de la cuenta del usuario que los construye.

En la implementación de este módulo se anexa (en la Vista) la implementación del módulo anterior "Editor gráfico de árboles", y para así completar el sistema.

A continuación, se describe la arquitectura del sistema de este módulo junto con las clases de mayor relevancia.

4.3.1. Arquitectura

Se utiliza la arquitectura MVC basado en el framework Ruby on Rails. En la Figura 4.12 se ilustra la arquitectura implementada, básicamente la MVC un poco más detallada.

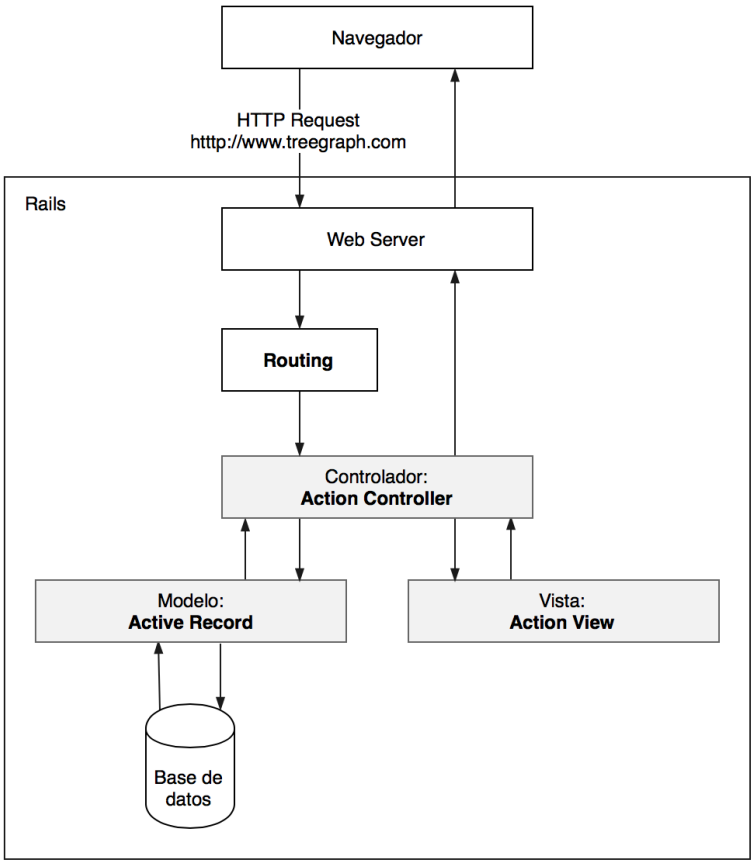


Figura 4.12: Arquitectura del módulo de almacenamiento de árboles y cuentas de usuario

Modelo compuesto por las clases responsables de representar los datos, y la lógica de negocio y de datos de la aplicación. Estas clases o modelos heredan de `ActiveRecord::Base`.

Active Record es una biblioteca de Ruby para trabajar con bases de datos relacionales SQL, que facilita la creación y el uso de objetos de negocio cuyos datos requieren un almacenamiento persistente a una base de datos. Active Record es una implementación del patrón Active Record que en sí es una descripción de un sistema de **Mapeo Relacional de Objetos** (abrevidado como ORM por sus siglas en inglés Object Relational Mapping).

Los objetos Active Record no especifican sus atributos directamente, se deducen a partir de la definición de tabla con las que están vinculados. Añadir, eliminar, cambiar atributos y su tipo se realiza directamente en la base de datos. Cualquier cambio se refleja inmediatamente en los objetos Active Record. El mapeo que se une a una clase Active Record dada a una determinada tabla se producirá automáticamente en los casos más comunes, pero puede ser sobrescrito.

Los modelos son creados en el directorio `/app/models/` de la aplicación y son nombrados de acuerdo a la convención de nombres del framework:

- Tablas de base de datos: plural con guiones de separación de las palabras (por ejemplo, book_clubs).
- Clase modelo: singular con la primera letra de cada palabra en mayúsculas (por ejemplo, Bookclub).

Controlador compuestos por las clases responsables de gestionar las solicitudes y producir las salidas apropiadas con ayuda tanto del Modelo como de la Vista. Estas clases o controladores heredan de **ApplicationController**, y éste, de forma predeterminada, hereda de `ActionController:Base`.

Los **Action Controllers** son la base de las peticiones web en Rails. Se componen de una o más acciones que se ejecutan bajo petición y entonces o bien se hace una plantilla o se redirige hacia otra acción. Una **acción** se define como un método público en el controlador al alcance del servidor web a través del enturatodor de Rails.

El **enrutador** de Rails a partir del reconocimiento de las URLs y en las rutas definidas en el directorio `/config/routes.rb`, determina qué controlador y acción debe ejecutar cuando la aplicación recibe una petición. Luego, Rails crea una instancia de ese controlador y corre el método con el mismo nombre que la acción.

Un controlador puede ser pensado como el intermediario entre los modelos y las vistas. El controlador hace que los datos de los modelos estén disponibles en las vistas para el usuario, y que sean guardados o actualizados desde el usuario hacia el modelo.

Los controladores son creados en el directorio `/app/controllers/` de la aplicación y la convención de nomenclatura de los controladores en Rails favorece la pluralización de la última palabra en el nombre del controlador, aunque no es estrictamente

necesario (por ejemplo ApplicationController). Por ejemplo, es preferible UsersController que UserController o DocumentsController que DocumentController.

Vista compuesto por las plantillas que permiten visualizar los datos que el Controlador ha recogido. De forma predeterminada, estas plantillas son reproducidas, luego de que se ejecuta la acción de un controlador con su mismo nombre. Por lo tanto, cada acción debería tener su plantilla.

Action View y Action Controller son componentes de **Action Pack**. Este componente se encarga de manejar las peticiones web, divide el trabajo entre el controlador (realización de lógica) y la vista (convirtiendo una plantilla). Por lo general, Action Controller se ocupa de la comunicación con la base de datos y la realización de acciones CRUD cuando es necesario (reservando el código "pesado" para los modelos), y Action View es responsable de la compilación de la respuesta.

Las plantillas Action View permiten incrustar código Ruby dentro del HTML y compartir los datos con los controladores a través de variables mutuamente accesibles [25]. Rails se encarga de traducir las plantillas a HTML "puro" antes de enviarlos al usuario y puedan ser entendidos por el navegador.

Cada acción debería tener su plantilla con el mismo nombre y dentro de un fichero con el mismo nombre del controlador que contiene dicha acción, a su vez, éste debe estar en el directorio `app/views/` de la aplicación. Las plantillas deben tener la extensión `.erb`. Por ejemplo, la acción `index` de DocumentsController pudiera reproducir la plantilla `app/views/documents/index.html.erb` por defecto después de asignar el valor a la variable de instancia `@documets`.

4.3.1.1. Contexto tecnológico

Las interfaces de usuario se desarrollan mediante **HTML5** con código Ruby emebebido (plantillas **ERB**), **CSS3**, **JavaScript** 1.8 y JQuery.

En el desarrollo del servicio de aplicaciones, y como ya es notable, se utiliza el lenguaje de programación Ruby 2.2.3 con el apoyo del framework **Ruby on Rails** 4.2.4.

En el sistema de gestión de base de datos relacional se utiliza **MySQL** 5.6.

Para el servidor web se utiliza **WEBrick** 1.3.1 (un servidor web que trae Ruby por defecto).

4.3.2. Modelos

La aplicación define únicamente dos modelos:

User Vinculada a la tabla users, es la clase que representa los datos de usuario con los cuales opera el sistema.

Un usuario puede tener de ninguno a varios documentos, por lo tanto, el modelo User mantiene una relación opcional *oneToMany* con el modelo Document, relación que se define en la clase de la siguiente manera:

```
1 class User < ActiveRecord::Base
2   has_many :documents, dependent: :destroy #relacion de cero a
        muchos
3 end
```

Está conformada por los métodos CRUD[3] y las consultas que obtienen la listas de los usuarios ordenados según el campo. Además, en esta clase se especifican las validaciones de los datos antes de ser persistidos (validaciones que también están implementadas en el lado del cliente antes de enviar las peticiones al servidor).

Documents Vinculada a la tabla documents, es la clase que representa los datos de los documentos creado por los usuario. Un documento contiene los árboles dibujados en una hoja del editor.

Un documento obligatoriamente debe pertenecer a un solo usuario, por lo tanto, el modelo Document mantiene una relación *oneToOne* con el modelo Document, relación que se define en la clase de la siguiente manera:

```
1 class Document < ActiveRecord::Base
2   belongs_to :user #relacion de uno a uno
3 end
```

Al igual que en User, está conformada por los métodos CRUD y las consultas que obtienen la listas de los documentos ordenados según el campo. Además, en esta clase se especifican las validaciones de los datos antes de ser persistidos (validaciones que también están implementadas en el lado del cliente antes de enviar las peticiones al servidor).

4.3.3. Controladores

Se definen cuatro controladores:

UsersController encargadas de gestionar las peticiones de los usuarios y responder la información solicitada a través de las vistas users.

DocumentsController encargadas de gestionar las peticiones de los usuarios y responder la información solicitada a través de las vistas documents.

[3]Es el acrónimo de Crear, Leer, Actualizar y Borrar (del original en inglés: Create, Read, Update and Delete)

HomeController encargada de gestionar las peticiones de los usuarios y responder la información solicitada a través de la vista home, página principal de la aplicación.

SessionController encargada de crear y destruir las sesiones de usuario.

4.3.4. Vistas

Se definen las plantillas para los controladores de UsersController, DocumentsController y HomeController. En la siguiente sección se muestra el aspecto de las plantillas después de ser interpretadas por el navegador.

Como anteriormente se mencionó en los modelos, los campos de los formularios de las vistas están validadas antes ejecutar la petición a la acción.

users contiene las plantillas que permiten acceder y modificar los registros del modelo User. Visible únicamente para los usuarios registrados.

documents contiene las plantillas que permiten acceder y modificar los registros del modelo Document. Visible para todos los usuarios que ingresen a la aplicación.

Para modificar el contenido de un objeto Document se utiliza el editor de árboles, es aquí donde se incluye en una de las plantillas (`/app/views/documents/show.html.erb`) el documento HTML5 del editor que funciona en conjunto con las interfaces JavaScript mencionadas en el "Módulo de editor de árboles".

home contiene una única plantilla, la que define la página de inicio de la aplicación.

4.4. Diseño de la interfaz gráfica de usuario

4.4.1. Inicio

En la página de inicio, mostrada en la Figura 4.13, se encuentra el logo, una brevísima descripción de lo que ofrece la aplicación, y el primer paso para ingresar al editor por medio de la elección de alguna de las dos siguientes alternativas:

Logear permite ingresar al editor desde la cuenta del usuario. Mediante el botón *Logear* se presenta el formulario que solicita los datos necesarios para la autenticación. La Figura 4.14 muestra la interfaz para esta validación.

Si el usuario no posee una cuenta, puede solicitarla o **registrarse** completando el formulario después de presionar *registrarse*. En la Figura 4.15 se muestra el formulario de registro.

