Universidad Autónoma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación

# “Implementación de una interfaz gráfica interactiva para algoritmos de ordenamiento y estructuras tipo árbol”

Autor:

Mares Martínez Juan Carlos – 204203725

Asesores:

Dr. Risto Fermín Rangel Kuoppa

Dr. Francisco Javier Zaragoza Martínez

**Departamento de Sistemas**

# Resumen

Durante el transcurso de la carrera de ingeniería en computación los algoritmos de ordenamiento y las estructuras tipo árbol son una parte indispensable en el desarrollo académico del alumno y una parte fundamental es que el alumno entienda de forma clara la implementación y el funcionamiento de éstos. Debido a que muchos de estos algoritmos (de ordenamiento y estructuras tipo árbol) son recursivos y los demás contienen dos o más ciclos anidados su implementación y su funcionamiento pueden ser no muy claros o evidentes en el momento en que el alumno los está estudiando y analizando. Teniendo esto en cuenta se pensó en desarrollar una herramienta que le permitiera a los alumnos entender de una forma clara el funcionamiento de los algoritmos de ordenamiento y las estructuras tipo árbol.

El presente documento toma como piedra angular el funcionamiento de los algoritmos de ordenamiento y de las estructuras tipo árbol, dejando de lado los pormenores de su implementación, y describe el desarrollo e implementación de una interfaz gráfica de usuario que le permitirá al alumno entender de una forma gráfica e interactiva el funcionamiento de los mismos, lo cual se logrará mediante el modelado en 3D de algunos de los algoritmos de ordenamiento y estructuras tipo árbol más importantes y a los cuales se enfrentará de forma constante durante el desarrollo de la carrera de ingeniería en computación.

# Índice de contenido

## Lista de figuras

## Lista de tablas

## Objetivo General

Implementar un programa con interfaz gráfica en la que los usuarios puedan interactuar con diferentes estructuras tipo árbol y algoritmos de ordenamiento de un modo sencillo e intuitivo, para así facilitar el estudio y comprensión de las operaciones y comportamiento de los mismos.

## Objetivos Particulares

1. Seleccionar los algoritmos para la implementación de las estructuras tipo árbol y algoritmos de ordenamiento.

2. Elaborar los módulos para la manipulación lógica de las estructuras tipo árbol y algoritmos de ordenamiento.

3. Crear los módulos para el almacenamiento y recuperación de datos de las estructuras tipo árbol y algoritmos de ordenamiento.

4. Construir los módulos para la manipulación gráfica de las estructuras tipo árbol e interacción con los algoritmos de ordenamiento.

5. Desarrollar los módulos que dibujen en pantalla las estructuras de datos y permitan manipular los parámetros de los algoritmos de ordenamiento.

6. Desarrollar los módulos de opciones para especificar los parámetros de las estructuras tipo árbol y algoritmos de ordenamiento.

7. Elaborar un módulo de ayuda que ofrezca información sobre las operaciones y comportamiento de las estructuras tipo árbol y algoritmos de ordenamiento.

8. Integrar todos los módulos en una interfaz gráfica de usuario.

9. Elaborar los manuales de usuario, instalación y programador como son: diagramas de clases, estados y flujo.

# Antecedentes

En Internet se puede encontrar información acerca de estructuras de datos y algoritmos de ordenamiento. Esta información está sostenida en textos dedicados al estudio de estas estructuras y algoritmos, pero esto sigue siendo de manera teórica. Además de esto existen aplicaciones basadas en ejecuciones animadas mediante *applets* [1] [2] [3] [4] programados en Java, en algunos sitios web se uso de los *applets* [1] [2] [3] [4] para mostrar de forma gráfica el comportamiento de algunos de estos algoritmos de ordenamiento, el uso de estas aplicaciones se ofrece como una alternativa para el estudio y la comprensión de dichos algoritmos de ordenamiento y estructuras de datos.

De manera informal, en algunos sitios web se puede tener acceso a videos de animaciones en 3D [5] [6] [7] [8] desarrolladas en OpenGL [9]. De alguna forma, estas animaciones muestran el comportamiento de algunas estructuras de datos y algunos algoritmos de ordenamiento, pero a diferencia de los applets [1] [2] [3] [4] mencionados anteriormente no se tiene acceso a la aplicación, por lo que no se tiene conocimiento de que la aplicación tenga implementadas opciones para la manipulación de datos. Así que, el uso de esta herramienta queda limitada a un caso particular.

En la Facultad de Informática de la Universidad Complutense de Madrid (UCM) [10] se ha implementado una herramienta llamada “Herramienta para el estudio de estructuras de datos y algoritmos” [11], A través de un campus virtual, los alumnos matriculados en la UCM ingresan al campus para utilizar la herramienta. Haciendo uso de animaciones en flash, se representan de manera gráfica las operaciones y los cambios que se efectúan en una estructura de datos. La herramienta permite generar al usuario sus propias entradas de datos y, si así lo considera el estudiante, puede resolver los test que ahí mismo se le presentan para medir el grado de comprensión que obtuvo, una vez utilizada la herramienta. No hay acceso a la herramienta, ya que se necesita estar matriculado en la UCM.

Un antecesor de este trabajo lo constituye una propuesta de Proyecto Terminal [12] de UAM-A que en esencia persigue el mismo objetivo: presentar una herramienta que facilite el estudio y la comprensión de las estructuras de datos. Este proyecto se propuso para su implementación en Java haciendo uso de *applets* y trabaja sólo con algunas estructuras de datos (colas, pilas, listas y grafos). El nuevo proyecto a implementar se pretende desarrollar en OpenGL, con lo cual se hará uso de animación 3D; las estructuras con las que trabajará son estructuras tipo árbol y algoritmos de ordenamiento.

En Internet, se encuentra una herramienta de licencia GPL, Data Structures & Algorithms 1.0 [13] que está desarrollada en OpenGL [9]. Se puede descargar su código fuente para ser compilado en Linux utilizando OpenGL [9], SDL [14] y SDL_tff [15]. También puede ejecutarse en Windows y OS X. Esta aplicación es algo cercano a lo que se pretende desarrollar en esta propuesta. Las animaciones que presenta DSA 1.0 no muestran muchas operaciones que se efectúan sobre las estructuras de datos, además del factor interactivo del que carece, ya que el usuario no puede definir la entrada de datos con la cual trabajará la aplicación con la estructura de datos o el algoritmo de ordenamiento. Con la implementación de este proyecto se busca mejorar la animación y agregar el factor interactivo, de los cuales

carece la herramienta DSA 1.0.

## Justificación

En la ingenierías en computación resulta importante desarrollar la habilidad para programar aplicaciones sencillas o sistemas muy complejos que resuelvan algún problema específico. Preferentemente, que se haga de la manera más eficiente y, para lograr esto, resulta necesario en la mayoría de las veces tener un conocimiento profundo sobre estructuras de datos, algoritmos de ordenamiento, métodos de programación, etc.

Así que el desarrollar una herramienta interactiva que facilite el estudio de las estructuras de datos y algoritmos de ordenamiento se vuelve un tema importante (para este trabajo a desarrollar). Como se sabe, el concepto de estructura de datos es abstracto, cuando se quiere implementar alguno de éstos resulta una tarea difícil si no se entiende el comportamiento de la estructura de datos que se quiere. Por esto, el principal objetivo de este proyecto es visualizar dicho comportamiento.

En principio, se pretende que la herramienta sea interactiva, es decir, que permita al usuario final la manipulación de las entradas de datos y que las animaciones reflejen el cambio en los parámetros del algoritmo. Así mismo, al detener o reanudar las operaciones a efectuar, las herramientas basadas en *applets* [1] [2] [3] [4] y el proyecto DSA 1.0 [13] basado en OpenGL [9] funcionan sobre una entrada definida previamente y una vez iniciada la ejecución de las operaciones no se puede interactuar con la herramienta hasta que termina de procesar la entrada. Además, con la ayuda de la animación se busca que la visualización del comportamiento de cada estructura de datos o algoritmo de ordenamiento refleje cada operación relevante de los algoritmos, para facilitar de este modo la comprensión de las operaciones que se realizan.

En caso de que el usuario no tenga conocimiento previo sobre algún algoritmo o estructura, se podrá visualizar la ejecución completa de los algoritmos con una entrada de datos definida por el mismo usuario o alguna otra dada de manera aleatoria.

El desarrollo de este proyecto busca reunir atributos que se encuentran en herramientas ya existentes. Éstas carecen del carácter interactivo y de la manipulación de gráficas lo cual se pretende implementar en este proyecto, así como mejorar y agregar otras funcionalidades que no se han implementado. En cuanto a qué se va a mejorar, las animaciones reflejarán las operaciones relevantes que se efectúan dentro de una estructura de tipo árbol o algoritmo de ordenamiento. Esto es, el recorrido de un nodo a través del árbol hasta que se se inserta en él, y las comparaciones entre los valores de los nodos. En el caso de un algoritmo de ordenamiento, se podrá visualizar el segmento del arreglo en que se está trabajando, los intercambios de valores que haga el algoritmo, etc.

Las herramientas mencionadas antes (*applets* y DSA 1.0) no brindan la posibilidad de interactuar con los algoritmos, en el sentido de que se ocultan pasos de los algoritmos al usuario, no muestran las operaciones que se efectúan dentro de las estructuras tipo árbol o algoritmos de ordenamiento, por consiguiente no se logra una clara comprensión de éstos.

Esta interacción es la que se pretende implementar en el proyecto para permitir al usuario definir las entradas de datos y ver la ejecución de los algoritmos paso a paso. Esto último, se refiere a que se podrán ver de manera gráfica las operaciones y los cambios sobre las estructuras tipo árbol y algoritmos de ordenamiento.

## Descripción Técnica

El propósito principal del proyecto es la implementación de una herramienta de software con una interfaz gráfica de usuario (GUI, por sus siglas en inglés) que permita la interacción con las distintas estructuras tipo árbol, así como con los parámetros de los algoritmos de ordenamiento. El proyecto será dividido en cuatro bloques principales: la intercepción de eventos, la manipulación gráfica, la manipulación lógica y la generación de gráficos en la pantalla (renderización) (Figura 1).

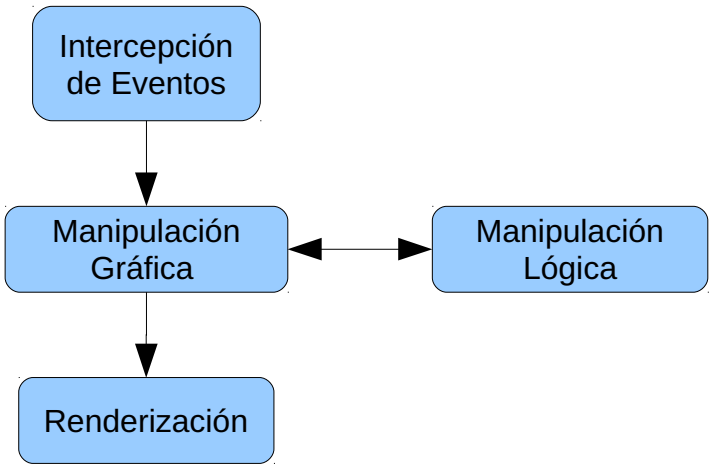



*Figura 1: Diagrama general de bloques de la GUI*

La intercepción de eventos es el bloque que permitirá la interacción del usuario con la GUI, ya que en éste serán interceptados todos los eventos externos (de *mouse* y teclado) que el usuario pudiera ejecutar sobre la GUI. Estos eventos se pueden ejecutar directamente sobre la representación gráfica de las estructuras tipo árbol o algoritmos de ordenamiento; o bien, podrían ser ejecutados sobre las opciones de la GUI (menús, botones, barras de desplazamiento, etc.). Todos estos eventos serán enviados al bloque de manipulación gráfica para que sean interpretados y representados en la parte gráfica y/o lógica de las estructuras tipo árbol o algoritmos de ordenamiento.

El bloque de la manipulación lógica se refiere a los cambios que puedan surgir sobre la representación lógica de las estructuras tipo árbol o algoritmos de ordenamiento, a través de la interacción con la representación gráfica de los primeros. Por otro lado, los cambios en la representación lógica pueden ser resultado de las operaciones definidas en la

implementación de las estructuras tipo árbol o algoritmos de ordenamiento y éstos serán enviados al bloque de manipulación gráfica para que se vean reflejados en la interpretación gráfica.

El bloque de manipulación gráfica se encargará de la gestión de las animaciones (desplazamientos, inserciones, recorridos, etc.) derivadas de las operaciones sobre las estructuras tipo árbol y algoritmos de ordenamiento. Además, ya que la representación lógica de las estructuras tipo árbol y algoritmos de ordenamiento está cambiando en forma constante, debido a las operaciones definidas en su implementación, estos cambios se tienen que ver reflejados en la parte gráfica; por tanto, éste bloque también se encargará de manipular todos estos cambios. Por otro lado, los eventos interceptados sobre la GUI se reciben en este bloque y se interpretan para su posterior representación gráfica, y si es el caso, se envían al bloque de manipulación lógica. Tanto los cambios en la representación lógica, como las animaciones, son enviados al bloque de renderización para su visualización en pantalla.