Probar el usuario ingresa directamente al editor sin tener que iniciar una sesión. Esta opción tiene la desventaja de no permitir almacenar los documentos. Mediante el botón *Probar* el usuario opta por este método de ingreso.

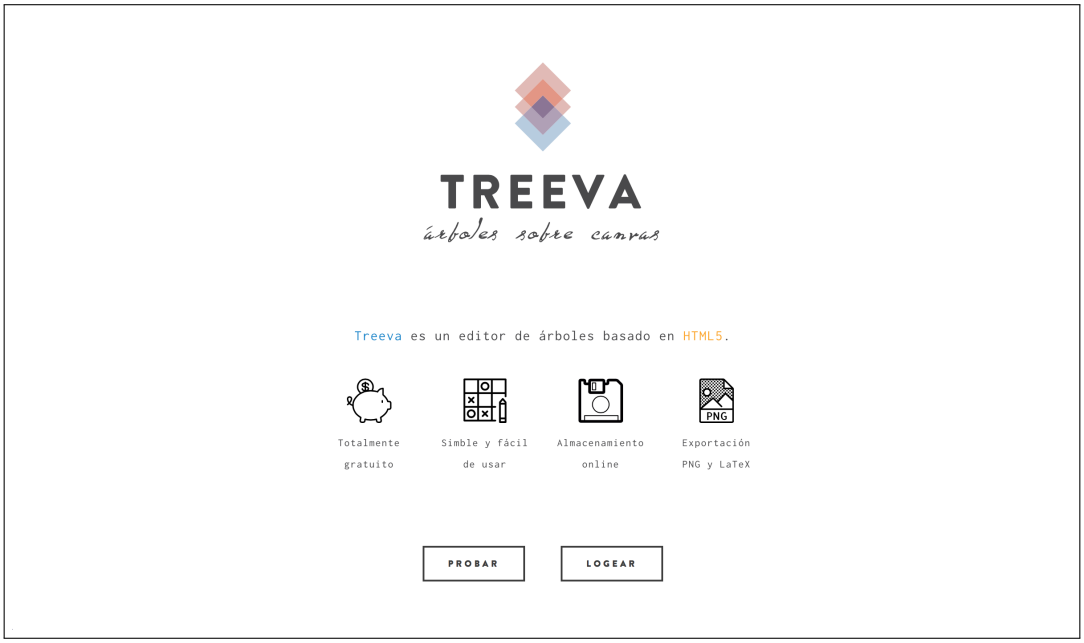


Figura 4.13: Página de inicio

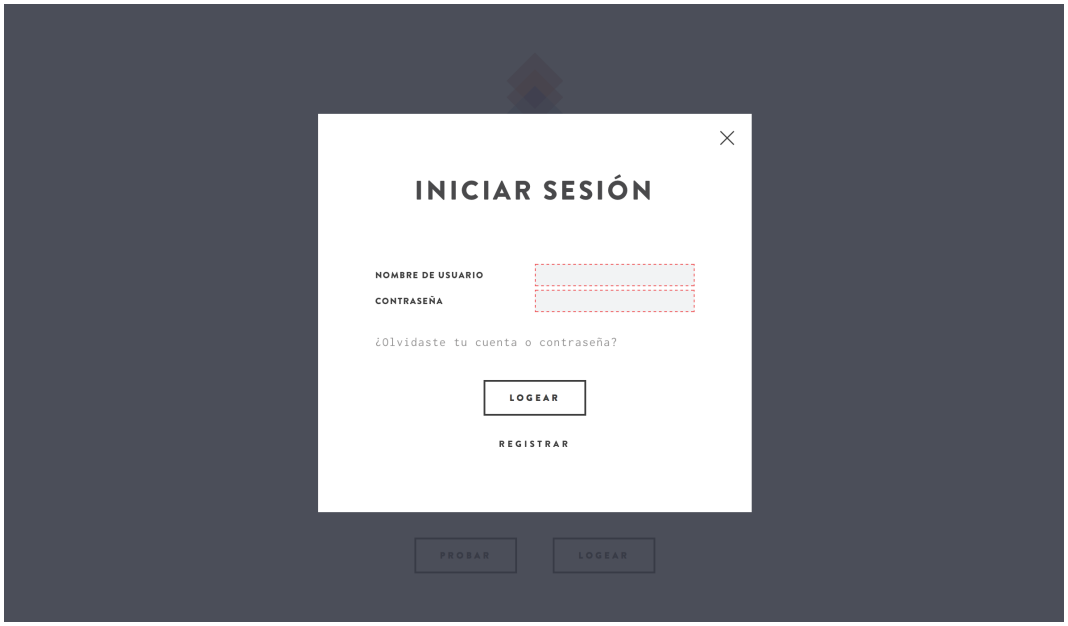


Figura 4.14: Inicio de sesión

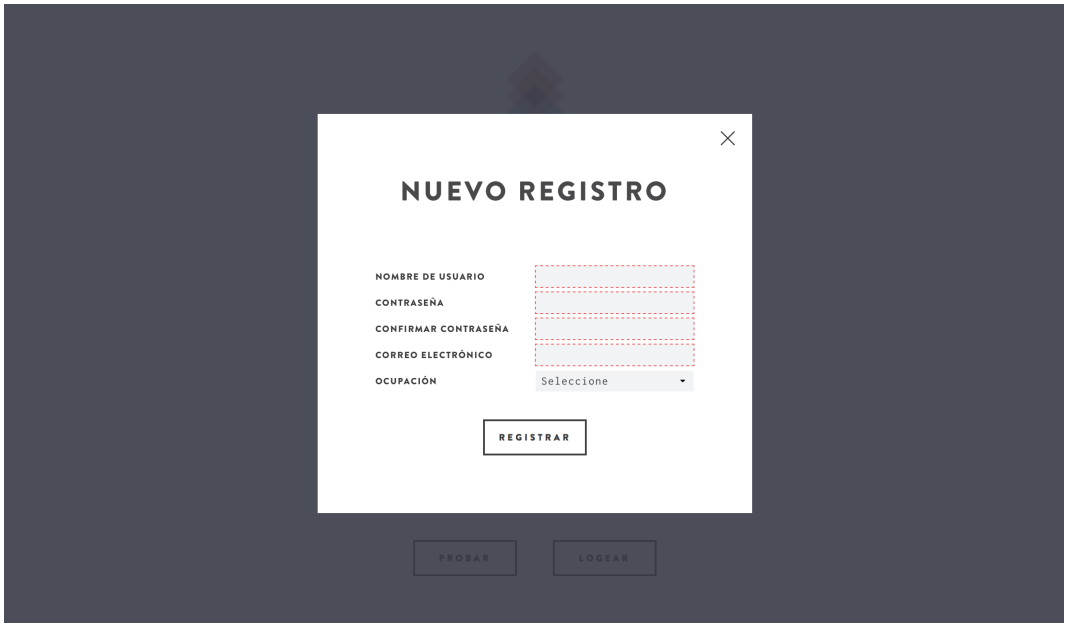


Figura 4.15: Registro de usuario

4.4.2. Menú de navegación

El menú de navegación ofrece una serie de opciones que el usuario puede elegir para **redirigirse** a la diferentes páginas o secciones de la aplicación, incluyendo el **cierre de sesión**.

La lista del menú se adapta al tipo de usuario que ingresa al sistema, puesto que no todas las secciones están disponibles para todos los usuarios:

- Usuarios administradores: *Documentos*, *Usuario* y *Cuenta* del menú. La Figura 4.16 muestra el menú de visualizan los administradores.
- Usuarios no administradores: *Documentos* y *Cuenta*.
- Usuarios no registrados: no es visible.

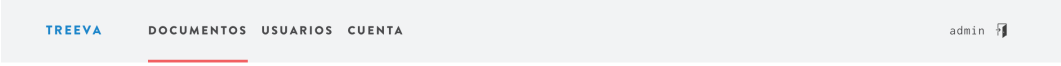


Figura 4.16: Menú de navegación para administradores

4.4.3. Sección Usuarios

Se accede mediante la opción *Usuarios* del menú de navegación y está disponible solo para los administradores. La página dispone de las siguientes operaciones:

Listar es la operación por defecto al ingresar a la página. Mediante una tabla se lista los usuarios registrados en el sistema, exceptuando el usuario de la sesión. En la Figura 4.17 se tiene la vista global de la sección, compuesto por la tabla de los registros, y la barra de opciones en la parte superior de la tabla, con botones que al ser presionados permiten añadir, editar y eliminar los elementos de la lista.

Nuevo permite añadir nuevos usuarios al sistema. Mediante el botón *Nuevo* se presenta una modal que contiene el formulario por el cual se solicita al usuario los datos necesarios para persistir un nuevo registro. En la Figura 4.18 se muestra la interfaz para añadir un registro.

Editar permite actualizar los datos de un usuario seleccionado en la tabla. Mediante el botón *Editar* se presenta una modal que contiene el formulario por el cual se solicita al usuario los datos necesarios para modificar un registro. En la Figura 4.19 se muestra la interfaz para editar un registro.

Eliminar permite eliminar uno o varios registros simultáneamente seleccionados en la tabla. Mediante el botón *Eliminar* se remueve los registros, sin embargo, antes de ejecutar la acción se solicita la confirmación del usuario para proceder. En la Figura 4.20 se muestra la interfaz de confirmación.