El bloque de renderización dibujará en pantalla la representación lógica de las estructuras tipo árbol y los algoritmos de ordenamiento, así como las animaciones provenientes del bloque de manipulación gráfica. La representación gráfica de las estructuras tipo árbol podría ser mediante esferas, las cuales representarán los nodos y una línea que las interconecte para representar las ligas entre nodos. Para representar el contenido de los nodos se puede añadir información a las esferas, esta información debe ser los datos almacenados en la estructura de datos tipo árbol. Para los algoritmos de ordenamiento, podrían utilizarse en un principio esferas de diferentes tamaños, aunque el uso de esferas con información también es viable.

Con el fin de hacer que los bloques de manipulación gráfica y manipulación lógica sean más específicos, éstos serán divididos en módulos que permitan tener una visión más detallada del funcionamiento general del bloque (Figura 2).

El bloque de manipulación gráfica se divide en dos módulos, los cuales son:

- Módulo de opciones.- Éste, a su vez, se divide en dos módulos, uno para las opciones de las estructuras tipo árbol y otro para los parámetros de los algoritmos de ordenamiento. En estos módulos se indicarán algunas características, como son: el tipo de datos de entrada (aleatorios, ordenados o introducidos en forma manual), la cantidad de datos, el tipo de estructura tipo árbol o algoritmo de ordenamiento, según corresponda, etc.

- Módulo para la manipulación gráfica.- Este módulo, al igual que el anterior, se divide en dos, para estructuras tipo árbol y algoritmos de ordenamiento. Este módulo se encargará de manejar las representaciones gráficas de las estructuras tipo árbol y algoritmos de ordenamiento, así como sus respectivas animaciones y transformaciones.

El bloque de manipulación lógica será dividido en dos módulos, éstos son:

- Módulo de la manipulación lógica.- Este módulo se divide en dos, uno para las estructuras tipo árbol y otro para los algoritmos de ordenamiento. Este módulo contendrá los algoritmos referentes a las estructuras tipo árbol y algoritmos de ordenamiento, así como las operaciones sobre los mismos.

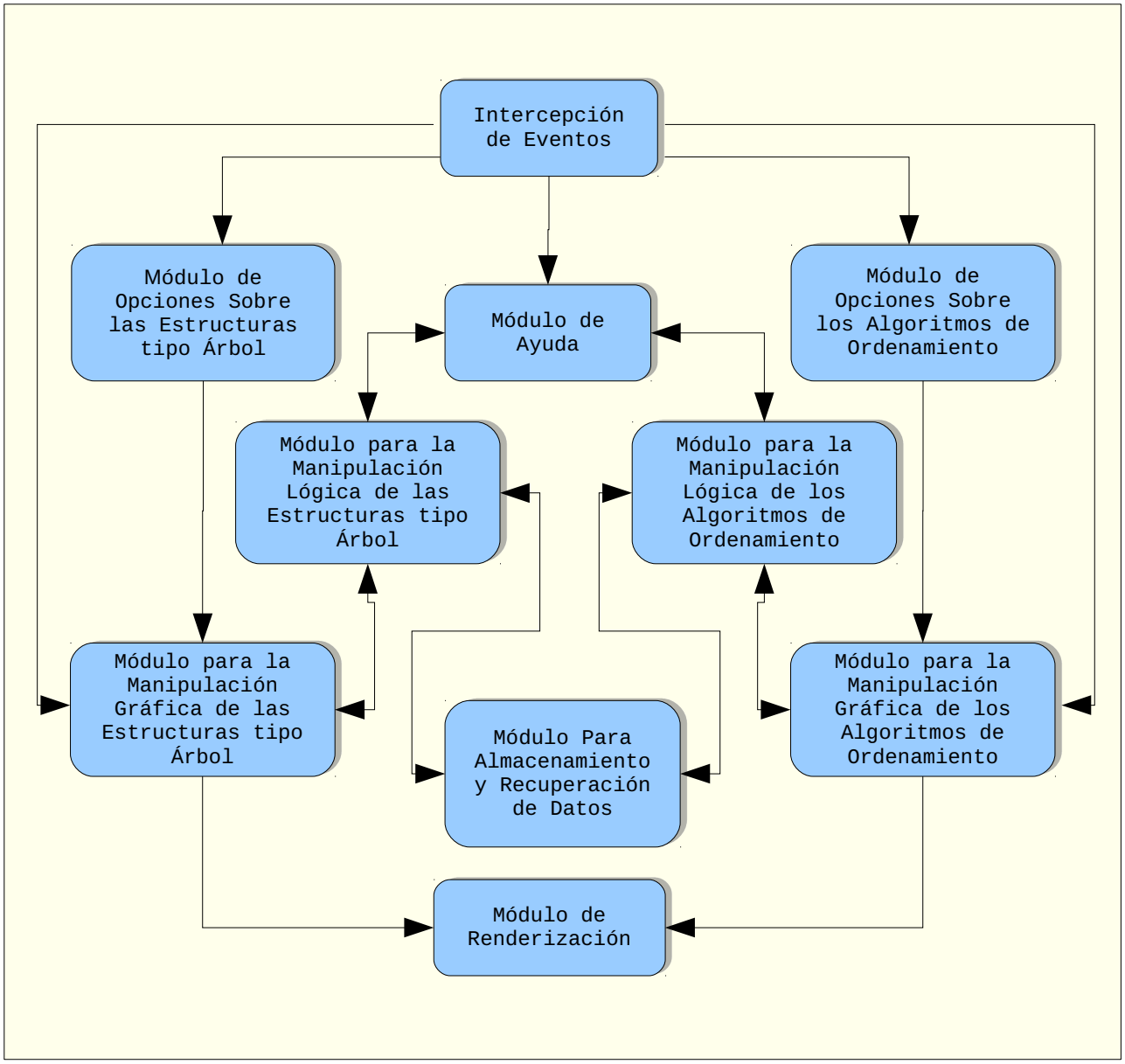



*Figura 2: Diagrama de módulos de la GUI*

- Módulo para almacenamiento y recuperación de datos.- Este módulo guardará en un dispositivo de almacenamiento secundario la estructura tipo árbol actual o el orden de los datos de algún algoritmo de ordenamiento. También permitirá recuperar alguna estructura tipo árbol o algoritmo de ordenamiento almacenado con anterioridad en modo texto.

El bloque de renderización no está dividido en módulos y su funcionalidad queda como se había especificado con anterioridad, en la descripción del bloque de renderización.

Como parte adicional, se integrará un módulo de ayuda. Este módulo mostrará información referente a las estructuras tipo árbol y para los algoritmos de ordenamiento. El módulo contendrá los algoritmos referentes a las estructuras tipo árbol y algoritmos de ordenamiento, así como las operaciones sobre los mismos.

La plataforma de trabajo que se utilizará es Linux, distribución Fedora Core 10 [16], de momento no se utilizará ningún ambiente de desarrollo.

Para desarrollar la interfaz gráfica se utilizará la biblioteca GTK+ 2.0 [17] así como la biblioteca GTKGLExt [18]. Para la representación gráfica y animaciones se utilizará OpenGL [9] y el lenguaje de programación será C/C++.

## Especificaciones Técnicas

Las estructuras tipo árbol se basarán únicamente en aquellas que sean del tipo árbol de búsqueda, así como en las operaciones de éstos (Tabla 1).

| Estructura de Árbol | Operaciones |
|---|---|
| Árboles Binarios de Búsqueda (BST) | Inserción<br>Eliminación<br>Búsqueda |
| Árboles AVL | Inserción<br>Eliminación<br>Búsqueda |
| Árboles Roji-Negros | Inserción<br>Eliminación<br>Búsqueda |
| Árboles 2-3-4 | Inserción<br>Eliminación<br>Búsqueda |

*Tabla 1: Estructuras de Árbol*

Debido a que existen una cantidad importante de algoritmos de ordenamiento y variantes de los mismos, el proyecto se basará en aquellos vistos en las UEA's (Unidad de Enseñanza – Aprendizaje) del Plan de estudios de la carrera Licenciatura en Ingeniería en Computación de la Universidad Autónoma Metropolitana – Azcapotzalco (Tabla 2).

| Algoritmos de Ordenamiento |
|---|
| Ordenamiento por mezcla |
| Ordenamiento de Hoare |
| Ordenamiento por montículos |
| Ordenamiento de shell |
| Ordenamiento por inserción |
| Ordenamiento por intercambio directo |
| Ordenamiento por selección |

*Tabla 2: Algoritmos de Ordenamiento*

Para las estructuras tipo árbol los datos que contendrán los nodos serán números o arreglos de caracteres de modo que se pueda hacer una comparación entre los nodos. Para determinar qué nodo es mayor que otro se hará una comparación de su valor numérico o en su caso una comparación alfabética de los arreglos de caracteres.

El número máximo de datos a ingresar será de 30 datos para el caso de trabajar con una estructura tipo árbol, esto quiere decir, que se podrá generar un árbol de a lo más 30 nodos, y si se va trabajar un algoritmo de ordenamiento la entrada máxima permitida será de 50 datos. La cantidad de datos se determino en base a que la herramienta no realizará ningún análisis de tiempos de ejecución, el fin practico de la herramienta consiste en ilustrar el comportamiento de estructuras tipo árbol y algoritmos de ordenamiento para su entendimiento, por lo que llevar a cabo el procesamiento de entradas de datos muy grandes no resulta útil.

Para el caso de los algoritmos de ordenamiento los elementos que conformarán su representación gráfica serán figuras geométricas de tamaños diferentes, o en su caso de tamaños iguales y contendrán un valor numérico para distinguir un elemento de otro.

Tanto las estructuras de árbol como los algoritmos de ordenamiento en ningún caso se permitirá la repetición de elementos, esto es, elementos con la mismo valor o elementos del mismo tamaño (según sea el caso).

Para que todos los elementos puedan ser representados en pantalla se considerará el número total de elementos y el tamaño de la ventana en que serán representados, para de este modo hacer un cálculo del tamaño de cada elemento y así evitar que los elementos vayan aumentando o disminuyendo de tamaño conforme se agreguen o eliminen éstos.

La capacidad máxima del proyecto no se alcanzara debido a que la aplicación no será multiplataforma, en un principio, además de que muchas estructuras de árbol y algoritmos de

ordenamiento no serán implementados.

Para que el proyecto pueda considerarse como concluido cada uno de los bloques que lo conforma deberán de cumplir con sus especificaciones antes mencionadas (ver Descripción Técnica ) y al final la integración de los mismos deberán satisfacer el objetivo general del proyecto.

Como la parte final del proyecto se entregara en un DVD toda la documentación referente al proyecto, esto es, manual de usuario, manual de instalación así como todo el código fuente del proyecto y manual del programador que incluye los diagramas de estado, flujo y entidad-relación; y los formatos de archivos de entrada y salida.

# Desarrollo del proyecto

## Selección de estructuras tipo árbol

### Árbol binario de búsqueda

Esta es la versión mas sencilla de una estructura de tipo árbol,  básicamente la estructura comienza con un nodo raíz, el cual cambiara solo y solo si este es eliminado del árbol, y sera sustituido por otro elemento dentro del árbol la regla que sigue para asignar una nueva raíz es sustituir por el mayor de los menores. Un elemento nuevo se ubicara hacia la izquierda de la raíz si es menor a esta, de lo contrario se ubicara a la derecha.

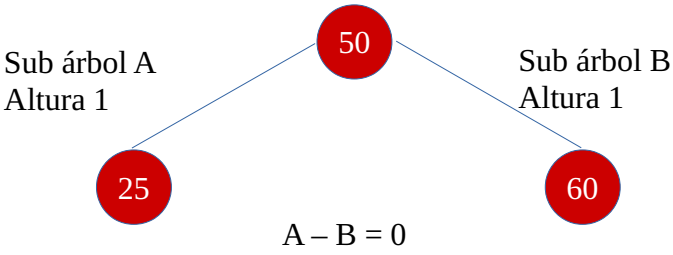
Para la implementar  el árbol binario se utilizó un algoritmo recursivo:

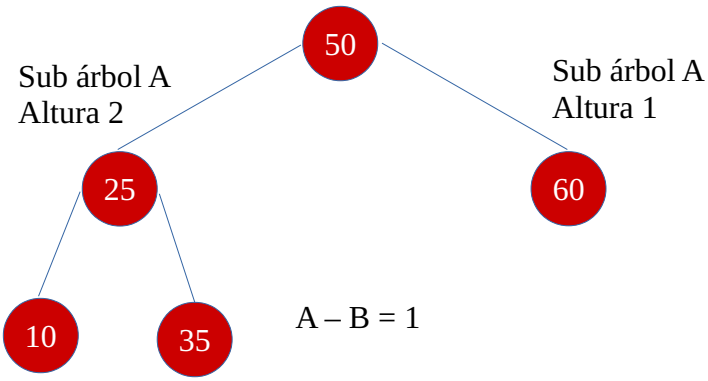
```
tarbol ** insertarAB(tarbol **a, int val){

    if (*a== NULL){
        *a=(tarbol * )malloc(sizeof(tarbol));
        (*a)->clave = val;
        (*a)->izq = (*a)->der = NULL;
        return a;
    }
    else{
        if (val < (*a)->clave )
            insertarAB(&(*a)->izq, val);
        else
            insertarAB(&(*a)->der, val);
    }

}

void  borrar(tarbol **a, int elem)
{
   tarbol *aux;

  if (*a == NULL)   /* no existe la clave */
```

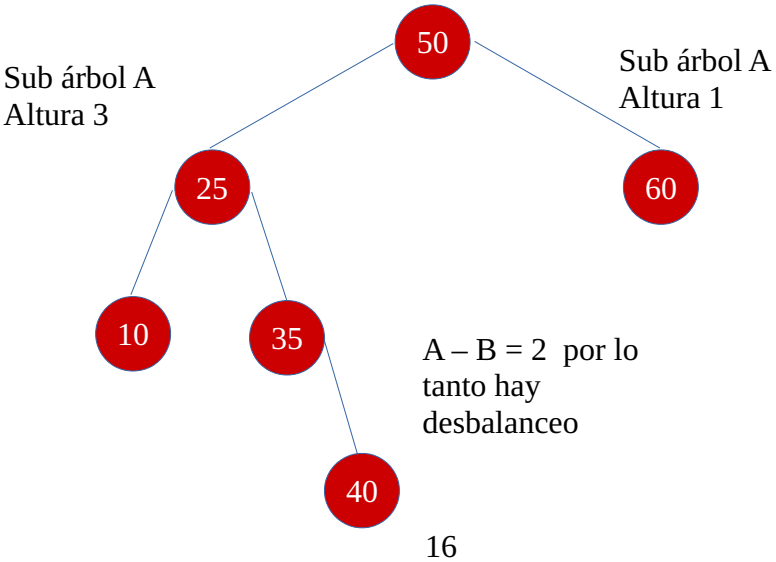
```
  {
       return ;
  }
  if ((*a)->clave < elem)
          ret = borrar(&(*a)->der, elem);
  else if ((*a)->clave > elem)
          ret = borrar(&(*a)->izq, elem);
  else if ((*a)->clave == elem)
  {
          aux = *a;
          if ((*a)->izq == NULL)
          {
          if ((*a)->der == NULL)
          {
                 *a = (*a)->der;  //es una hoja
             return ;
          }
       else
       {

          *a = (*a)->der;
       }
    }
    else if ((*a)->der == NULL)
    {
       *a = (*a)->izq;
    }
    else
    {
       sustituir(&(*a)->izq, &aux); /* se sustituye por el mayor de los menores
*/
    }
    free(aux);
  }
return ret;
}

void sustituir(tarbol **a, tarbol **aux )
{

  if ((*a)->der != NULL)
      sustituir(&(*a)->der, aux);
  else
  {
    (*aux)->clave = (*a)->clave;
    *aux = *a;
    *a = (*a)->izq;
   }
}
```

**Árbol AVL**

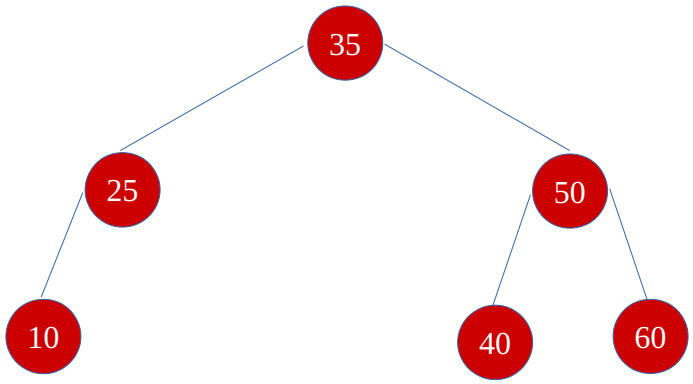
Esta es una variación del árbol binario, la inserción se realiza de la misma forma, pero a diferencia del árbol binario, un árbol AVL se re configura con el paso de las inserciones o bien al eliminar elementos, la re estructuración del árbol esta en función de una propiedad denominada altura del sub árbol, esta propiedad dice que la diferencia entre las alturas de los sub árboles de un nodo raíz no debe ser mayor a 1, es decir:



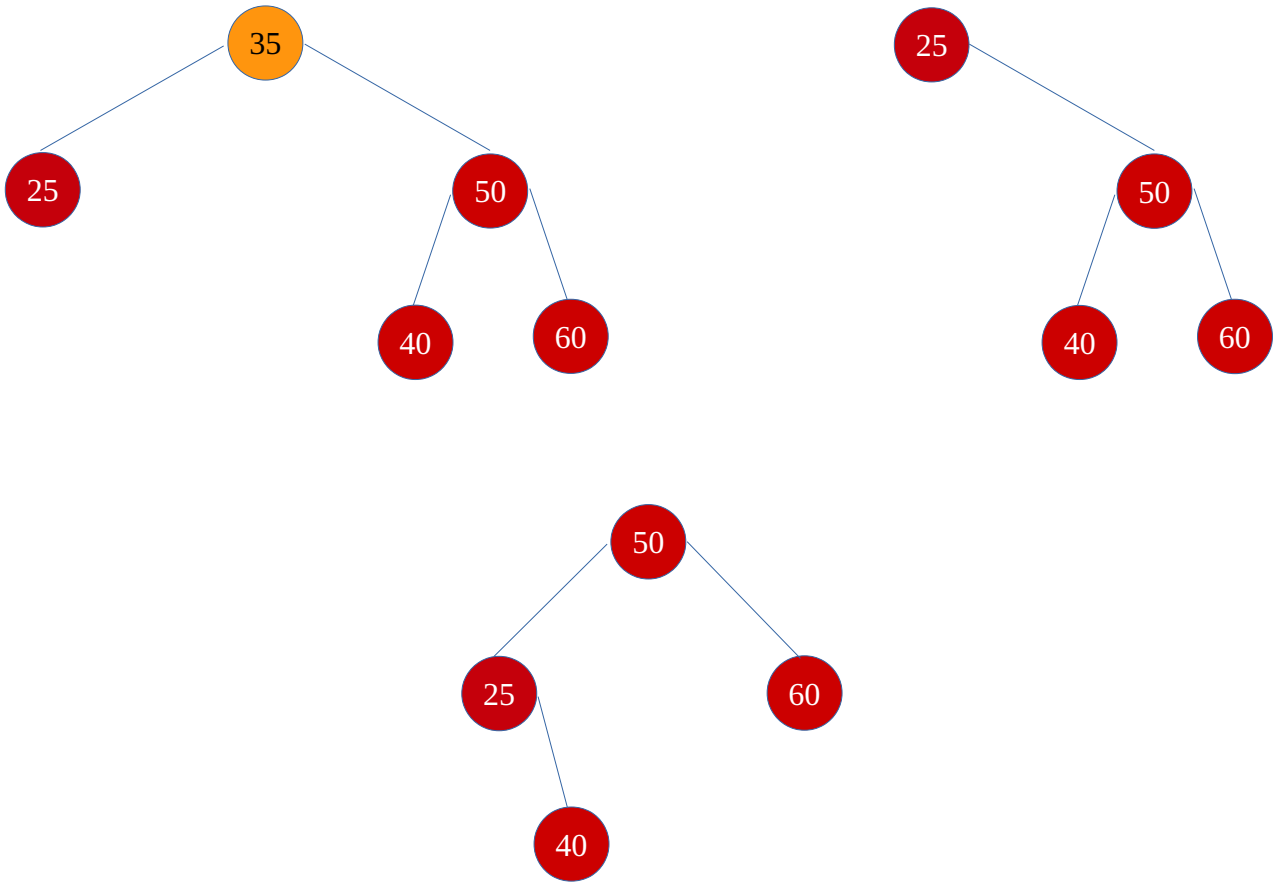
El algoritmo algoritmos realiza una serie de pasos llamadas rotaciones, para balancear el árbol esto implica que la raíz del árbol cambiara.




El Algoritmo funciona de forma muy similar a la inserción en un árbol binario con la diferencia de que posterior a la inserción se debe balancear el árbol.

```
AVLtree **insertarAVL (AVLtree **t, int elem)
{

AVLtree **nodo;
    if (es_vacio (*t))
    {
          *t = (AVLtree *)malloc(sizeof(AVLtree));
          (*t)->izq = NULL;
          (*t)->der = NULL;
          (*t)->clave = elem;
          return t;
    }
    else
    {
       if (elem < (*t)->clave)
       {
           insertarAVL (&(*t)->izq, elem);
       }
       else
       {
           insertarAVL (&(*t)->der, elem);
       }
       balancear (t);
       actualizar_altura (*t);
    }
}
```

Cuando se lleva acabo una eliminación de un elemento, este algoritmo en particular sigue la regla de sustitución por el mayor de los menores, una vez hecha la sustitución se procede a verificar las alturas del árbol, y de ser necesario se balancea nuevamente tal y como sucede después de insertar un elemento nuevo.




El código implementado para la eliminación en árbol AVL es el siguiente:

```
void  eliminarAVL(AVLtree **a, int elem)
{
   AVLtree *aux;

  if (*a == NULL)   /* no existe la clave */
  {
          return ;
  }
  if ((*a)->clave < elem)
          eliminarAVL(&(*a)->der, elem);
  else if ((*a)->clave > elem)
          eliminarAVL(&(*a)->izq, elem);

else if ((*a)->clave == elem)
{
          aux = *a;
    if ((*a)->izq == NULL)
```

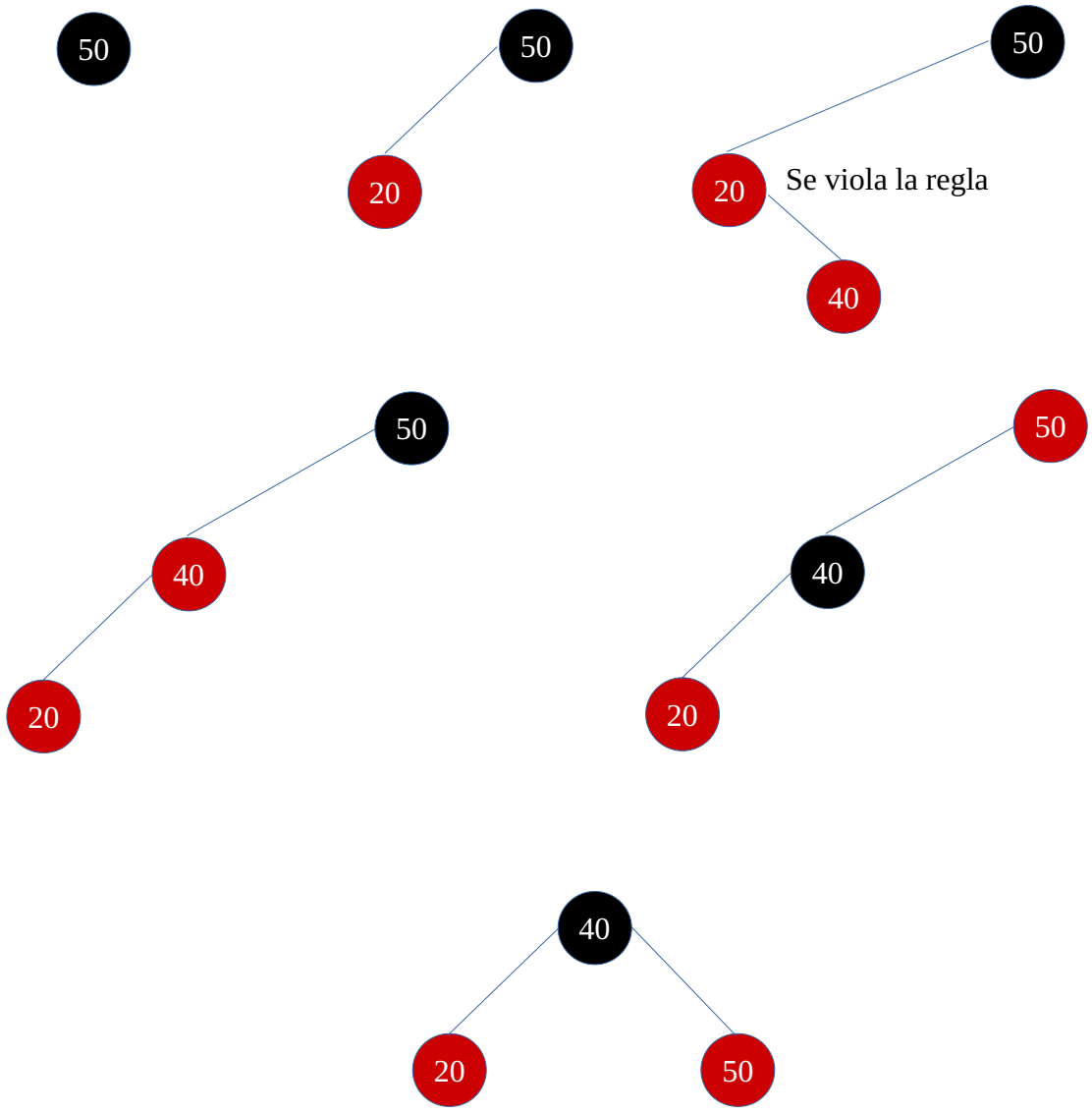
```
    {
       if ((*a)->der == NULL)
       {

              *a = (*a)->der;
                 return ; /// hoja
       }
       else
       {
          *a = (*a)->der;
          balancear(a);
          actualizar_altura(*a);
       }
    }
    else if ((*a)->der == NULL)
    {
       *a = (*a)->izq;
       balancear(a);
       actualizar_altura(*a);
    }
    else
    {
       sustituir(&(*a)->izq); /* se sustituye por la mayor de las menores */
       balancear (a);
       actualizar_altura(*a);
    }

  }
}

void sustituir(AVLtree **a)
{
   if ((*a)->der != NULL)
          sustituir(&(*a)->der);
       balancear(a);
       actualizar_altura(*a);
   else
   {
       AVLtree *aux = (*a);
       *a = (*a)->izq;
       free(*aux);
       balancear (a);
       actualizar_altura (*a);
   }

}
```

**Árbol roji-negro**

Para esta variación de árbol se deben cumplir algunas propiedades:

- Un nodo puede ser rojo o negro.
- Un nodo rojo tiene dos hijos negros.
- Todo camino desde la raíz a cualquier hoja del árbol debe pasar por el mismo número de nodos negros.
- El nodo raíz siempre es negro.
- Los nodos nulos son negros.