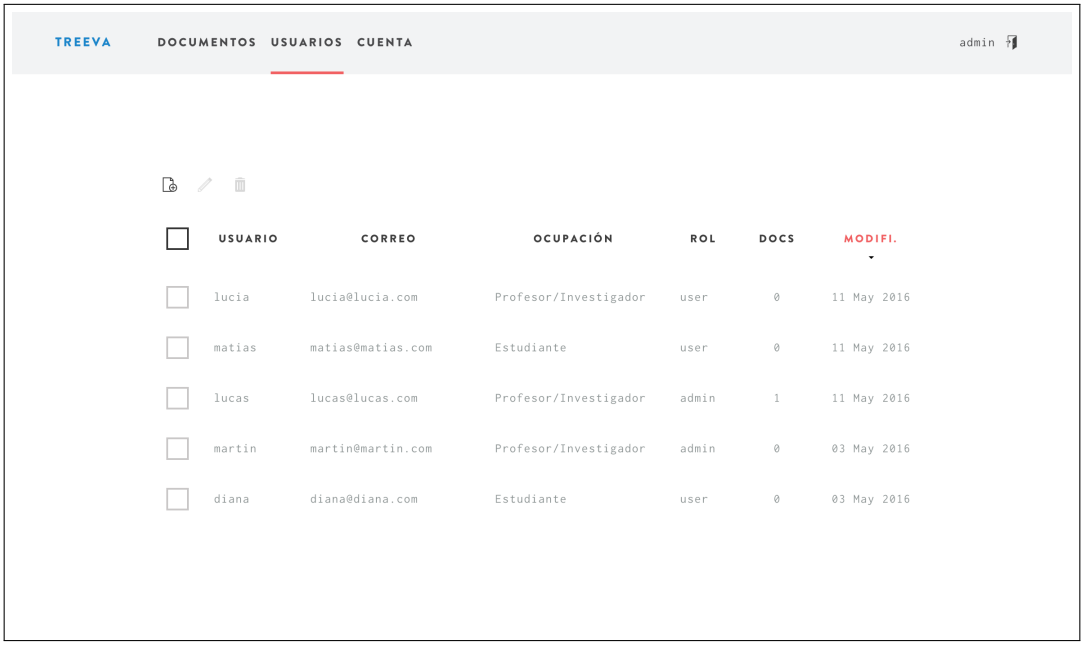


Figura 4.17: Página de Usuarios: listar

Figura 4.18: Página de Usuarios: nuevo

Figura 4.19: Página de Usuarios: editar

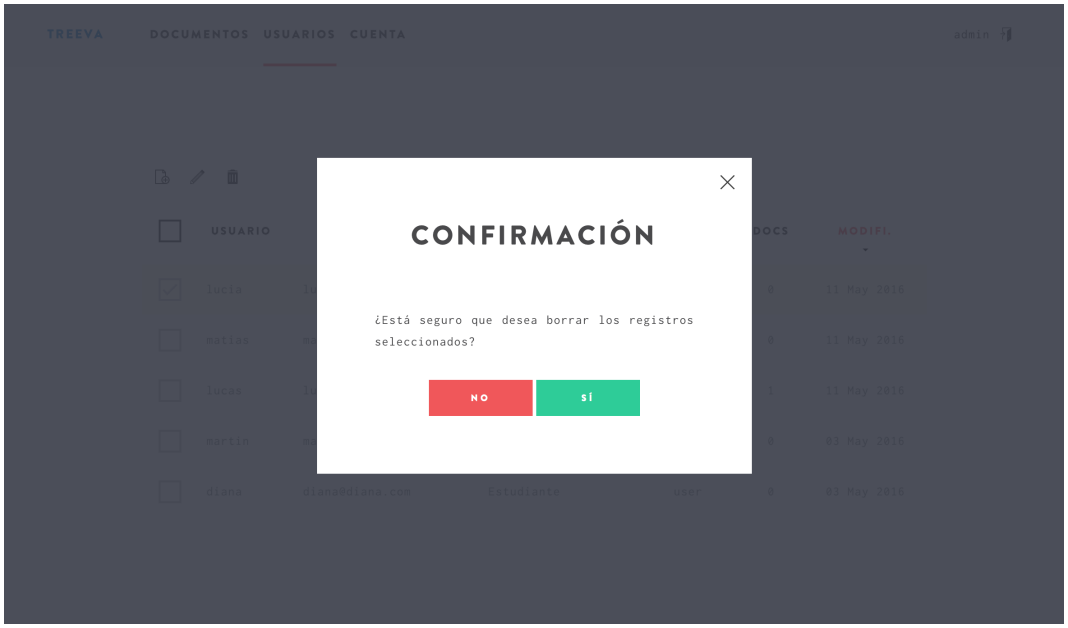


Figura 4.20: Página de Usuarios: eliminar

4.4.4. Sección Cuenta

Se accede mediante la opción *Cuenta* del menú de navegación y se encuentra accesible solo para los usuarios registrados. La página dispone de la siguiente operación:

Editar permite actualizar los datos de cuenta del usuario de la sesión, específicamente los valores de *correo electrónico*, *ocupación* y *contraseña*. En la Figura 4.21 se muestra el contexto general de la página, compuesto por dos formularios por los cuales se solicitan los datos al usuario. Un formulario para actualizar los atributos de *correo electrónico* y *ocupación*, y un segundo formulario para cambiar la *contraseña* de ingreso al sistema.

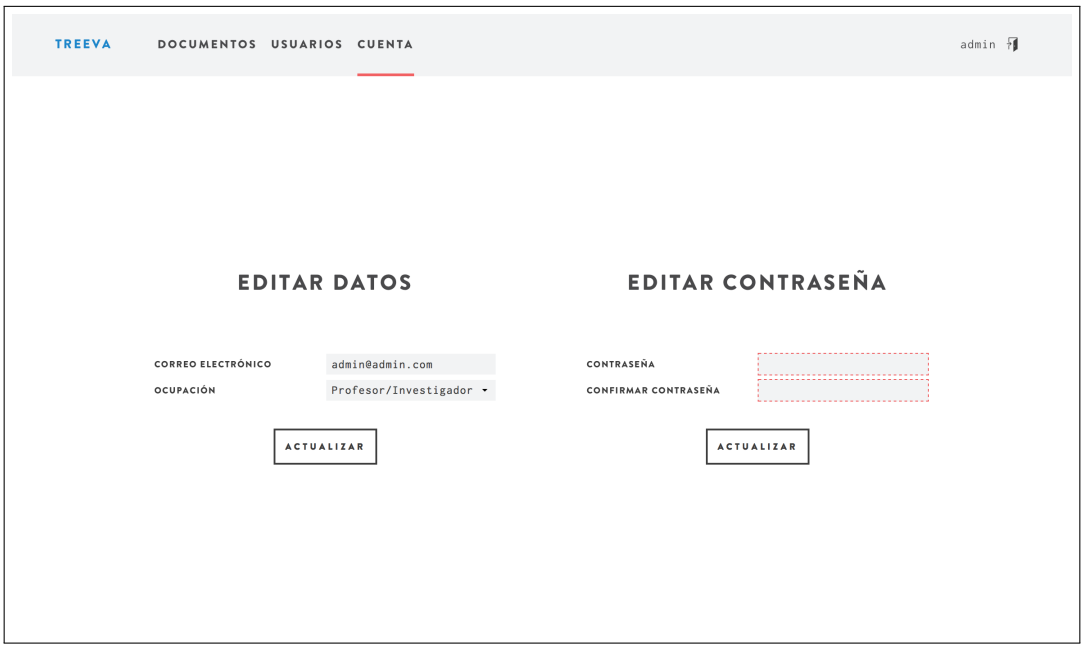


Figura 4.21: Página de Cuentas: editar

4.4.5. Sección Documentos

Se accede mediante la opción *Documentos* del menú de navegación y se encuentra disponible solo para los usuarios registrados. La página dispone de las siguientes operaciones:

Listar es la operación por defecto al ingresar a la página. Mediante una tabla se lista los documentos creados por el usuario de la sesión. En la Figura 4.22 se tiene la vista global de la sección, compuesto por la tabla de los registros, y la barra de opciones en la parte superior de la tabla, con botones que al ser presionados permiten añadir, editar y eliminar los elementos de la lista.

Nuevo permite añadir nuevos documentos en blanco. Mediante el botón *Nuevo* se agrega una nueva fila a la tabla, esta fila es diferente al resto pues en ella se debe agregar el título del documento para que sea haga efectiva la persistencia. En la Figura 4.23

se muestra un ejemplo de esta nueva fila por agregar.

Existe otra alternativa que permite crear un documento en blanco sin guardar, pues redirige a la página del editor de árboles con la hoja en blanco. Para optar por esta opción se presionado el otro botón *Nuevo* ubicado encima de la barra de opciones.

Editar titulo permite actualizar el título de un documento seleccionado en la tabla. Mediante el botón *Editar* la columna nombre de la fila seleccionada queda editable para solicitar el cambio. En la Figura 4.24 se muestra un ejemplo de una fila editable.

Eliminar permite eliminar uno o varios documentos simultáneamente seleccionados en la tabla. Mediante el botón *Eliminar* se remueve los documentos, sin embargo, antes de ejecutar la acción se solicita la confirmación del usuario para proceder. En la Figura 4.20 se muestra la interfaz de confirmación.

Abrir documento permite desplegar el contenido del documento seleccionado en el editor de árboles. Para abrir un documento se presiona sobre ella en la lista, inmediatamente se redirecciona a la página del editor con el dibujo solicitado.

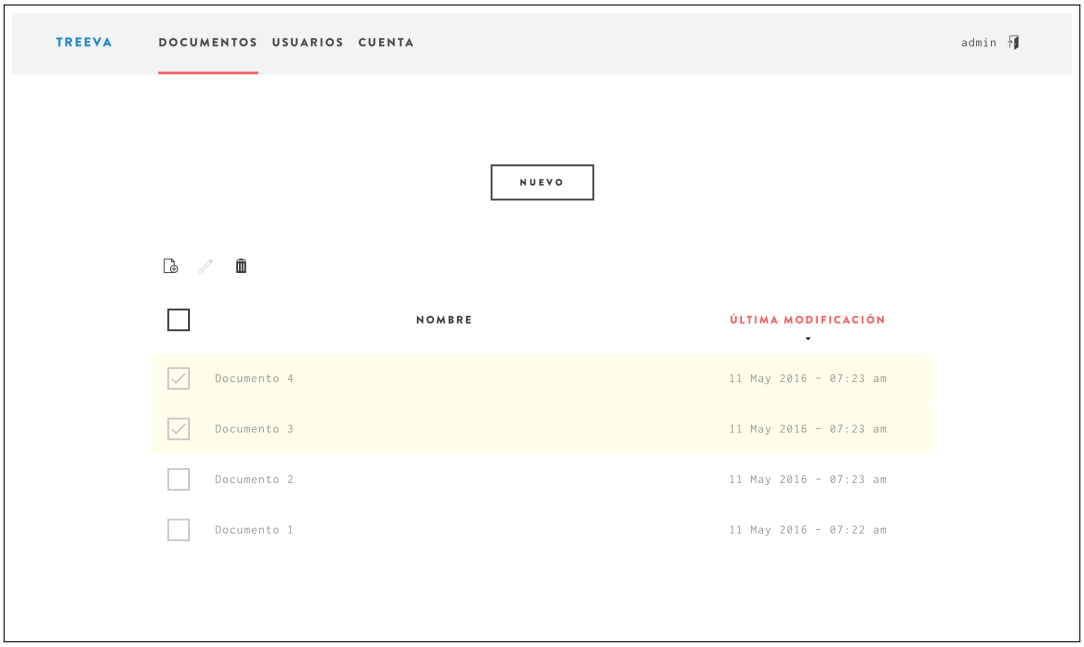


Figura 4.22: Página de Documentos: listar

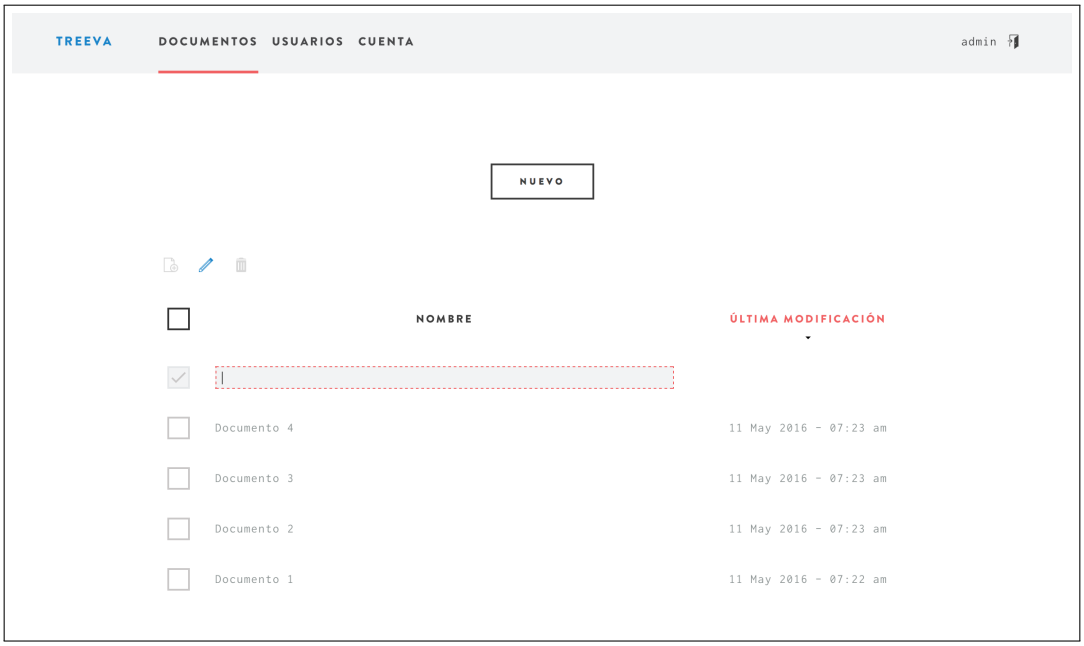


Figura 4.23: Página de Documentos: nuevo

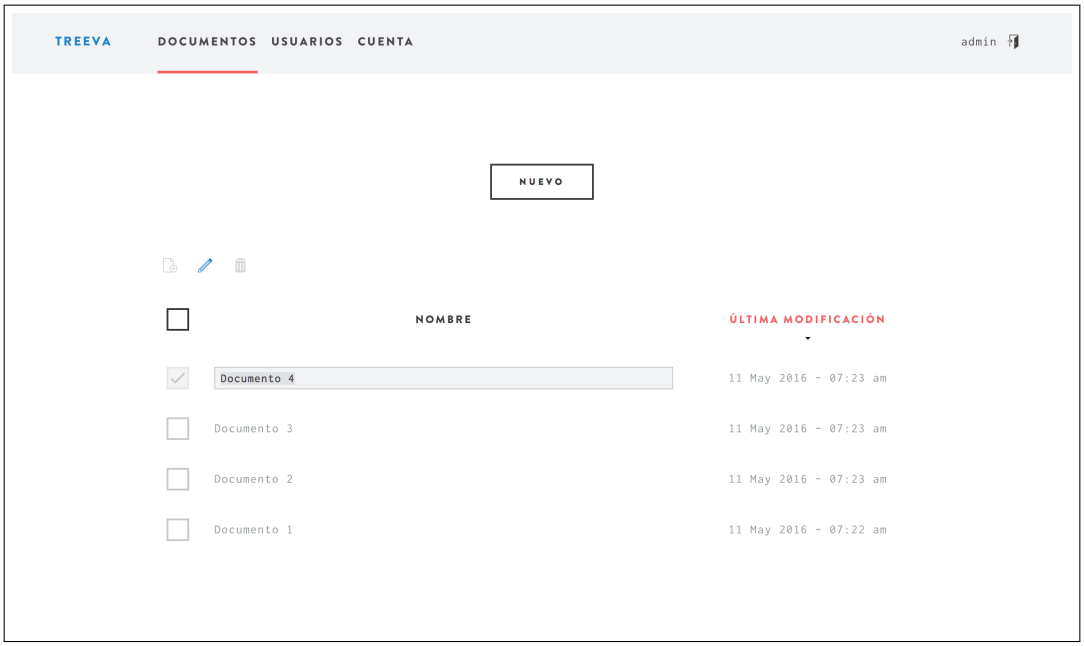


Figura 4.24: Página de Documentos: editar

4.4.6. Sección Editor

Es el núcleo de la aplicación, pues es donde se encuentra el espacio de trabajo para la edición de árboles. Disponible para todos los usuarios que ingresan a la aplicación, para no autenticados se accede desde la página de inicio, en caso contrario, desde la sección *Documentos*.

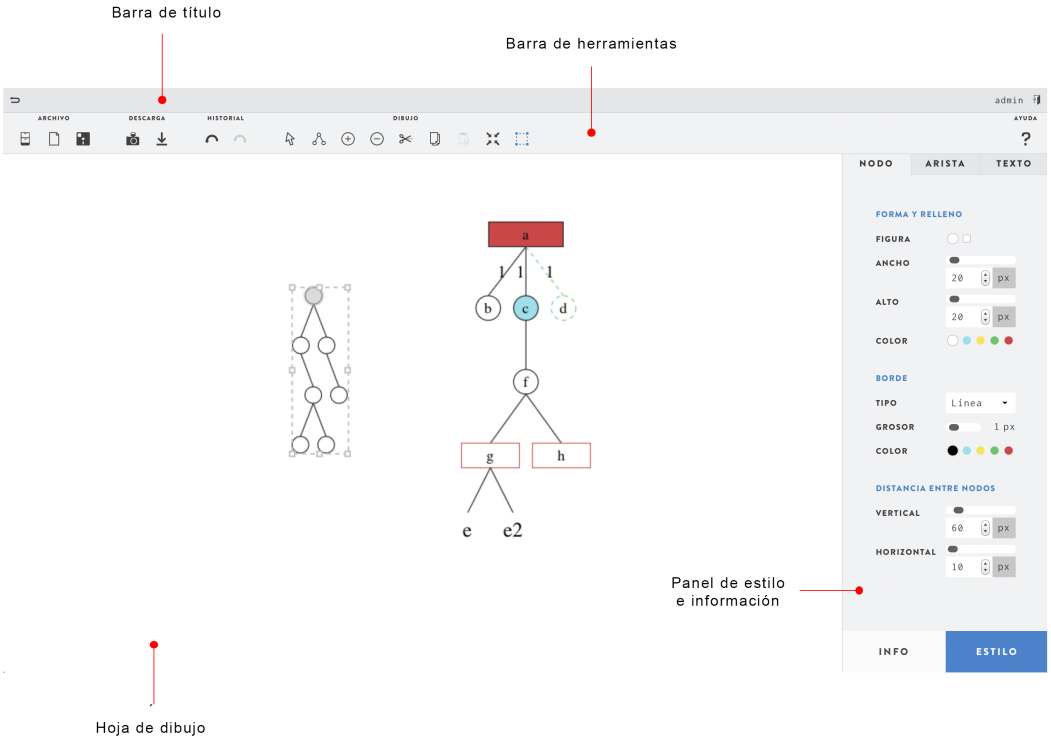


Figura 4.25: Página de Editor de árboles

El editor está formado por las siguientes áreas:

Barra de título

Ubicado en el tope superior de la sección. A la izquierda el botón *Retroceso* que ubica al usuario a la página anterior, en el centro el título del documento (en caso de estar guardado) y a la derecha el *cierre de sesión* para usuarios autenticados.

Barra de herramientas

Ubicado debajo de la barra de título. Esta barra está divida en secciones que agrupa los botones de acuerdo a su funcionalidad, y a su vez, algunos de estos botones pueden desplegar subopciones. La mayoría de estas funciones también pueden ser efectuadas mediante **atajos de teclado**.

No todos los comandos de la barra están siempre disponibles. Si un comando está deshabilitado en un momento dado, se mostrará de un color más opaco o claro. Depen-

diendo del estado de los árboles y de la selección de nodos, los comandos pueden activarse y desactivarse de la barra para indicar que no es una operación válida. Por ejemplo, el botón *Eliminar* está deshabilitado cuando no se ha seleccionado algún nodo.

- Archivo

  Abrir fichero despliega el explorador de proyectos, un panel del lado izquierdo que lista los documentos guardados, como se muestra en la Figura 4.26. Este panel contiene las opciones de agregar, editar título y eliminar documento como en la sección *Documentos*.

  Nueva hoja limpia la hoja de dibujo actual.

  Guardar Guarda el documento actual.

- Descarga

  Captura permite exportar una área seleccionado de la hoja de dibujo a PNG. En la Figura 4.27 se muestra una selección de captura.

  Descarga permite exportar un árbol seleccionado a PNG o LaTeX.

- Historial

  Deshacer deshace una acción sobre los árboles.

  Rehacer rehace una acción sobre los árboles.

- Dibujo

  Seleccionar permite seleccionar los nodos de un árbol.

  Nuevo árbol añadir una nueva raíz n-ario o binario.

  Añadir agregar hijos a los nodos seleccionados.

  Remover eliminar a los nodos seleccionados.

  Cortar corta un nodo seleccionado.

  Copiar copia un nodo seleccionado.

  Pegar pegar sobre los nodos seleccionados.

  Colapsar contrae o expande los descendientes de los nodos seleccionados.

  Escalar activa o desactiva el modo de escalamiento.