La inserción en un principio es igual que en un árbol binario de búsqueda, con cada elemento insertado se debe verificar que no se violen las propiedades que debe conservar este tipo de árbol, de ser así se procede ajustar los colores de los nodos y por ende se re estructura.



Este árbol al igual que el tipo AVL se re configura posterior a una inserción o a eliminación de elementos, con la diferencia en sus criterios de balanceo, mientras que un árbol AVL obedece estrictamente a las alturas de los sub árboles de un nodo, un árbol roji  negro lo hace basado en las propiedades antes mencionadas.

```
nodeRB** insertRB (treeRB *tree ,nodeRB** t, int key){
    nodeRB** newNode;
    if (*t == tree->nil)
    {

       (*t) = (nodeRB*) SafeMalloc(sizeof(nodeRB));
       (*t)->key = key;
       (*t)->nivel = nivel;
       (*t)->parent = parent;
       (*t)->left=tree->nil;
       (*t)->right=tree->nil;
       return t;

    }
    else
    {
   if (key < (*t)->key)
   {

       newNode = insertRB (tree, &(*t)->left,key);
   }
   else
   {

       newNode = insertRB (tree, &(*t)->right,key);
   }
    }
(*newNode)->red=1;

insertFixUp(tree, newNode );

return newNode;
}
```

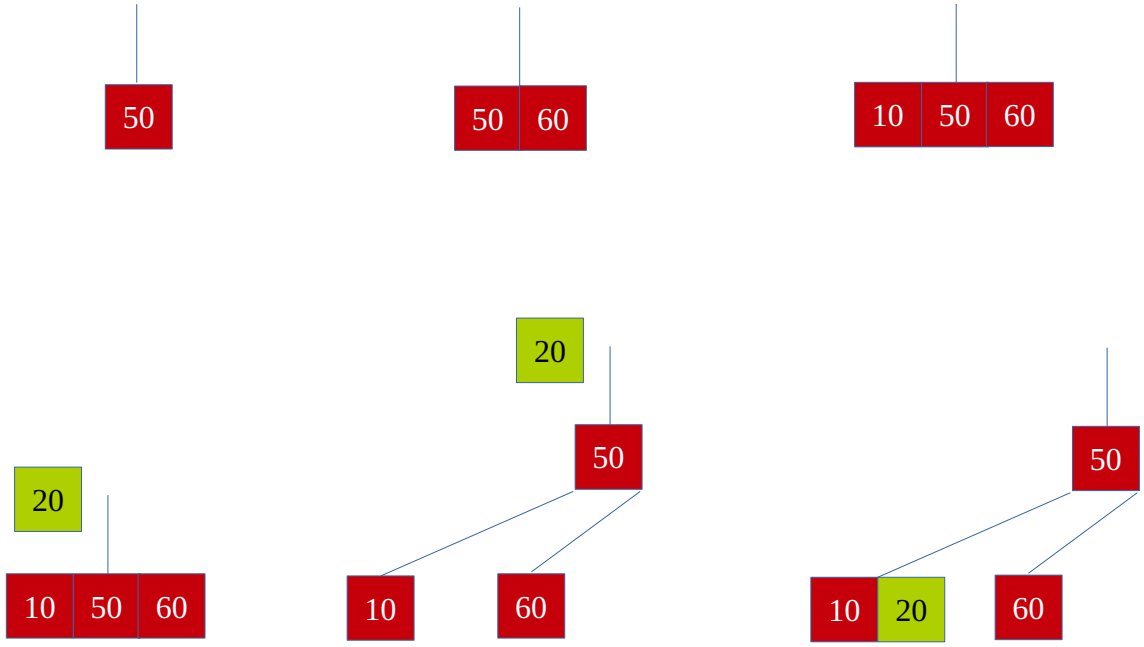
las regla para eliminar cambia para este caso, este código sustituye por el menor de los mayores, y nuevamente terminada la eliminación el árbol debe actualizar colores de nodo, y de ser necesario realizar rotaciones para conservar sus propiedades.

```
void deleteRB (treeRB *tree, nodeRB*z ){
     nodeRB* y;
     nodeRB* x;
     nodeRB* nil=tree->nil;
     nodeRB* root=tree->root;

        if (z->left == nil || z->right == nil)
            y = z;
```

```
        else
            y= TreeSuccessor(tree, z);

        if (y->left == nil)
                x = y->right;
        else
                x = y->left;

        if (root == (x->parent = y->parent)){
                root->left = x;
        }
        else
        {
                if (y == y->parent->left){
                y->parent->left = x;
                }else{
                y->parent->right = x;
                }
        }


        if (y != z){
                if (!(y->red)){
                    deleteFixUp(tree, x);
             }
                y->left = z->left;
                y->right = z->right;
                y->parent = z->parent;
                y->red = z->red;
                z->left->parent = z->right->parent = y;
                if (z == z->parent->left){
               z->parent->left = y;
                }else{
               z->parent->right = y;
                }
             free (z);

        }
        else{
                if (!(y->red)){

                   deleteFixUp( tree, x);
                }
                free (y);
        }
}
```

**Árbol 2-3-4**

Esta estructura es diferente al árbol convencional anteriormente se trabajaron nodos con una sola llave, para este caso particular el nodo puede tener hasta 3 llaves, y hasta 4 hijos nodos, la inserción para este árbol es ligeramente diferente al árbol binario inicialmente se empieza con un nodo raíz con una llave, las inserciones se hacen en el mismo nodo hasta llegar a 3 elementos ordenando de menor a mayor, posterior a esto se parte el nodo para agregar un nuevo elemento.




el código para implementar el árbol 2.-3-4 es el siguiente:

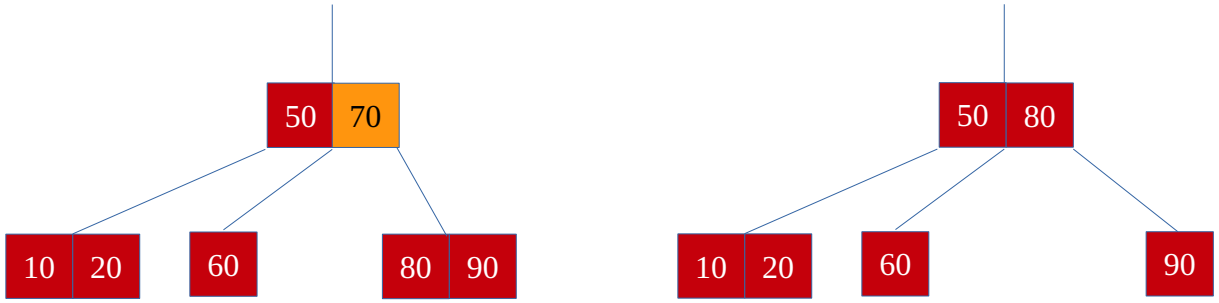
```
void Arbolttf::insertar(int k)
{
  Node *x = root;

    if (buscar(root,k) == 1)
        cout <<"el elemento ya existe!!!!\n";
    else
    {
        if (x->n == 3)
        {
            Node *s = new Node(0,-1,true);
            root = s;
            s->leaf = false;
            s->l[0] = x;

            partir_hijo (s,1,x);
            insertar_nonfull (s,k);
        }
```

```
       else
          insertar_nonfull (x,k);
    }
}


void Arbolttf::insertar_nonfull (Node *x, int k)
{

    int i = (x->n);

    if (x->leaf)
    {

       while (i >= 1 && k < x->key[i-1].val)
       {
          x->key[i].val = x->key[i-1].val;
          i--;
       }
       x->key[i].val = k;
       x->n ++;

    }
    else
    {

       while (i >= 1 && k < x->key[i-1].val)
          i--;
       i++;
       if (x->l[i-1]->n == 3)
       {
          partir_hijo(x,i,x->l[i-1]);
          if (k > x->key[i-1].val)
             i++;
       }
       insertar_nonfull (x->l[i-1], k);

    }
}

void Arbolttf::partir_hijo(Node *x, int i, Node *y)
{
     Node *z;
     int j;
     z = new Node(0,-1,true);
     z->leaf = y->leaf;
     z->n = 1;
     z->key[0].val = y->key[2].val;

     if (!y->leaf)
     {
    z->l[1] = y->l[3];
    z->l[0] = y->l[2];
```

```
    y->l[3] = NULL;
    y->l[2] = NULL;
     }
     y->n = 1;

     for (j = (x->n + 1); j >= (i + 1); j--)
     {
    x->l[j] = x->l[j-1];
    x->key[j-1].val = x->key[j-2].val;
     }
     x->l[i] = z;
     x->key[i-1].val = y->key[1].val;
     x->n++;
}
```

El proceso de eliminación de un elemento implica una re estructuración del árbol esto si se da un caso en el que el nodo queda sin elementos.


O bien, si un elemento que se elimina tiene nodos hijos.



## Creación de la interfaz gráfica

### Glade

Para la creación de la interfaz gráfica del proyecto se utilizó la herramienta llamada Glade, esta herramienta permite crear de forma visual una interfaz de usuario; en pocas palabras es una GUI (Interfaz Gráfica de Usuario) para crear Interfaces Gráficas de Usuario. En la Figura 3 se aprecia una captura de pantalla de la ventana principal de la herramienta Glade.

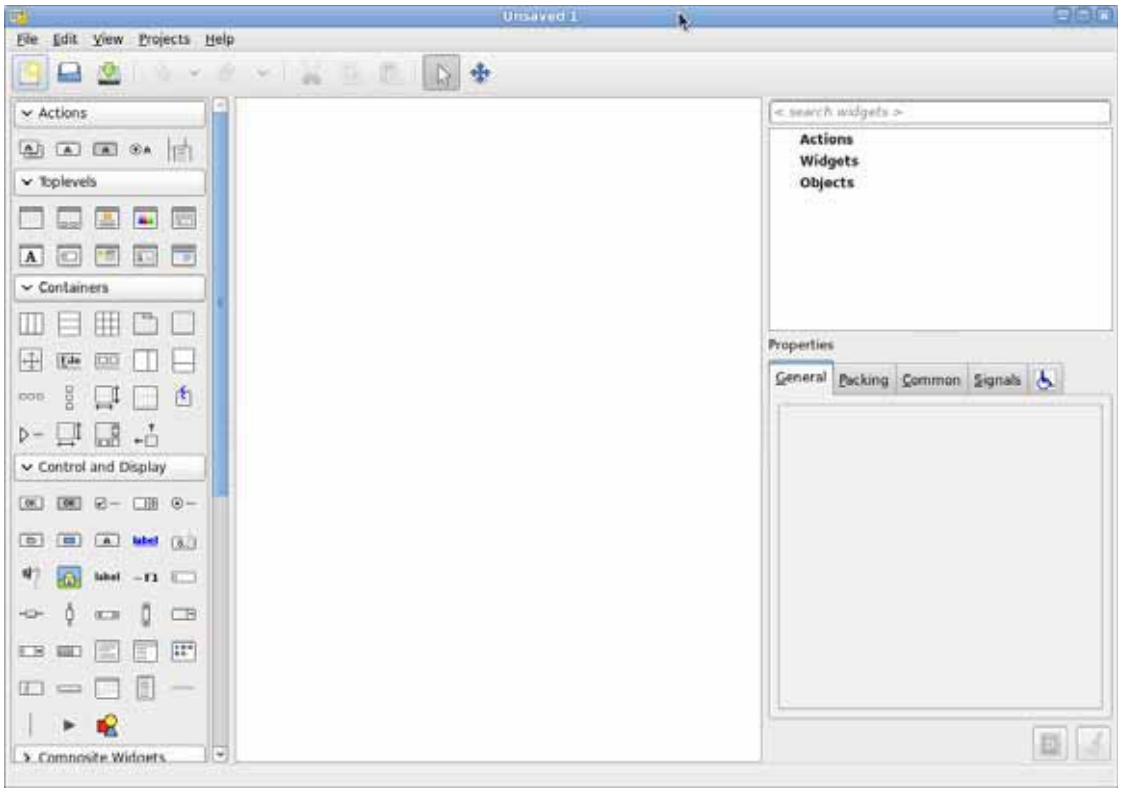



*Figura 3: Ventana principal de la herramienta Glade*

Glade contiene una lista de elementos que, mediante la filosofía *drag-and-drop* (arrastrar y soltar), se pueden ir agregando a un área de diseño para la creación de interfaces gráficas de todo tipo.