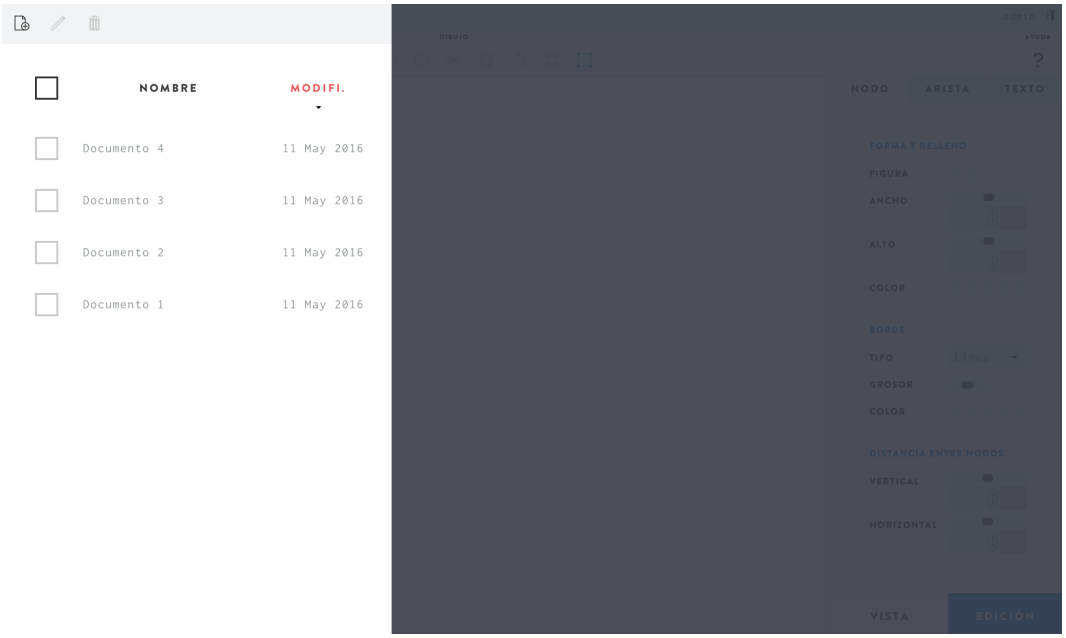


Figura 4.26: Página de Documentos: menú lateral derecho

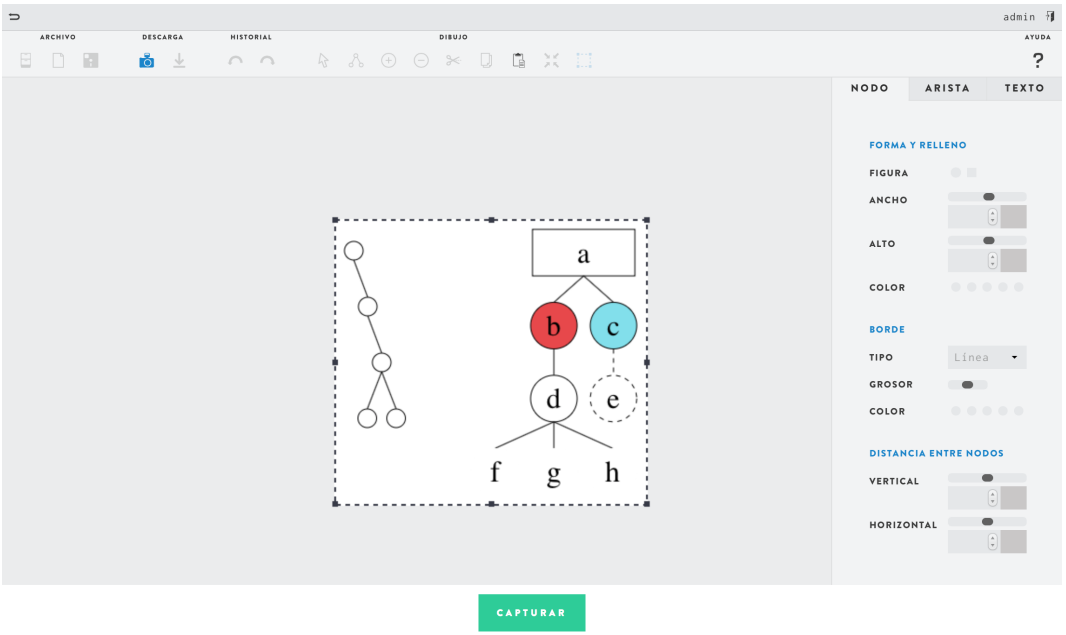


Figura 4.27: Página de Documentos: menú lateral derecho

Panel de estilo e información

Ubicado del lado derecho de la sección. Compuesto por dos pestañas:

- Información: contiene información estructural del árbol, como su tipo o grado.
- Estilo

  Forma establece la forma de los nodos seleccionados.
  Tamaño establece la anchura y la altura de los nodos seleccionados.
  Color establece el color de relleno y línea de los nodos seleccionados y sus aristas.
  Línea establece el tipo de línea de los nodos seleccionados y sus aristas.
  Distancia establece la separación entre nodos de los árboles seleccionados.
  Texto permite nombrar los nodos seleccionados y sus aristas.

Hoja de dibujo

Este es el espacio de dibujo del editor, por lo tanto ocupa la mayor parte de la página.

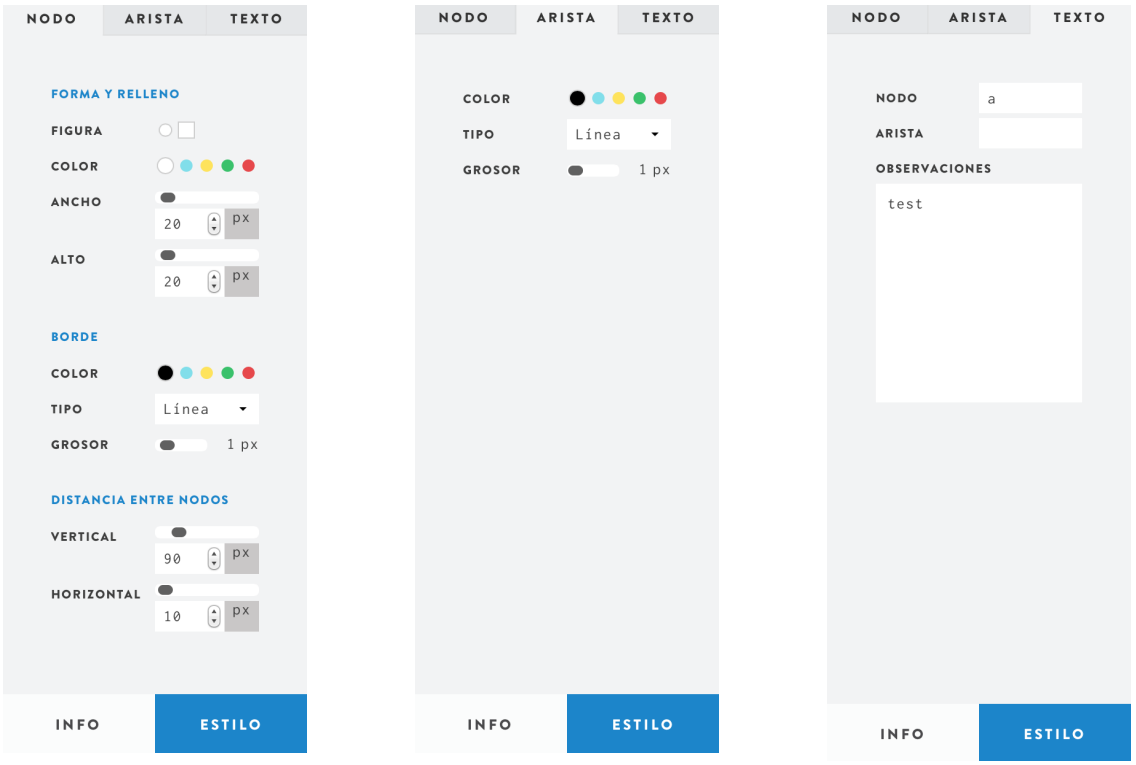


Figura 4.28: Página de Documentos: menú lateral derecho

Figura 4.29: Página de Documentos: menú lateral derecho

Capítulo 5

Pruebas y Resultados

En este capítulo se expone la potencialidad del editor mediante la descripción del proceso de creación de un árbol empleando la colección de recursos que dispone la aplicación.

Para validar la efectividad de este proceso, el editor pasa por una serie de pruebas cualitativas obtenidas mediante la realización de encuestas a un grupo voluntario de profesores y estudiantes.

Por último se identifican las diferencias y se realiza un análisis comparativo entre la solución implementada con el resto de las aplicaciones disponibles, ésto con el objetivo de comprobar el aporte del trabajo realizado cubriendo sustancialmente las necesidades que no han sido atendidas, al menos por las aplicaciones encontradas durante la investigación.

5.1. Resultados en ejecución

1. **Raíz**: en la Figura 5.1 se muestra el nodo raíz de un árbol n-ario (derecha de la figura) que se crea mediante el botón *Nuevo n-ario* de la barra de herramientas (izquierda de la figura).

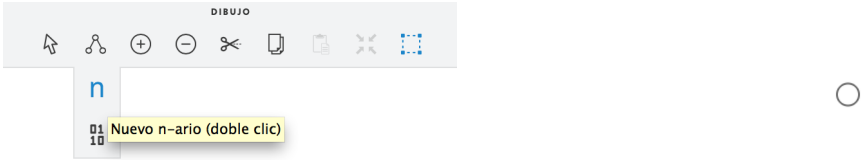


Figura 5.1: Nuevo árbol

2. **Descendientes**: en la Figura 5.2 se observa la raíz anterior con descendientes (derecha de la figura), los nodos hijos son agregados con el uso de los botones *Agregar hijo*, *Agregar izquierdo* y *Agregar derecho* de la barra de herramientas (izquierda de la figura). Otros botones de la barra de herramienta pueden intervenir en el proceso, como *Eliminar*, *Cortar*, *Copiar* y *Pegar* nodo.

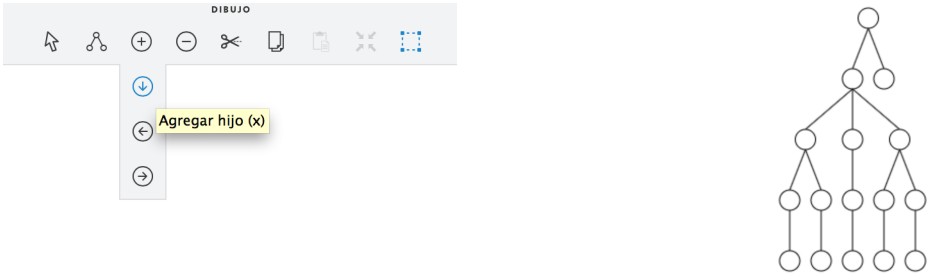


Figura 5.2: Árbol con descendientes

3. **Estilo de forma y relleno de nodos**: en la Figura 5.3 se empieza a mostrar el árbol con diferentes estilos a los establecidos por defecto en cada nodo. En esta figura se muestra modificaciones en los valores de forma, color de relleno y tamaño en algunos de sus nodos (derecha de la figura). Para estas configuraciones se utiliza las herramientas del panel de estilo ubicado a la derecha en la pestaña *Nodo* sección *Forma y Relleno* (izquierda de la figura).

Figura 5.3: Árbol con estilo en forma y relleno

4. **Estilo de línea de nodos**: en la Figura 5.4 se muestra el árbol con modificaciones en los valores de color, tipo y tamaño en la línea de algunos de sus nodos (derecha de la figura). Para estas configuraciones se utiliza las herramientas del panel de estilo ubicado a la derecha en la pestaña *Nodo* sección *Borde* (izquierda de la figura).

Figura 5.4: Árbol con estilo de borde

5. **Estilo de linea de aristas**: en la Figura 5.5 se muestra el árbol con modificaciones en los valores de color, tipo y tamaño de línea de algunas las aristas de sus nodos (derecha de la figura). Para estas configuraciones se utiliza las herramientas del panel de estilo ubicado a la derecha en la pestaña *Arista* (izquierda de la figura).

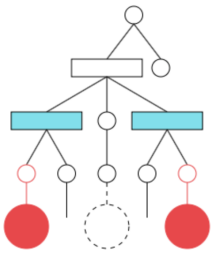
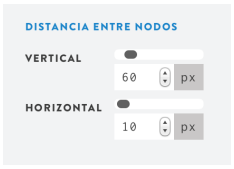


Figura 5.5: Árbol con estilo en aristas

6. **Distancia entre nodos**: en la Figura 5.6 se muestra el árbol con modificaciones en los valores de distancia de separación vertical entre los nodos (derecha de la figura). Para estas configuraciones se utiliza las herramientas del panel de estilo ubicado a la derecha en la pestaña *Nodo* sección *Distancia entre Nodos* (izquierda de la figura).



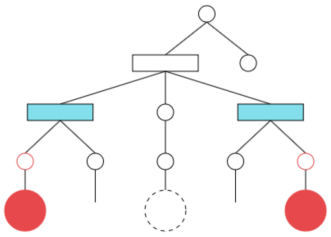


Figura 5.6: Árbol con separación

7. **Valores de nodos y aristas**: en la Figura 5.7 se muestra el árbol con valores en algunos de sus nodos y aristas (derecha de la figura). Se utiliza las herramientas del panel de estilo ubicado a la derecha en la pestaña *Texto* para nombrar nodos y aristas (izquierda de la figura).