### Elementos de la interfaz gráfica

Una interfaz gráfica creada en Glade es una composición de un conjunto de elementos ordenados de forma jerárquica, i.e. que puede haber elementos dentro de otros elementos que compartan ciertas propiedades y atributos. A continuación se mencionan algunos de los elementos más importantes que fueron utilizados para la creación de la interfaz gráfica:

- **Window (ventana)**: es el contenedor principal de todos los elementos y es el que tiene la mayor jerarquía. Este contenedor ya cuenta con algunos elementos como son los botones de minimizar, maximizar y cerrar.

- **Horizontal box (caja horizontal)**: este es un contenedor que sirve para agrupar y organizar elementos dentro de él, la organización de elementos se hace de forma horizontal (izquierda a derecha).

- **Vertical box (caja vertical)**: como el elemento anterior, este contenedor agrupa y organiza los elementos que contiene en una forma vertical (de arriba a abajo).

- **Menu bar (barra de menú)**: este elemento es un contenedor que agrupa unos elementos especiales llamados *menu item* (ítems de menú); y en pocas palabras es la barra de menú que se puede apreciar en la mayoría de las interfaces gráficas.

- **Menu Items (Ítems de menú)**: como se mencionó con anterioridad estos son elementos especiales que se agrupan en el contenedor *barra de menú* y que al ser seleccionados con el apuntador del ratón, o mediante comandos del teclado, despliegan una lista de elementos llamados *menús emergentes*.

- **Popup menu (menú emergente)**: son elementos que están contenidos dentro de los *ítems de menú* mencionados anteriormente.

- **Label (etiqueta)**: estos son elementos que pueden contener cualquier texto y sirven para poner etiquetas dentro de la interfaz gráfica.

- **Combo box (cuadro combinado)**: este elemento es una caja que contiene una lista de elementos en forma de texto la cual al ser seleccionada se expande y muestra la lista completa de los elementos que contiene y su función principal es el poder seleccionar sólo uno de esos elementos.

- **Spin button (botón de giro)**: este elemento es un botón que permite seleccionar elementos en forma de numero, mediante el incremento o decremento de un rango de números.

- **Text entry (entrada de texto)**: este es un elemento donde se puede introducir texto mediante el teclado, o algún otro método de entrada.

- **Horizontal button box (caja de botones horizontales)**: este es un contenedor que agrupa y ordena elementos especiales llamados *butones* de una forma horizontal (de izquierda a derecha).

- **Button (botón)**: este elemento es un botón que al ser presionado desencadena una cierta acción.

- **Horizontal separator (separador horizontal)**: este elemento es una línea horizontal que sirve para separar visualmente diferentes secciones dentro de la interfaz gráfica.

En la Figura 4 se puede apreciar el diseño de la interfaz gráfica del proyecto implementada

en la herramienta Glade. Este diseño es guardado en un archivo especial con extensión `.glade`, este archivo se puede encontrar en la carpeta principal del proyecto dentro de la siguiente ruta: `Arboles/Código Fuente/src/UI.glade`.

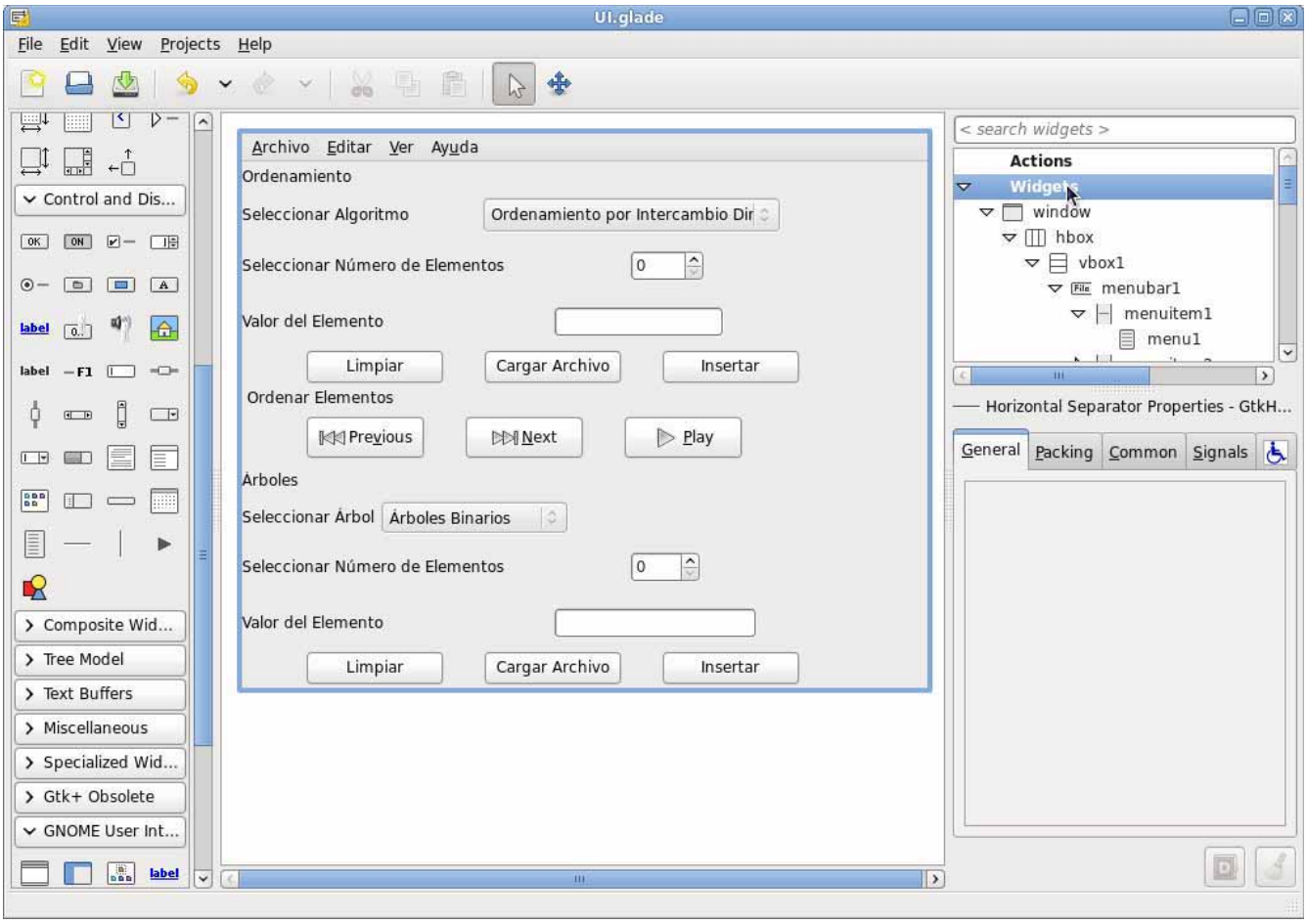



*Figura 4: Diseño de la interfaz gráfica en Glade*

### Conversión del archivo creado por Glade

Los archivos creados por Glade son archivos con extensión `.glade`, el problema de estos archivos especiales es que sólo Glade los puede interpretar por lo que para poder integrar estos archivos en un proyecto que utiliza lenguaje C primero se tienen que convertir a un formato XML. Para realizar esta conversión se utilizó una herramienta llamada: ***gtk-builder-convert.*** Está herramienta se utilizó desde una terminal y se ejecutó de la siguiente manera:

```
gtk-builder-convert UI.glade UI.xml
```

### Lanzando la interfaz gráfica desde lenguaje C

Ya que el archivo en formato XML fue creado la interfaz gráfica pudo ser lanzada desde un programa en lenguaje C, para este efecto se tuvieron que realizar una serie de pasos que a continuación se describen:

1. Conexión con el archivo XML.

```
builder = gtk_builder_new ();
gtk_builder_add_from_file (builder, "UI.xml", NULL);
```

   En este paso la interfaz gráfica contenida en `UI.xml` fue copiada en la estructura `builder`

2. Obtención de los Widgets (elementos de la interfaz gráfica) que se utilizaron.

```
widgets.screen.window = GTK_WIDGET (gtk_builder_get_object (builder, "window"));
widgets.hbox = GTK_BOX (gtk_builder_get_object (builder, "hbox"));

widgets.expander1 = GTK_WIDGET(gtk_builder_get_object(builder, "vbox2"));
widgets.expander2 = GTK_WIDGET(gtk_builder_get_object(builder, "vbox3"));

...
```

   En este paso se guardaron, en una estructura local, los elementos de la interfaz gráfica diseñada en Glade que se utilizaron en la implementación del proyecto. Por ejemplo, el elemento `window` se almacenó en la estructura `widgets.screen.window`.

3. Conexión de las señales de los *elementos de la interfaz gráfica* con sus respectivas *Callbacks (llamadas de retorno)*.

```
gtk_builder_connect_signals (builder, &widgets);
```

   En este paso se asociaron las señales almacenadas en la estructura `builder` con los elementos contenidos en la estructura `widgets`. Este proceso será explicado más a fondo en la sección **Intercepción de Eventos**.

4. Liberación de la estructura `builder`.

```
g_object_unref (G_OBJECT (builder));
```

   Ya que las señales fueron asociadas con los elementos de la estructura local `widgets` la estructura `builder` ya no era necesaria y fue liberada.

5. Mostrando la interfaz gráfica en pantalla.

```
gtk_widget_show (widgets.screen.da);
gtk_widget_show_all (widgets.screen.window);
gtk_widget_hide (widgets.expander2);
```

   Es hasta este punto que la interfaz gráfica fue mostrada en pantalla. La primer sentencia muestra el área designada para la renderización de OpenGL, esto se explicará con más detalle en la sección **Renderización**. La segunda sentencia

muestra en pantalla el elemento `window`, y ya que este elemento es el más alto en la jerarquía de elementos y además contiene a todos los demás elementos, sólo fue necesario mostrar este para que toda la interfaz fuera visualizada. La tercer sentencia esconde el elemento expander2, este elemento es el que contiene la interfaz gráfica para las estructuras tipo árbol.

Una vez que se terminaron de ejecutar estos pasos entonces se visualizó una interfaz como la mostrada en la Figura 5.

*Figura 5: Interfaz Gráfica del proyecto*

**Añadiendo eventos en Glade**

Cuando se está diseñando la interfaz gráfica en Glade es posible seleccionar que eventos serán ejecutados cuando el usuario tenga cierta interacción con algún elemento de la interfaz gráfica. Por ejemplo, para añadir un evento a la acción de hacer clic con el ratón sobre el botón “Cargar Archivo”, primero se selecciona el botón deseado, como se muestra en **1** en la Figura 6; posteriormente se selecciona el “*Tab*” “*Signals*”, se elige la acción deseada, en este caso “*clicked*”, y se escribe el nombre de la *llamada de retorno* a la cual estará asociada la señal, como se muestra en **2** en la Figura 6

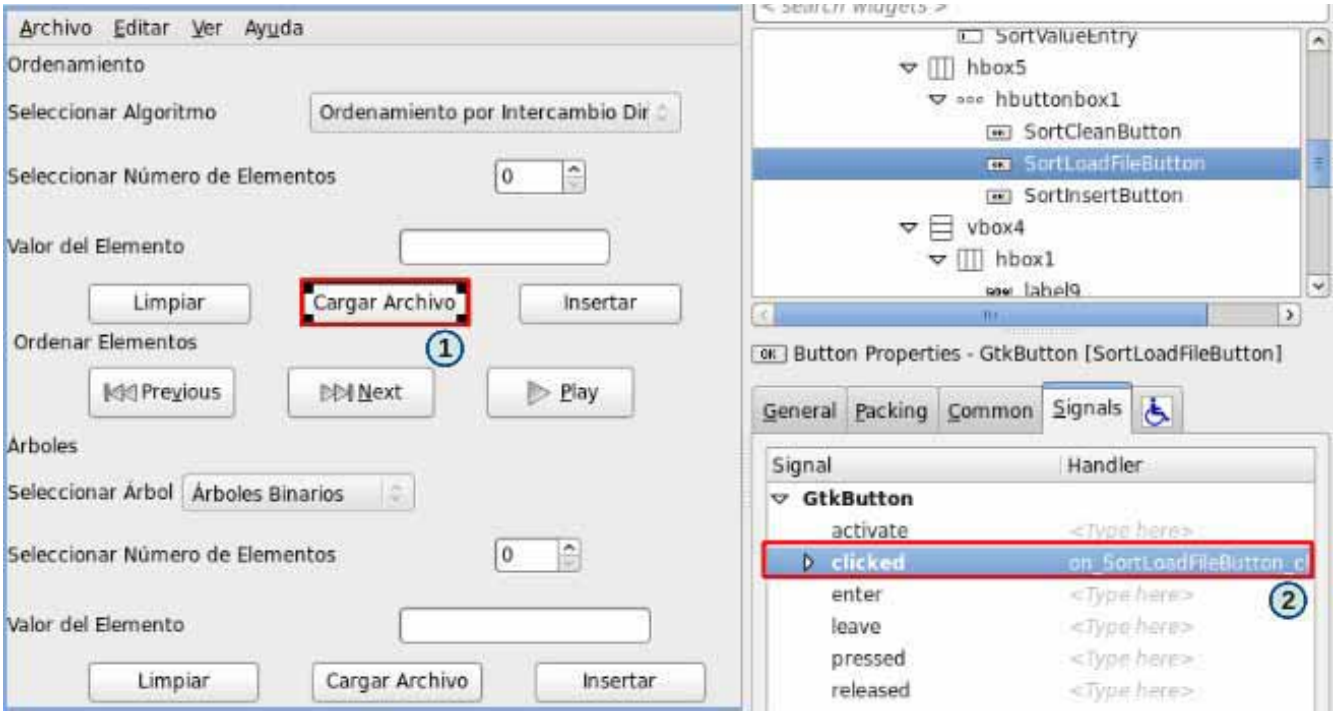



*Figura 6: Añadiendo eventos en Glade*

En la Tabla 3 se puede apreciar la relación de los elementos de la interfaz que ejecutan un evento cuando se tiene interacción con ellos, así como la respectiva *llamada de retorno* que ejecutará.