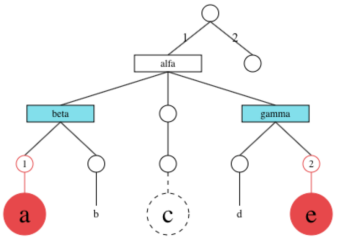


Figura 5.7: Árbol valorado

8. Para dibujar un bosque como el de la Figura 5.8 se repite el proceso desde el primer paso tantas veces como árboles se requiera.

En las siguientes figuras 5.9 y 5.10 se observan distintos ejemplos de árboles construidos con el editor.

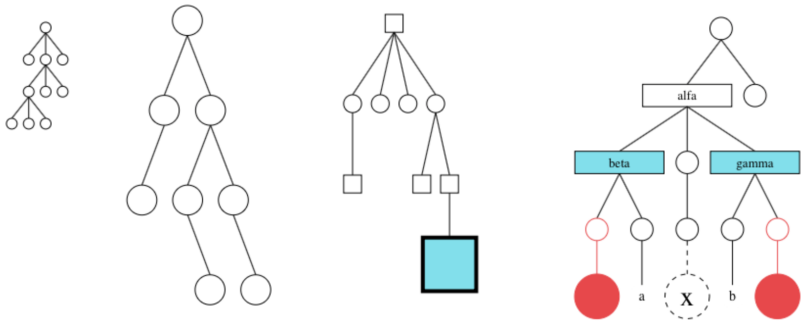


Figura 5.8: Ejemplo de resultado 3

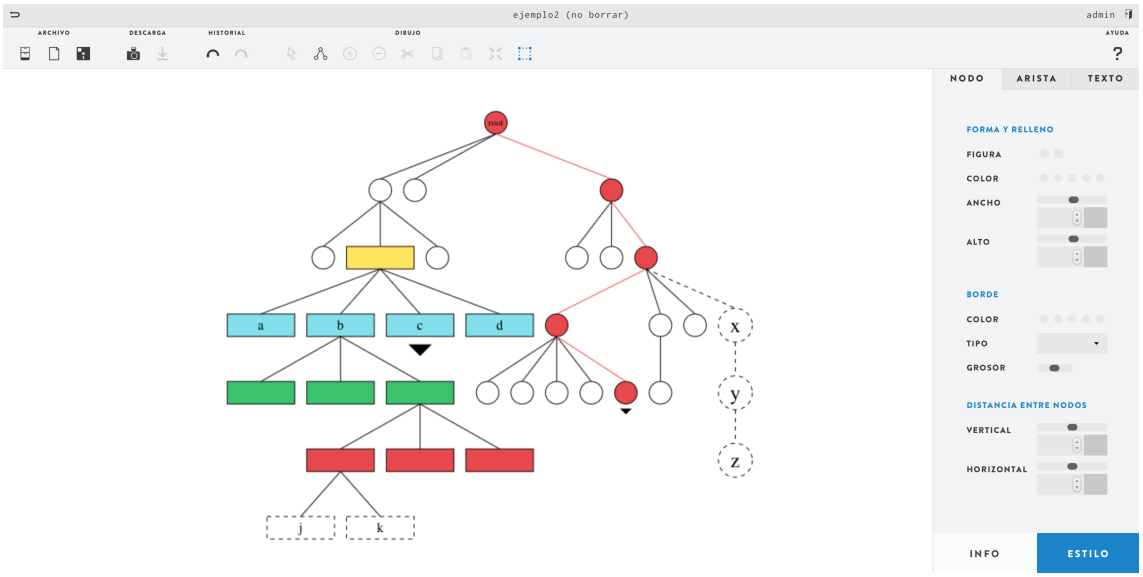


Figura 5.9: Ejemplo de resultado 1

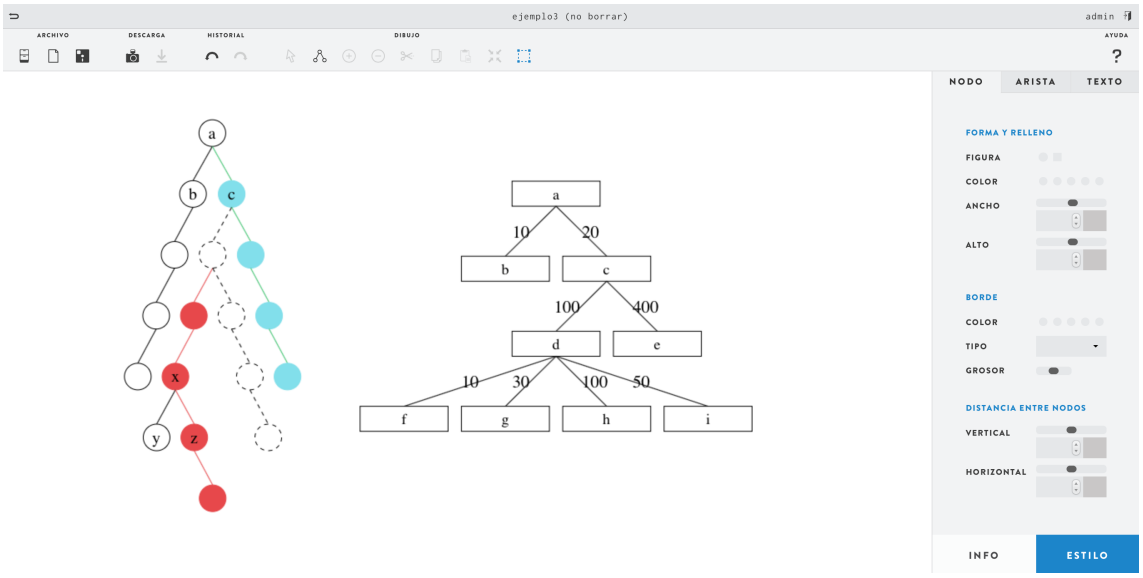


Figura 5.10: Ejemplo de resultado 2

5.2. Pruebas y resultados cualitativos

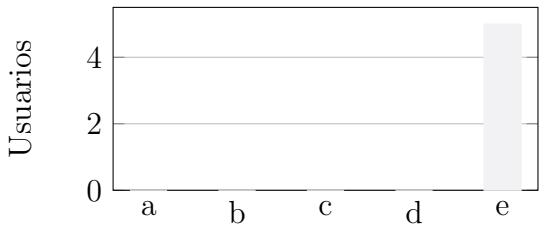
Las pruebas cualitativas se basan en medir la apreciación que tienen los usuarios acerca del sistema. Para estas pruebas se realizaron encuestas a un grupo de profesores y estudiantes de ciencias de la computación que debieron responder basado en la experiencia obtenida durante la interacción con el sistema.

Este grupo de usuarios tenían la tarea de dibujar los árboles de la Figura 5.8, desde el más sencillo (izquierda) al más complejo (derecha).

A continuación se lista los criterios evaluados con sus respectivos resultados:

1. Definición de estilos.

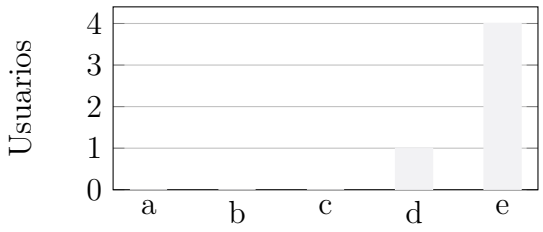
   a) pésimo
   b) mal
   c) indiferente
   d) bien
   e) excelente



   Promedio 5

2. Distribución de los elementos

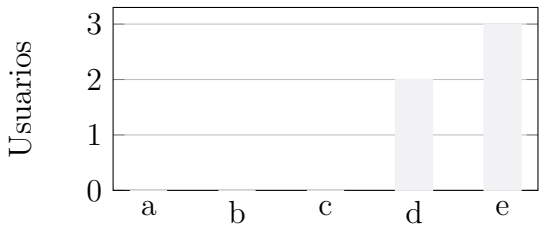
   a) desordenada
   b) un poco desordenada
   c) regular
   d) casi ordenada
   e) ordenada



   Promedio 4.8

3. Formas gráficas y lenguaje

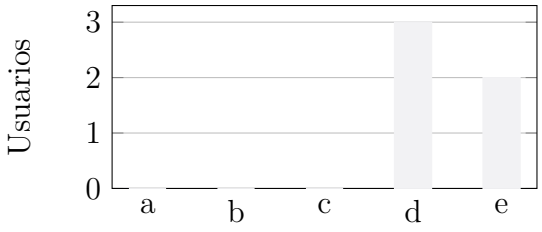
   a) pésimo
   b) mal
   c) indiferente
   d) bien
   e) excelente



   Promedio 4.6

4. Disponibilidad de recursos para definir la estructura de árboles

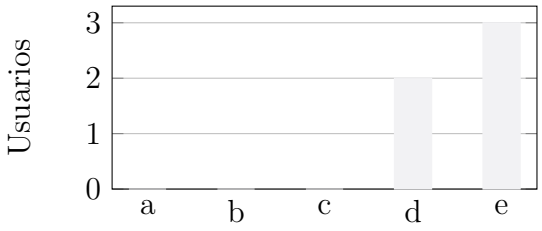
   a) insuficiente
   b) casi insuficiente
   c) básico
   d) casi suficiente
   e) completo



   Promedio 4.4

5. Disponibilidad de recursos para definir el estilo de árboles

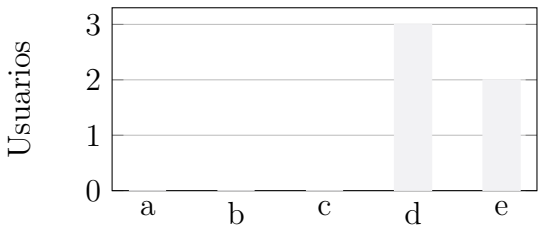
   a) insuficiente
   b) casi insuficiente
   c) básico
   d) casi suficiente
   e) completo



   Promedio 4.6

6. Complejidad de las herramientas para definir la estructura de árboles

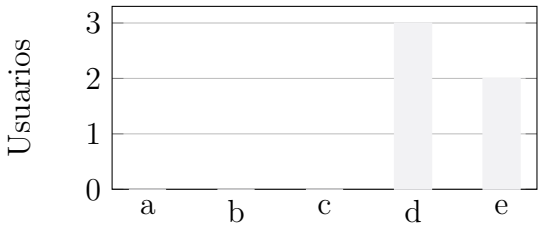
   a) frustrante
   b) difícil
   c) normal
   d) fácil
   e) muy fácil



   Promedio 4.4

7. Complejidad de las herramientas para definir el estilo de árboles

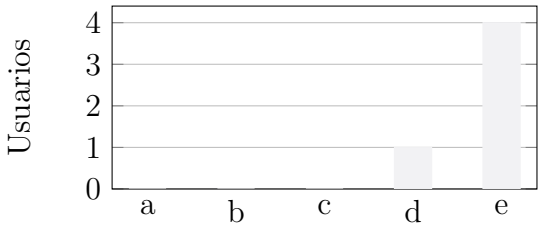
   a) frustrante
   b) difícil
   c) normal
   d) fácil
   e) muy fácil



   Promedio 4.4

8. ¿Está de acuerdo con los valores de estilo sugeridos por el sistema para los nodos y las aristas?

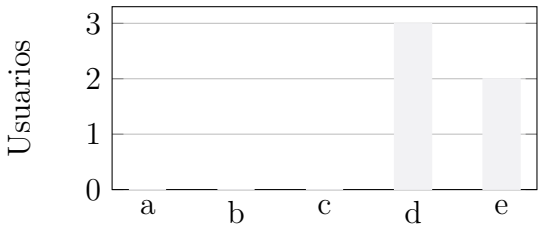
   a) nunca
   b) pocas veces
   c) a veces
   d) casi siempre
   e) siempre



   Promedio 4.8

9. Respuestas inesperadas en la ejecución de acciones

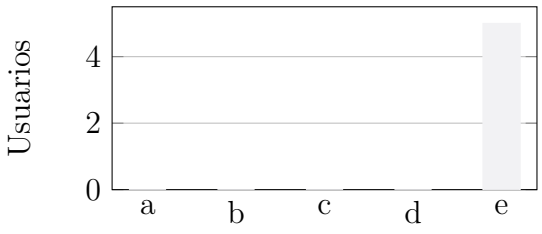
   a) siempre
   b) casi siempre
   c) a veces
   d) pocas veces
   e) nunca



   Promedio 4.4

10. Velocidad de respuesta en el tiempo preciso

    a) nunca
    b) pocas veces
    c) a veces
    d) casi siempre
    e) siempre



    Promedio 5

5.2.1. Análisis de los resultados del estudio cualitativo

Se analizan los resultados de los criterios evaluados en cuatro tópicos:

Presentación en esta categoría se evalúa la apariencia de la interfaz gráfica de usuario (GUI por su nombre en inglés, Graphical User Interface). Se considera la definición de estilos, como la paleta de colores y los formatos de texto, la distribución de los elementos, las formas gráficas o imágenes, como botones e íconos y el lenguaje. La GUI es la carta de presentación del producto, que en muchas ocasiones, es la que determina si la aplicación será o no utilizada para resolver los problemas para los cuales fue diseñada.

Los resultados de las encuestas indican que **la GUI cumple satisfactoriamente con los estándares de diseño para ofrecer un entorno visual sencillo y amigable**, lo que contribuye a la realización de las tareas de manera rápida y eficaz.

Disponibilidad de recursos se refiere al conjunto de herramientas que dispone el editor para la construcción de árboles que cumplan con las expectativas de los usuarios.

Según el resultado de las encuestas, **tanto para la construcción del esqueleto de los árboles como la definición de estilos, el editor dispone de una serie suficiente funcionalidades** que permiten cubrir las exigencias de los usuarios, no está totalmente completa pero tampoco excede los recursos al punto de abarcar funcionalidades innecesarias.

Complejidad del manejo de las herramientas se evalúa la interfaz desde el punto de vista de usabilidad del editor, esto se refiere a la complejidad del proceso con la que el usuario puede manejar las herramientas para la construcción de los árboles. Los criterios de presentación y disponibilidad de recursos, anteriormente mencionados, colaboran con el grado de complejidad del manejo de las herramientas; una buena presentación ayuda a recrear un entorno de trabajo sencillo y amigable, y una gama completa de recursos permite cubrir las peticiones de los usuarios en menor cantidad de pasos.

La apreciación obtenida por los usuarios señala que **el proceso de edición tanto para la construcción del esqueleto de los árboles como la definición de estilos, pueden ser operados con facilidad**, no la óptima, pero suficiente para deducir que la solución implementada cumple satisfactoria con los criterios de usabilidad, aquella característica que hace que la aplicación sea fácil de utilizar y fácil de aprender.

Confiabilidad en la respuesta se refiere al comportamiento del sistema de acuerdo a lo que se espera de él, es decir, la ejecución de las acciones deseadas sin errores o respuestas inesperados en el tiempo preciso.

Los resultados de las encuestas sugieren que **la solución es efectiva, hace lo que se espera que haga, y es eficiente, realiza las tareas en el tiempo necesario y sin errores**.

5.3. Comparaciones

Las siguientes tablas muestran el resultado de las comparaciones entre el editor de la solución (conocido como Treeva) y los editores web investigados, más adelante se realiza el análisis de estos resultados.

En la Tabla 5.1 se puede observar que **tipos de elementos HTML** usa cada uno de los editores para dibujar sus árboles. La siguiente Tabla 5.2 se muestra un resumen de alto nivel entre Canvas y SVG [15].

La Tabla 5.3 lista las comparaciones considerando las **herramientas de dibujo** que disponen.

Existen otras herramientas que no están relacionadas directamente con el dibujo pero que ofrecen un **valor agregado** en la percepción del usuario a las aplicaciones, la Tabla 5.4 compara estas herramientas.

	JsTree Graph	**Tree Graph**	**Tree View**	**Treeva**
Elemento HTML de dibujo	div	canvas	svg	canvas

Cuadro 5.1: Comparación con editores existentes: elemento HTML

Canvas	**SVG**
Basados en píxeles: no admiten escalabilidad; un grafo en canvas perdería fidelidad rápidamente cuando se usa el zoom.	**Basados en su forma**: son escalables.
Elemento HTML único	Múltiples elementos gráficos, que forman parte de DOM
Modificado mediante script solamente	Modificado mediante script y CSS
La interacción de usuario/modelo de eventos es pormenorizada (x,y): para la detección de eventos sobre el dibujo el programador debe traducir una coordenada `mouseX`, `mouseY` en el elemento único de la etiqueta `<canvas>` y después redirigir ese comando a una forma que se encuentre en una estructura con memoria.	**La interacción de usuario/modelo de eventos es resumida (rect, ruta)**: SVG está integrado en el documento con elementos, comportándose de manera similar a un `<div>`, estos elementos como objetos DOM responden a los eventos.
El rendimiento es mejor con una superficie menor, un número más grande de objetos (mayor 1000) o ambos.	El rendimiento es mejor con un número menor de objetos (menor 1000), una superficie más grande o ambos.

Cuadro 5.2: Comparación entre Canvas y SVG

Las mediciones de rendimiento de la Tabla 5.2 no son necesariamente precisas y, sin dudas, pueden cambiar según la implementación y la plataforma, ya sea si se usan gráficos completamente acelerados por hardware o no, y la velocidad del motor de JavaScript [15].

	JsTree Graph	Tree Graph	Tree View	Treeva
Edición en tiempo real	✗	✗	✓	✓
Añadir	✗	✓	✓	✓
Remover	✗	✗	✓	✓
Cortar	✗	✗	✗	✓
Copiar	✗	✗	✗	✓
Pegar	✗	✗	✗	✓
Colapsar descendientes	✓	✓	✗	✓
Escalar	✗	✗	✗	✓
Trasladar	✓	✗	✗	✓
Forma	✗	✗	✗	✓
Color de nodo	✗	✓	✓	✓
Color de borde nodo	✗	✗	✓	✓
Tipo de línea de borde nodo	✗	✗	✗	✓
Grosor de borde nodo	✗	✗	✓	✓
Color de arista	✗	✗	✗	✓
Tipo de línea arista	✗	✗	✓	✓
Grosor de arista	✗	✗	✓	✓
Distancia entre nodos	✗	✗	✓	✓
Valor nodo	✓	✓	✓	✓
Valor arista	✗	✗	✓	✓
Valor adicional	✓	✗	✗	✓
Automatización de posiciones	✓	✓	✓	✓
Orientación del árbol	✓	✗	✗	✗
Árbol binario	✗	✗	✓	✓
Múltiples árboles	✗	✗	✓	✓

Cuadro 5.3: Comparación con editores existentes: edición

	JsTree Graph	Tree Graph	Tree View	Treeva
Información de la estructura	✗	✓	✗	✓
Historial	✗	✗	✗	✓
Exportar	✗	✗	✓	✓
Guardar	✗	✗	✗	✓
Scroll	✗	✗	✗	✗
Zoom	✓	✗	✗	✗
Animación	✓	✗	✗	✗

Cuadro 5.4: Comparación con editores existentes: valor agregado

5.3.1. Análisis de los resultados del estudio comparativo

Para crear formas gráficas en HTML se pueden utilizar los elementos div, svg y canvas, cada uno de ellos ofrece prestaciones diferentes. Para dibujar árboles la opción menos recomendada es utilizar elementos div, pues cuando se necesita trabajos más elaborados, esta tarea se vuelve engorrosa. Por ejemplo, hacer un círculo con el uso de la etiqueta `<div>` sólo es posible redondeando las esquinas del elemento utilizando CSS, para formas más complejas se tendría que involucrar hasta cientos de diminutos div.

Las tecnologías de canvas y svg de HTML5 son una mejor alternativa para la construcción de árboles. Con ambos es posible dibujar prácticamente cualquier gráfico vectorial, los gráficos vectoriales son formas geométricas básicas (formas, puntos, líneas y polígonos) que se basan en los vectores para representar imágenes. A pesar de sus diferencias se puede lograr resultados casi idénticos, la decisión de cual es el significativamente mejor para la solución dependerá del punto de vista del programador y de lo que considere más apropiado para los requerimientos.

Basado en las comparaciones de las herramientas de dibujo entre los editores de la Tabla 5.1, se puede afirmar que la solución desarrollada incluye casi en su totalidad las funcionalidades del resto de los editores e incorpora muchas otras que son necesarias y que carecen los otros editores. Entre éstas se pueden destacar las funcionalidades clásicas como copiar, cortar, pegar y las transformaciones como escalar y trasladar. Estas operaciones generalmente están disponibles en cualquier buen editor de diagramas, de imágenes y de textos; probablemente los usuarios están familiarizados con éstos, lo que les permite tener un proceso más natural del manejo de la edición.

Además, la aplicación desarrollada permite la posibilidad de definir la apariencia de los nodos y de las aristas con mayor precisión que el resto de los editores, desde el color, el tamaño, el tipo de trazo, la asignación de valores y el ordenamiento automático de nodos y aristas que a diferencia de los otros permite personalizar la distancia de separación entre los nodos de un mismo nivel y de estos con otros niveles.

La diferencia más notable recae sobre la interfaz gráfica de usuario, en los demás editores la construcción del árbol dependen de una entrada de texto o una variable JSON que representa la traducción del éste en un formato que varía por aplicación, forzando al usuario a dominarlos o simplemente perderles el interés. A diferencia de éstos, el editor de la solución tiene un conjunto de herramientas dentro de la interfaz que ejecutan acciones sobre los elementos que han sido seleccionados sobre la imagen del canvas.