| Elemento (Nombre) | Evento | Llamada de retorno |
|---|---|---|
| `Menu Item (Ordenamiento)` | `activate` | `on_Ordenamiento_activate` |
| `Menu Item (Arboles)` | `activate` | `on_Arboles_activate` |
| `Spin Button (SortMaxNum)` | `value-changed` | `on_SortMaxNum_value_changed` |
| `Button (SortCleanButton)` | `clicked` | `on_SortCleanButton_clicked` |
| `Button (SortLoadFileButton)` | `clicked` | `on_SortLoadFileButton_clicked` |
| `Button (SortInsertButton)` | `clicked` | `on_SortInsertButton_clicked` |
| `Button (SortStepButton)` | `clicked` | `on_SortStepButton_clicked` |
| `Button (SortSortingButton)` | `clicked` | `on_SortSortingButton_clicked` |

*Tabla 3: Eventos de la interfaz gráfica*

**Llamadas de retorno**

Las *llamadas de retorno* como se mencionó con anterioridad son funciones especiales que son ejecutadas cada que se detecta un evento dentro de la interfaz gráfica, en la sección pasada se agregaron eventos sobre algunos elementos de la interfaz gráfica. Dentro del directorio principal del proyecto en la ruta: `Arboles/Código Fuente/src/GtkSignals.h` se encuentran los prototipos de las *llamadas de retorno* del proyecto. A continuación se listan algunos prototipos de función:

```
void
realize(GtkWidget *widget, gpointer data);

gboolean
expose_event(GtkWidget *da, GdkEventExpose *event, gpointer user_data);

gboolean
configure_event(GtkWidget *da, GdkEventConfigure *event, gpointer user_data);

gboolean
timeout (gpointer user_data);

extern "C" void
on_SortMaxNum_value_changed(GtkSpinButton *spin, gpointer user_data);

extern "C" void
on_SortLoadFileButton_clicked(GtkButton *button, gpointer user_data);

...
```

Los primeros cuatro prototipos corresponden a eventos que se llevan a cabo cuando se está haciendo la renderización de la escena. Los siguientes dos corresponden a eventos que fueron configurados desde Glade y como este tipo de eventos están declarados fuera del ámbito del proyecto es por eso que llevan `extern “C”`.

**Asociando los eventos con sus llamadas de retorno**

Existen dos formas de asociar los eventos con sus *llamadas de retorno*. La primer forma fue examinada en el paso 3 de la sección **Lanzando la interfaz gráfica desde lenguaje C**, en este paso todos los eventos que fueron declarados desde Glade se asocian con las *llamadas de retorno* contenidas en el archivo de encabezado `GtkSignals.h`. La segunda forma es asociándolas directamente desde el lenguaje C mediante la función `g_signal_connect()`, a continuación se listan tres eventos que fueron asociados con esta función:

```
  g_signal_connect (G_OBJECT (widgets.screen.da), "realize",
                          G_CALLBACK (realize), &widgets);

  g_signal_connect (G_OBJECT (widgets.screen.da), "configure_event",
          G_CALLBACK (configure_event), &widgets);
```

```
g_signal_connect (G_OBJECT (widgets.screen.da), "expose_event",
        G_CALLBACK (expose_event), &widgets);
```

En estas tres funciones se asocian tres eventos diferentes a un mismo elemento, que es `widgets.screen.da`, y cada uno de ellos tiene su propia *llamada de retorno*: `realize`, `configure_event`, `expose_event`.

## Manejo de opciones

El manejo de las opciones dentro de la interfaz gráfica es una parte de suma importancia ya que con esto se puede decidir la forma en que se comportan los algoritmos de ordenamiento dentro del proyecto.

**Cargando opciones de renderización**

Las opciones de renderización decidirán el modo en el que se representaran gráficamente los elementos dentro de la interfaz, por ejemplo el color de los elementos, el color de las fuentes, la posición de las luces, el tamaño de los elementos, etc.

Para llevar a cabo esta parte se necesitan tres partes importantes: un archivo de configuración, una estructura para almacenar las opciones y un *parser (analizador)* para leer las opciones del archivo de configuración y guardarlas en la estructura.

Archivo de configuración

El archivo de configuración se encuentra dentro del directorio principal del proyecto en la ruta: `Arboles/Código Fuente/src/config.`

Este archivo de configuración es un archivo de texto que puede ser editado para cambiar el comportamiento de la renderización dentro de la interfaz gráfica. Este archivo está dividido en tres grupos: `ligths`, `color` y `render;` en cada grupo hay diferentes opciones que pueden ser modificadas.

```
[lights]

lightPos=-50.0;100.0;-100.0;1.0;
lightPosMirror=-50.0;100.0;-100.0;1.0;
noLight=0.0;0.0;0.0;0.0;
lowLight=0.2;0.2;0.2;1.0;
brightLight=1.0;1.0;1.0;1.0;

[colors]

groundColor=1.0;1.0;1.0;0.4;
backGroundColor=0.2;0.2;0.2;1.0;
isHLV=false;
```

```
elementColor=1.0;0.2;0.2;1.0;
fontColor=1.0;1.0;1.0;1.0;
limitColor1=0.0;0.0;1.0;0.2;
limitColor2=1.0;1.0;0.0;0.4;
linkColor=0.0;0.0;1.0;0.6;
biggerColor=1.0;0.0;1.0;0.6;
smallerColor=0.0;1.0;1.0;0.6;

[render]

isFont=true;
showLimits=true;
sortWidth=5.0;
sortSpacing=2.5;
treeWidth=10.0;
```

El nombre del grupo se encuentra encerrado entre corchetes [] y forman parte de él todos los elementos que se encuentran por debajo hasta encontrar un nuevo grupo o llegar al final del archivo.

En estos tres grupos se pueden encontrar tres tipos de elementos diferentes:

1. Elemento tipo vector, este elemento almacena cuatro valores de punto flotante que servirán para determinar un color en el formato RGBA (Rojo, Verde, Azul y Alfa). El primer valor del elemento determinará la cantidad de rojo, el segundo la cantidad de verde, el tercero la cantidad de azul y el último el porcentaje de alfa (o transparencia). El rango de estos números van del 0.0 al 1.0. El formato en que deben ser introducidos los datos es el siguiente:

   <elemento> = <valor1>; <valor2>; <valor3>; <valor4>;

2. Elemento tipo *boolean*, este elemento sólo acepta dos valores `true` o `false`, y determinará la presencia o ausencia de cierto elemento gráfico al momento de la renderización. El formato en que deben ser introducidos los datos es el siguiente:

   <elemento> = <true | false>;

3. Elemento tipo flotante, este elemento acepta un valor de tipo flotante, y determinara el tamaño de los elementos gráficos al momento de la renderización. El formato en que deben ser introducidos los datos es el siguiente:

   <elemento> = <valor>;

Estructura

La estructura se puede encontrar dentro del archivo de cabecera `GlobalFunctions.h` en la ruta: `Arboles/Código Fuente/src/GlobalFunctions.h`. Esta se encuentra definida de la siguiente forma:

```
typedef struct
{
  GLfloat     *lightPos;
  GLfloat     *lightPosMirror;
  GLfloat     *noLight;
  GLfloat     *lowLight;
  GLfloat     *brightLight;
  GLfloat     *groundColor;
  GLfloat     *backGroundColor;
  gboolean    *isHLV;
  gboolean    *isFont;
  GLfloat     *elementColor;
  GLfloat     *fontColor;
  GLfloat     *limitColor1;
  GLfloat     *limitColor2;
  GLfloat     *linkColor;
  GLfloat     *biggerColor;
  GLfloat     *smallerColor;
  gboolean    *showLimits;
  GLfloat     sortWidth;
  GLfloat     treeWidth;
  GLfloat     sortSpacing;

}Settings;
```

Analizador

El *analizador* es el encargado de leer los grupos y elementos del archivo de configuración, esto lo hace de la siguiente forma:

1. Busca un grupo dentro del archivo de configuración

2. Cuando encuentra el grupo lee un elemento del mismo y lo guarda en la estructura.

La sección de código que manda a llamar al *analizador* es la siguiente:

```
if((widgets.conf = load_config_file((gchar*)"config", &keyfile, &error)) == NULL)
  {
    g_print("Error: %s\n", error->message);
    g_print("Loading default values...\n");
    g_error_free(error);
    error = NULL;

    widgets.conf = load_config_file((gchar*)".defaults", &keyfile, &error);
  }
  else
  {
    g_print("\"config\" file loaded.\n");
  }
```

Una parte importante que se puede observar en este código es que cuando por alguna razón hubo una falla al leer el archivo de configuración “config” se vuelve a llamar al *analizador* con

un archivo oculto llamado “`.defaults`”. Este archivo “`.defaults`” es una copia del archivo “config” que permanece oculta y nunca es modificada, de este modo aunque se tenga un archivo “config” corrupto se garantiza que la aplicación siempre será ejecutada.

### Seleccionando estructuras tipo árbol sobre la interfaz gráfica

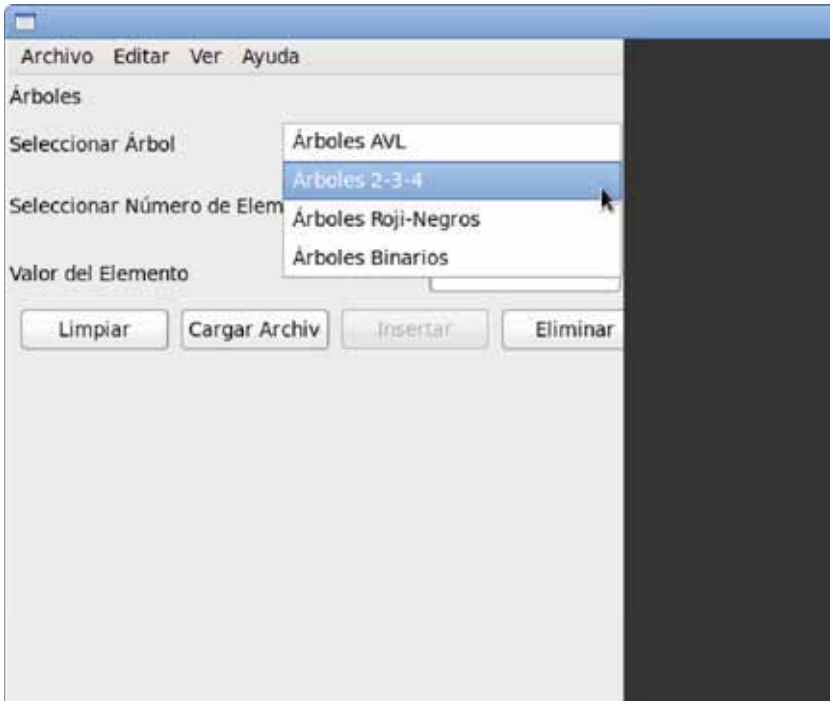



*Figura 7: Cuadro combinado de árboles*

El *cuadro combinado* de árboles permite seleccionar el tipo de árbol que se va a ejecutar, y para saber cuál es algoritmo que se ha seleccionado se hace uso de la siguiente sentencia:

```
select = gtk_combo_box_get_active(GTK_COMBO_BOX (widgets->TreeSelection));
```

Esta sentencia almacena en la variable `select` un número entero que va desde el 0 hasta el 3, el valor 0 corresponde al primer árbol en el *cuadro combinado* (de arriba a bajo) y el número 3 corresponde al último de estos, esto se puede apreciar de forma clara en la Figura 7.

### Valor del elemento

El valor del elemento se debe de teclear en el *entrada de texto* llamada “Valor del Elemento”, como se muestra en la Figura 8.

La parte del código que se encarga de obtener el valor de la *entrada de texto* es la siguiente:

```
const gchar *text = gtk_entry_get_text(GTK_ENTRY(widget));

 if(text == NULL)
    return 0;

  i = 0;
  while(i < gtk_entry_get_text_length(GTK_ENTRY(widget)))
    if(!isdigit(text[i++]))
      return -1;

  value = atoi(text);
```

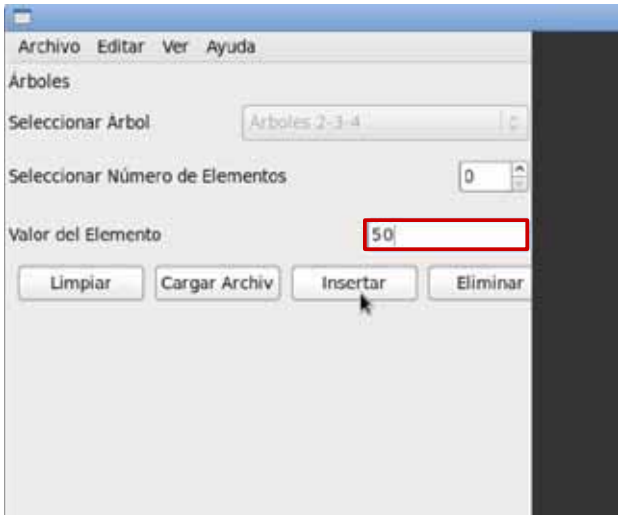



*Figura 8: Valor del elemento*

Esta sección de código realiza dos cosas de suma importancia:

1. Verifica que cada caracter que se encuentra en la *entrada de texto* sea un número, y

2. Ya que ha verificado que todos los caracteres son números los convierte en un valor entero mediante la función `atoi()` y almacena el valor obtenido en la variable `value`.

### Insertando elementos

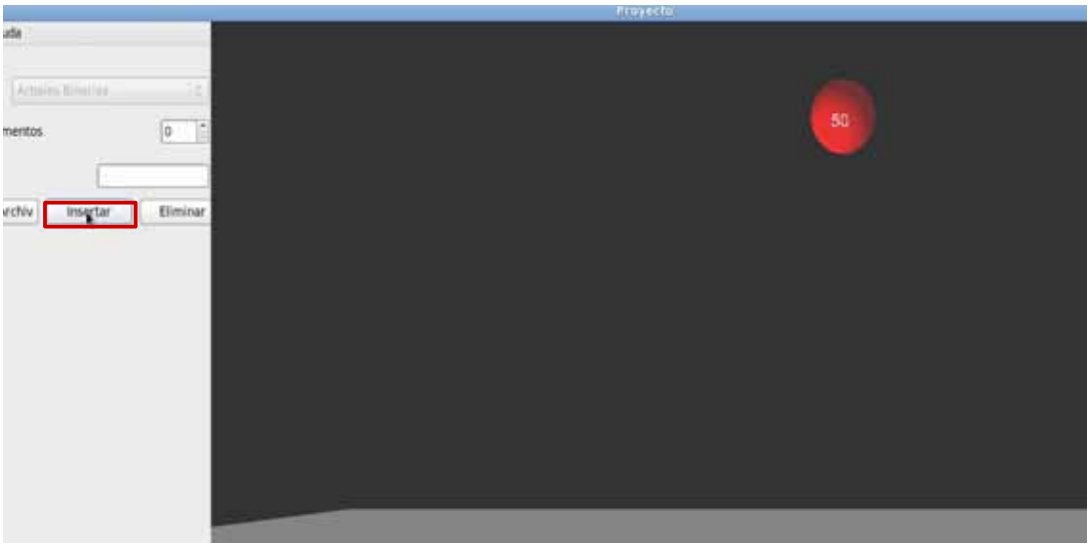



*Figura 9: Insertar elementos*

El botón “Insertar” (vease Figura 9) es el encargado de insertar elementos en la interfaz gráfica y como se puede observar en la Tabla 3 el evento `clicked` está asociado con la *llamada de retorno* `on_TreeInsertButton_clicked.` Esta *llamada de retorno* hace uso de los datos obtenidos en las secciones **valor del elemento** para poder insertar un elemento gráfico dentro de la interfaz.

## Manipulación gráfica de la estructura tipo árbol

Cuando la *llamada de retorno* del botón Insertar es llamada es el momento en el que comienza la manipulación gráfica del árbol, para llevar acabo dicha manipulación se necesitan vectores y en algunos casos arreglos de vectores, y además un vector principal especial.

### Vector principal

Ele vector principal es un vector de estructuras tipo `TreeElement`, este tipo especial de estructura contiene datos como: valor del elemento, posición del elemento en el espacio de coordenadas, color del elemento, el indice del nodo padre, componentes de ligas (esta a su vez es otra estructura tipo `Link`) la estructura es la siguiente:

```
typedef struct
{
   GLfloat      positionf[3];
   GLint        ID;
   GLint        val;
   GLint        father;
```

```
   Link         left;
   Link         right;
   Color        *color;
   GLint        level;
   GLint        indColor;
   GLint        nodoColor;
   GLint        indCubo;
}TreeElement;
```

El vector principal está definido de la siguiente forma:

```
TreeElement  tree_vect[50];
GLint       tree_index;
```

Este vector es llenado con datos cada vez que en la interfaz se agrega un elemento nuevo. Como se menciono con anterioridad además de guardar la información del valor del elemento, guarda otro tipo de información, la cual se explicará en secciones posteriores.

**Matriz de cambios**

En esta matriz será almacenada información relevante al momento que se ejecuten de forma lógica los algoritmos de ordenamiento. Está definida como:

Este vector almacena información acerca de los cambios que sufre el árbol al momento de su ejecución lógica, e interactua directamente con el vector principal para que estos cambios se vean reflejados al momento de la animación.

```
TreeElement    change_vecTree[50];
GLint          change_indexTree;
```

**Reproducción de la animación**

Para iniciar con la animación de una estructura tipo árbol, previamente se debió seleccionar alguna de las variantes de arboles que se quiera ejecutar, e introducir un valor para el elemento a insertar.

Una vez validada la estructura que se quiere y el valor a insertar la llamada de retorno del botón insertar crea un hilo adicional sobre el que se esta ejecutando y llama a una función llamada `InsertBinary()` , dependiendo del tipo de árbol que se haya seleccionado:

```
g_thread_create (InsertBinary, widgets, TRUE, NULL);
```

Esta función lo primero que hace es realizar la inserción en el árbol lógico, invoca una función llamada *`InsertarAB()`*, la cual realiza la inserción y retorna una variable de tipo *`tarbol`* que no es otra cosa mas que el nodo recién insertado, durante la ejecución lógica del

árbol se hicieron ademas de lo obvio que es la inserción como tal, una serie de cálculos tales como las coordenadas que ocupara el nodo en el espacio geométrico, también se determinó un dato importante para efectos de las animaciones como lo es el padre de el nodo, se determina si un nodo tiene ligas, es decir, si tiene un hijo derecho o hijo izquierdo, etc.

De tal forma que toda esta información devuelta por la función InsertaAB(), es copiada al vector principal:

```
aux= insertarAB(&(widgets->tree),widgets->value, 0,100,1,-1);

widgets->tree_vect[widgets->tree_index].val= (*aux)->clave;
widgets->tree_vect[widgets->tree_index].positionf[0]= (*aux)->x;
widgets->tree_vect[widgets->tree_index].positionf[1]= (*aux)->y;
widgets->tree_vect[widgets->tree_index].right.enable = (*aux)->right;
widgets->tree_vect[widgets->tree_index].left.enable= (*aux)->left;
widgets->tree_vect[widgets->tree_index].left.size = -1.0;
widgets->tree_vect[widgets->tree_index].right.size = -1.0;
widgets->tree_vect[widgets->tree_index].indCubo = 0;
widgets->tree_vect[widgets->tree_index].nodoColor = 1;
```

Posterior a esto se hace uso de un vector auxiliar llamado *path_vec[ ]*, el cual contiene de manera cronológica los indices de elementos dentro del vector principal, y que representan la ruta que sigue un nodo cuando es insertado. Este vector es llenado mediante la función *get_path()*:

```
if (widgets->tree_index > 0)
   {
     w = search_father_index ((*aux)->p, widgets->tree_vect);
     widgets->tree_vect[widgets->tree_index].father = w;
   }
   else
     widgets->tree_vect[widgets->tree_index].father = (*aux)->p;

get_path(widgets->tree_vect[widgets->tree_index].father, widgets);

widgets->tree_index ++;
widgets->path_vect[widgets->path_index] = widgets->tree_index;
```

Previamente se determino a nivel de la matriz quien es el padre de determinado elemento, ya que a nivel lógico cada nodo tiene como información de su padre su llave, y en el vector principal necesitamos el indice de la posición que ocupa en dicho vector.

Una vez listo el vector que traza la ruta de nuestro nodo, debemos crear nuevamente un *hilo* adicional al ya creado por el evento del botón Insertar, el cual ejecutara la función *th_path()*:

```
th_id = g_thread_create(th_path, widgets, TRUE, NULL);
```

**Función *th_path()***

Esta función es importante ya que funciona de la mismo forma para todas las estructuras que nos ocupan en este proyecto, básicamente dentro de esta función se recorrerá el vector con la ayuda de un ciclo *for,* cada uno de los valores del vector *path_vect[ i ]* se utilizaran como coordenadas origen ( i ) y destino ( i + 1):

```
void *th_path(gpointer data)
{
  GLint i;
  SortTree *widgets = (SortTree *) data;
  TreeElement aux;
  aux.positionf[0] = widgets->tree_vect[widgets->tree_index-1].positionf[0];
  aux.positionf[1] = widgets->tree_vect[widgets->tree_index-1].positionf[1];
  for(i = 0; i < widgets->path_index; i++)
  {
    if((i+1) == widgets->path_index)
    {
      widgets->tree_vect[widgets->tree_index].positionf[0] = aux.positionf[0];
      widgets->tree_vect[widgets->tree_index].positionf[1] = aux.positionf[1];
      trace_path_anim(widgets->path_vect[i], widgets->tree_index, widgets);
    }
    else
              trace_path_anim(widgets->path_vect[i],  widgets->path_vect[i+1],
widgets);
  }

}
```

**Función Trace*_path_anim()***

Esta función como tal es la que realiza la animación

```
void trace_path_anim(GLint source, GLint target, SortTree *widgets){
...
}
```

podemos tener la seguridad de que el ultimo elemento en nuestro vector principal fue el ultimo elemento en insertarse, o sea que es el elemento que debe hacer un recorrido hasta llegar a su posición dentro del árbol,  las coordenadas finales de este elemento ya las conocemos y las almacenamos temporalmente en una variable *aux* de tipo *TreeElement,* tal como se muestra en la función *th_path()*

```
   aux.positionf[0] = widgets->tree_vect[widgets->tree_index-1].positionf[0];
   aux.positionf[1] = widgets->tree_vect[widgets->tree_index-1].positionf[1];
```

de manera que este elemento de nuestro vector principal, lo hacemos cambiar de posición desde la coordenada (150, 0) a (130, 0) cuando el origen es -1 lo cual quiere decir que va

entrando por la raíz, y posteriormente a las coordenadas del elemento *target*, haciendo uso de una ecuación punto pendiente para calcular los puntos por donde debe pasar el elemento.

```
    index = widgets->tree_index - 1;

  if(source == -1)
  {
    widgets->tree_vect[index].positionf[0] = widgets->tree_vect[target].positionf[0];
    y1 = 150.00;
    y2 = widgets->tree_vect[target].positionf[1];

    for(i = y1 ; i >= y2; i -=2.5)
    {
      g_static_mutex_lock (&(widgets->sync.mutex));
      widgets->tree_vect[index].positionf[1] = i;

        if (i <= y2)
            widgets->tree_vect[index].positionf[1]= y2;

      g_static_mutex_unlock (&(widgets->sync.mutex));
      wait_to_end_anim(widgets);
    }

  }
  else
  {
    x1 = widgets->tree_vect[source].positionf[0];
    y1 = widgets->tree_vect[source].positionf[1];
    x2 = widgets->tree_vect[target].positionf[0];
    y2 = widgets->tree_vect[target].positionf[1];
    father = widgets->tree_vect[index].father;

    m = (y1-y2) / (x1-x2);

    angle = atan(m); // en radianes
    angle = (180 * angle) / M_PI; // en grados

    if(father == source)
    {
      if(angle > 0)
      {
        widgets->tree_vect[father].left.ang = (-angle - 90);
    widgets->tree_vect[father].left.enable = 1;
      }
      else
      {
        widgets->tree_vect[father].right.ang = (-angle + 90);
    widgets->tree_vect[father].right.enable = 1;
      }
    }

    widgets->tree_vect[index].positionf[0] = x1;
    widgets->tree_vect[index].positionf[1] = y1;

    //To the left

    if(x1 > x2)
    {

      for(i = x1, j = y1; i >= x2; i-=2.5)
```

```
      {
        g_static_mutex_lock (&(widgets->sync.mutex));

        widgets->tree_vect[index].positionf[0] = i;
        widgets->tree_vect[index].positionf[1] = j = y1 - (m * (x1 - i));
        dist = sqrt(((x1 - i) * (x1 - i)) + ((y1 - j) * (y1 - j)));

    if (i <= x2)
    {
        widgets->tree_vect[index].positionf[0]= x2;
        widgets->tree_vect[index].positionf[1]= y2;
    }

        if(father == source)
          widgets->tree_vect[father].left.size = dist - 20.0;
        g_static_mutex_unlock (&(widgets->sync.mutex));
         wait_to_end_anim(widgets);
      }

    }

    // To the right
    else
    {

      for(i = x1, j = y1; i <= x2; i+=2.5)
      {
        g_static_mutex_lock (&(widgets->sync.mutex));
        widgets->tree_vect[index].positionf[0] = i;
        widgets->tree_vect[index].positionf[1] = j = y1 - (m * (x1 - i));

        dist = sqrt(((x1 - i) * (x1 - i)) + ((y1 - j) * (y1 - j)));

    if (i >= x2)
    {
        widgets->tree_vect[index].positionf[0]= x2;
        widgets->tree_vect[index].positionf[1]= y2;
    }

        if(father == source)
          widgets->tree_vect[father].right.size = dist - 20.0;

        g_static_mutex_unlock (&(widgets->sync.mutex));

     wait_to_end_anim(widgets);
      }

    }

  }
```

Las funciones marcadas realizan una tarea primordial ya que interrumpen momentáneamente el *hilo* principal que constantemente esta haciendo el renderizado y actualizando cualquier cambio en la estructura, la idea es impedir que dos procesos en este caso los hilos accedan al mismo segmento de memoria, o sea , el vector principal.