Según los resultados de la Tabla 5.4, la solución tiene una serie de funcionalidades adicionales que no están vinculadas directamente con el proceso de dibujo pero que le agregan mayor valor a la percepción y experiencia del usuario sobre la aplicación. Entre estas funcionalidades más importantes y que carecen los demás editores se puede mencionar el registro de historial de acciones sobre el dibujo que permite deshacer y rehacer cambios, la exportación en diferentes formatos para su anexión en otros documentos y la capacidad de salvaguardar los trabajos mediante la creación de cuentas de usuarios.

Capítulo 6

Conclusiones, Recomendaciones y Trabajos Futuros

Las conclusiones tras la investigación teórica, el desarrollo de la solución y los resultados obtenidos se centran en este capítulo. Adicionalmente se sugiere las recomendaciones y posibles trabajos a futuro para la continuación de la investigación y desarrollo del área.

6.1. Conclusiones

HTML se mantiene evolucionando para proporcionar mejoras y más ricos gráficos estandarizados, de este modo, los desarrolladores tienen la oportunidad de crear aplicaciones web ricas en gráficos usando tecnologías basadas en estándares evitando la instalación de complementos o códigos específicos del explorador.

La aplicación web desarrollada implementa un algoritmo que permite la creación de árboles genéricos y binarios que estima automáticamente el estilo y la separación vertical y horizontal entre los nodos del árbol de manera ordenada. Estos valores sugeridos por el sistema pueden ser modificados a conveniencia del usuario.

El editor de árboles de la solución, a diferencia del resto de los editores web disponibles encontrados, maneja una gran variedad de herramientas que permite al usuario mayor control y la posibilidad de salvaguardar su trabajo.

Los resultados basados en la apreciación de un grupo de usuarios de ciencias de la computación han probado que el cálculo predeterminado de los valores de estilo y posicionamiento ordenado de los nodos, sumado al conjunto de herramientas que brinda la interfaz gráfica de usuario facilita la construcción de árboles complejos en simples pasos minimizando el costo, el tiempo y el riesgo a errores.

La aplicación genera aportes en la comunidad docente y estudiantil en las materias que incluyen o sugieren en su plan de estudio la implementación de árboles para la resolución de problemas o notas de docencia.

6.2. Recomendaciones

En las notas de docencia de las materias dictadas en la Universidad Central de Venezuela:

Algoritmo y Estructura de Datos

Técnicas Avanzadas de Programación

la herramienta desarrollada puede ser sumamente útil para las notas de docencia de las materias o para explicar diversos tipos de algoritmos asociados a árboles.

Los estudiantes que cursan las materias mencionadas frecuentemente tienen que escribir algoritmos que se resuelven con la ayuda de árboles, usando la herramienta pueden analizar la solución de manera más práctica.

6.3. Trabajos Futuros

Algunos de los posibles objetivos que se plantean para continuar con la línea de desarrollo del sistema a partir de la investigación realizada son los siguientes:

- Orientación horizontal de los árboles, esto es, que los nodos descendientes de la raíz se posicionen a lo ancho de la hoja.
- Selección de múltiples nodos mediante *drag and drop*.
- Permitir hacer apuntes con áreas de textos o anotaciones a mano alzada en cualquier parte de la hoja de dibujo.
- Extensión de las opciones de estilo ampliando la paleta de colores, incluyendo nuevas formas y tipo de líneas para nodos y aristas, y permitir la personalización de fuente de letras.
- Barra de deslizamiento vertical y horizontal en la hoja de dibujo para aumentar el tamaño del espacio de trabajo.
- Capacidad de múltiple hojas de trabajo en diferentes pestañas en el mismo editor.
- Salvaguardado automático.
- Reconocimiento de evento offline. La aplicación podría detectar cuando está conectada y desconectada de Internet. Cuando se conoce que se está offline, las peticiones (como la de salvaguardado) se encolan al servidor para ser atendidas posteriormente, y cuando se detecta que se está online las aplicaciones se re-sincronizan con el servidor.
- Animación en los nodos para marcar la transición de una posición a otra, por ejemplo, la reubicación de los nodos al incorporarse una nueva en el árbol.

- Soporte en dispositivos móviles y tabletas. La detección de eventos es distinta de acuerdo al dispositivo y la interfaz debe acomodarse a los diferentes tamaño de pantalla.
- Envió de contraseña y nombre de usuario por correo electrónico.
- Permitir que el usuario se autentique y se registre después de haber dibujado sobre el editor para poder almacenar su trabajo (en caso de que el usuario haya optado primero por probar el editor en vez de iniciar sesión).
- Opción de ayuda que muestre una guía breve del uso del editor.

Bibliografía

[1] Adobe. *Aspectos básicos de las aplicaciones Web*. Accedido 12 de Marzo de 2016. URL: https://helpx.adobe.com/es/dreamweaver/using/web-applications.html.

[2] Word Wide Web Consortium. *Guía Breve de CSS*. Accedido 12 de Marzo de 2016. 2016. URL: http://www.w3c.es/Divulgacion/GuiasBreves/HojasEstilo.

[3] Word Wide Web Consortium. *HTML, The Web's Core Language*. Accedido 12 de Marzo de 2016. 2016. URL: https://www.w3.org/html/.

[4] CTAN. *CTAN: Package tikz-qtree*. Accedido 12 de Marzo de 2016. URL: https://www.ctan.org/pkg/tikz-qtree.

[5] Desarrolloweb. *Javascript a fondo*. Accedido 12 de Marzo de 2016. 2016. URL: http://www.desarrolloweb.com/javascript/#quees.

[6] Draw.io. *Flow Chart Maker and Online Diagram Software*. Accedido 12 de Marzo de 2016. URL: https://www.draw.io.

[7] Dia Diagram Editor. *Dia draws your structured diagrams: Free Windows, Mac OS X and Linux version of the popular open source program*. Accedido 12 de Marzo de 2016. URL: http://dia-installer.de.

[8] JS Tree Graph. *JS Tree Graph*. Accedido 12 de Marzo de 2016. URL: https://jstreegraph.codeplex.com.

[9] Tree Graph. *Tree Graph*. Accedido 12 de Marzo de 2016. URL: https://github.com/rodrigocfd/html5-tree-graph.

[10] R. Grimaldi. Addison- Wesley Iberoamericana, S.A., 1997.

[11] Icomparable. *¿Arquitectura n-Tier o Arquitectura n-Layer?* Accedido 12 de Marzo de 2016. 2008. URL: http://icomparable.blogspot.com/2008/10/arquitectura-n-tier-o-arquitectura-n.html.

[12] JavaScript. *JavaScript*. Accedido 12 de Marzo de 2016. 2016. URL: https://www.javascript.com.

[13] Sergio Luján Mora. Editorial Club Universitario.

[14] Microsoft. *Software profesional para diagramas — Microsoft Visio*. Accedido 12 de Marzo de 2016. URL: https://products.office.com/es-es/visio/flowchart-software.

[15] Microsoft. *SVG frente a Canvas: cómo elegir*. Accedido 12 de Marzo de 2016. 2016. URL: https://msdn.microsoft.com/es-es/library/gg193983(v=vs.85).aspx.

[16] MySQL. *MySQL :: MySQL 5.7 Reference Manual :: 1.3.1 What is MySQL?* Accedido 12 de Marzo de 2016. 2016. URL: https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html.

[17] Hector Navarro. *TreeView*. Accedido 12 de Marzo de 2016. URL: http://ccg.ciens.ucv.ve/~hector/treeview/.

[18] Mozilla Developer Network. *Clases*. Accedido 12 de Marzo de 2016. 2016. URL: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes.

[19] Mozilla Developer Network. *HTML5*. Accedido 12 de Marzo de 2016. 2016. URL: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5.

[20] Mozilla Developer Network. *Introducción a JavaScript orientado a objetos*. Accedido 12 de Marzo de 2016. 2016. URL: https://developer.mozilla.org/es/docs/Web/JavaScript/Introduccin_a_JavaScript_orientado_a_objetos.

[21] Mozilla Developer Network. *Javascript*. Accedido 12 de Marzo de 2016. 2016. URL: https://developer.mozilla.org/es/docs/Web/JavaScript.

[22] Mozilla Developer Network. *z-index*. Accedido 12 de Marzo de 2016. 2016. URL: https://developer.mozilla.org/es/docs/Web/CSS/z-index.

[23] Overleaf. *Overleaf: Real-time Collaborative Writing and Publishing Tools with Integrated PDF Preview*. Accedido 12 de Marzo de 2016. URL: https://www.overleaf.com.

[24] Cake PHP. *Entendiendo el Modelo - Vista - Controlador — documentación de CakePHP Cookbook - 2.x*. Accedido 12 de Marzo de 2016. 2016. URL: http://book.cakephp.org/2.0/es/cakephp-overview/understanding-model-view-controller.html.

[25] Ruby on Rails. *Action View*. Accedido 12 de Marzo de 2016. 2016. URL: http://guides.rubyonrails.org/action_view_overview.html.

[26] Ruby on Rails. *El desarrollo web que no molesta*. Accedido 12 de Marzo de 2016. 2016. URL: http://www.rubyonrails.org.es.

[27] Ruby on Rails. *Getting Started with Rails — Ruby on Rails Guides*. Accedido 12 de Marzo de 2016. 2016. URL: http://guides.rubyonrails.org/getting_started.html.

[28] Sharelatex. *ShareLaTeX, the Online LaTeX Editor*. Accedido 12 de Marzo de 2016. URL: https://es.sharelatex.com.

[29] Sharelatex. *The TeX family tree: LaTeX, pdfTeX, XeTeX, LuaTeX and ConTeXt - ShareLaTeX, Editor de LaTeX online*. Accedido 12 de Marzo de 2016. URL: https://es.sharelatex.com/blog/2012/12/01/the-tex-family-tree-latex-pdftex-xelatex-luatex-context.html.

[30] SmartDraw. *SmartDraw is the Smartest Way to Draw Anything*. Accedido 12 de Marzo de 2016. URL: https://www.smartdraw.com/.

[31] TeXlipse. *TeXlipse homepage - LaTeX for Eclipse*. Accedido 12 de Marzo de 2016. URL: http://texlipse.sourceforge.net/index.php.

[32] Texmaker. *Texmaker (free cross-platform latex editor)*. Accedido 12 de Marzo de 2016. URL: http://www.xm1math.net/texmaker.

[33] W3techs. *Extensive and reliable web technology surveys*. Accedido 12 de Marzo de 2016. URL: http://w3techs.com.

[34] WHATWG. *Embedded content*. Accedido 12 de Marzo de 2016. 2016. URL: https://html.spec.whatwg.org/multipage/dom.html#embedded-content-category.

[35] WHATWG. *FAQ*. Accedido 12 de Marzo de 2016. 2016. URL: https://wiki.whatwg.org/wiki/FAQ#What_is_HTML5.

[36] WHATWG. *The canvas element*. Accedido 12 de Marzo de 2016. 2016. URL: https://html.spec.whatwg.org/multipage/scripting.html#the-canvas-element.

[37] WHATWG. *WHATWG - The 2D rendering context*. Accedido 12 de Marzo de 2016. 2016. URL: https://html.spec.whatwg.org/multipage/scripting.html#canvasrenderingcontext2d.