**¿Cómo funciona la matriz de cambios?**

La matriz de cambios para el caso especifico de las estructuras tipo árbol, funciona para indicar la nueva coordenada a la que se debe desplazar un nodo en el árbol, es ocupada para reflejar las rotaciones cuando se trata de un árbol AVL o árbol roji negro, o cuando se hace una eliminación en cualquiera de los arboles, básicamente después de realizar una operación sobre el árbol lógico, se recorre por completo el árbol y se hace una comparación elemento por elemento entre la estructura lógica y el vector, si se encuentra una diferencia entre las coordenada en alguno de ellos, este elemento es agregado al vector de cambios *widgets->change_vecTree[ ].*

```
aux = insertRB (widgets->tree_RB,&(widgets->tree_RB->root->left),widgets-
                    >value,0,100,1,widgets->tree_RB->root);
   .
   .
   .
       Animación de Recorrido


insertFixUp (widgets->tree_RB, *aux, &vector);

if (vector.index_cambio > 0)
    RB_VectorToWidgets (widgets, &vector);


RB_reordena (widgets->tree_RB, &(widgets)->tree_RB->root->left,0,100,1);

  for (i = 0; i < widgets->tree_index ; i++)
  {
    w = RB_buscarPadre (widgets->tree_RB,widgets->tree_RB->root->left, widgets-
                >tree_vect[i].val);
    if (w > 0)
    {
        t = RB_search_father_index (w,widgets->tree_vect);
        widgets->tree_vect[i].father = t;
    }
    else if (w == -1)
        widgets->tree_vect[i].father = -1;
  }

  if (vector.index_cambio > 0)
  {

    widgets->sync.timeout_ID = gdk_threads_add_timeout (10, timeout, widgets);
    th_id = g_thread_create(RB_th_path_rotacionD, widgets, TRUE, NULL);
    g_thread_join(th_id);
    g_source_remove (widgets->sync.timeout_ID);
    RB_ligas (widgets->tree_RB, widgets->tree_RB->root->left);
    widgets->change_index = 0;
    RB_seek_changes (widgets);
    widgets->sync.timeout_ID = gdk_threads_add_timeout (10, timeout, widgets);
    th_id = g_thread_create(RB_th_path_rotacionS, widgets, TRUE, NULL);
    g_thread_join(th_id);
    g_source_remove (widgets->sync.timeout_ID);

    widgets->sync.timeout_ID = gdk_threads_add_timeout (1, timeout, widgets);
    th_id = g_thread_create(RB_th_link, widgets, TRUE, NULL);
```

```
    g_thread_join(th_id);
    g_source_remove (widgets->sync.timeout_ID);

    widgets->sync.timeout_ID = gdk_threads_add_timeout (1000, timeout, widgets);
    g_source_remove (widgets->sync.timeout_ID);
  }
  else
  {
    RB_ligas (widgets->tree_RB, widgets->tree_RB->root->left);
    widgets->change_index = 0;
    RB_seek_changes (widgets);
    widgets->sync.timeout_ID = gdk_threads_add_timeout (10, timeout, widgets);
    th_id = g_thread_create(RB_th_path_rotacionS, widgets, TRUE, NULL);
    g_thread_join(th_id);
    g_source_remove (widgets->sync.timeout_ID);

    widgets->sync.timeout_ID = gdk_threads_add_timeout (1, timeout, widgets);
    th_id = g_thread_create(RB_th_link, widgets, TRUE, NULL);
    g_thread_join(th_id);
    g_source_remove (widgets->sync.timeout_ID);

    widgets->sync.timeout_ID = gdk_threads_add_timeout (1000, timeout, widgets);
    g_source_remove (widgets->sync.timeout_ID);


  }
```

En el código anterior de puede ver como posterior a la primer animación que corresponde al recorrido del nodo hacia su posición, y como estamos hablando de un árbol roji - negro ahora se requiere validar las propiedades a las que obedece esta estructura la función `insertFixUp()` es la que se encarga de realizar esta operación, esta función ademas de que fue extraída de su ubicación original que era dentro de la función *`insertRB()`*, fue modificada para recibir como parámetro un apuntador a un vector, este vector se va a cargar con elementos si la función realizó rotaciones en la estructura, `if (vector.index_cambio > 0)` en esta parte del código se determina las invocaciones a los hilos encargados de realizar la animación de la re configuración del árbol.

### Función *`RB_th_path_rotacionD()`*

Esta función propiamente realiza la tarea de animar una rotación doble de un árbol, en realidad lo que hace es trasladar los elementos que se encuentran dentro del vector principal *`tree_vect[ ]`* a las coordenadas destino que se encuentran almacenadas en el vector de cambios *`change_vecTree[ ]`*, esto lo hace a través de un ciclo *`while`* el cual se va estar ejecutando mientras el vector de cambio contenga al menos un elemento en el cual su miembro *`change_vecTree[ i ].val != -1`*, como es posible que al re configurar un árbol mas de un nodo debe actualizar su posición, la animación debe simular que los nodos se están moviendo de manera paralela, es importante mencionar que esto no es verdad dentro del ciclo se recorre un elemento y se hace que se mueva una posición adelante del total de su trayecto, terminado ese paso, cede el turno a otro elemento del vector de cambio, y así sucesivamente hasta llegar a su posición final.

```
void *RB_th_path_rotacionD (gpointer data)
{

    SortTree *widgets = (SortTree *) data;
    int i,j,bit;
    bit = 1;

  while (bit == 1)
  {
      for (i = 0; i < widgets->tree_index; i++ )
    for (j = 0; j < widgets->rot_index; j++)
        if (widgets->tree_vect[i].val == widgets->primer_rot[j].val)
        {
       widgets->tree_vect[i].left.enable = 0;
       widgets->tree_vect[i].right.enable = 0;
       RB_trace_rotacionD (i,j,widgets);
        }

      bit = RB_rotar_bit (widgets);
  }

}
```

**Posición en el espacio de coordenadas**

Esta posición esta dada por el vector `positionf[3]`, a la primer posición del vector le corresponde la coordenada X, a la segunda posición la coordenada Y y a la tercera la coordenada Z.

El calculo de las posiciones de cada elemento del árbol se calcula en el momento que se hace la inserción en el árbol lógico:

```
tarbol **insertarAB(tarbol **a, int elem,int x,int y,int nivel,int papa)
{
int offset;

  if (*a == NULL)
  {
    *a = (tarbol *) malloc(sizeof(tarbol));
    (*a)->clave = elem;
    (*a)->x = x;
    (*a)->y = y;
    (*a)->izq = (*a)->der = NULL;
    (*a)->p = papa;
    (*a)->nivel = nivel;
    (*a)->right = 0;
    (*a)->left = 0;
    return a;
  }
  else
  {
    if ((*a)->clave < elem)
```

```
    {

        offset = calcula_offset (nivel);
        (*a)->right = 1;
        insertarAB(&(*a)->der, elem,x + offset,y-30,(nivel+1),(*a)->clave);
    }
    else
    {
      if ((*a)->clave > elem)
    {

        offset = calcula_offset(nivel);
        (*a)->left = 1;
        insertarAB(&(*a)->izq, elem, x - offset ,y-30,(nivel+1),(*a)->clave);

    }
    }
  }

}
```

de manera recursiva se van calculando las coordenadas que ocupara el nodo, una función llamada *`calcula_offset(nivel)`* determina la distancia que habrá desde el el centro de nuestro espacio geométrico hasta cada uno de los nodos, el calculo del *`offset`* esta en función del nivel de profundidad en el árbol en el que se encuentre el nodo.

Para poder representar las animaciones es necesario ejecutar en paralelo un algoritmo que vaya haciendo modificaciones sobre el vector principal y otro algoritmo que vaya dibujando estas modificaciones en la interfaz gráfica. El algoritmo encargado del dibujado se explicará en la sección de **Renderización**, por lo pronto hay que tener en mente que mientras se van haciendo modificaciones sobre el vector principal, estas se irán reflejando en el área de dibujo de la interfaz gráfica.

## Renderización

La renderización es la parte principal y más importante de todo el proyecto, ya que esta permitirá observar de forma visual el funcionamiento de los algoritmos de ordenamiento. La renderización toma partes de todo lo visto en las secciones anteriores y agrega partes nuevas para permitir visualizar las animaciones. Pero, hay que empezar desde el principio.

### Preparación del área de dibujo

El área de dibujo es una parte de la interfaz gráfica donde se visualizarán las animaciones de los algoritmos de ordenamiento, no fue sino hasta dos secciones atrás que se empezó a mencionar esta área de dibujo. Pues bien, como se habrá notado esta área no fue agregada desde el diseño de la interfaz gráfica en la herramienta Glade y para poder agregarla a la interfaz gráfica primero hay que hacer ciertos preparativos:

Obteniendo la resolución de la pantalla

Esto se realiza de la siguiente forma:

```
screen = gdk_screen_get_default();
widgets.screen.screenHeight = screenHeight = gdk_screen_get_height(screen);
widgets.screen.screenWidth = screenWidth = gdk_screen_get_width(screen);
```

Con estas sentencias almacenamos el alto y ancho de la pantalla dentro de la estructura Screen, la cual está implementada de la siguiente forma:

```
typedef struct
{

  GtkWidget   *window;
  GtkWidget   *da;
  GLfloat     screenWidth;
  GLfloat     screenHeight;
  GLfloat     aspect;
  GLfloat     fovy;
  GLfloat     daWidth;
  GLfloat     daHeight;

}Screen;
```

Configurando la capacidad visual de OpenGL

En este punto se verifica que la plataforma sobre la cual va a ejecutarse la aplicación tenga soporte para OpenGL, de lo contrario se mostrara un error en la terminal y la aplicación no se ejecutará. Esto se implementa de la siguiente forma:

```
glconfig = gdk_gl_config_new_by_mode (static_cast<GdkGLConfigMode> (GDK_GL_MODE_RGB  |
                                        GDK_GL_MODE_DEPTH    |
                                        GDK_GL_MODE_DOUBLE));

if (glconfig == NULL)
{
  g_print ("*** Cannot find the double-buffered visual.\n");
  g_print ("*** Trying single-buffered visual.\n");

   /* Try single-buffered visual */
  glconfig = gdk_gl_config_new_by_mode (static_cast<GdkGLConfigMode>  (GDK_GL_MODE_RGB |
                                        GDK_GL_MODE_DEPTH));
  if (glconfig == NULL)
  {
    g_print ("*** No appropriate OpenGL-capable visual found.\n");
    exit (1);
  }
```

```
}
```

Área de dibujo para la escena de OpenGL

En está sección se creará un área de dibujo y se le asignará un tamaño dependiendo de la resolución de la pantalla y el espacio ocupado por la interfaz gráfica desarrollada en Glade.

```
widgets.screen.da = gtk_drawing_area_new ();
gtk_widget_set_size_request (widgets.screen.da, (screenWidth - 425), 500);
```

Posteriormente se le agrega la capacidad para el manejo de OpenGL

```
gtk_widget_set_gl_capability (widgets.screen.da,
                              glconfig,
                              NULL,
                              TRUE,
                              GDK_GL_RGBA_TYPE);
```

Integración del área de dibujo a la interfaz gráfica

Finalmente el área de dibujo es unida con el resto de la interfaz gráfica, para este efecto se utiliza la sentencia:

```
gtk_box_pack_end(widgets.hbox, widgets.screen.da, TRUE, TRUE, 0);
```

la cual agregará al lado derecho de la interfaz el área de dibujo.

OpenGL cuenta con tres eventos principales que permiten hacer la renderización de los elementos en el área de dibujo, estos eventos son: “`configure_event`”, “`realize`” y “`expose_event`”.

**Evento “realize”**

Este evento tiene asociada una *llamada de retorno* con el mismo nombre y aquí se definen algunos parámetros del comportamiento que tendrá la renderización en OpenGL.

Color de fondo

El color de fondo está dado por la siguiente sentencia:

```
glClearColor(widgets->conf->backGroundColor[0], widgets->conf->backGroundColor[1],
            widgets->conf->backGroundColor[2], widgets->conf->backGroundColor[3]);
```

“Cull back” de los polígonos

El “*cull back*” permite decidir si las caras traseras de los polígonos serán dibujadas, generalmente es una buena practica deshabilitar el “*cull back*” para ahorrar recursos en hardware e incrementar el desempeño de la aplicación. Esto está dado por las siguientes sentencias:

```
glCullFace(GL_BACK);
glFrontFace(GL_CCW);
glEnable(GL_CULL_FACE);
glEnable(GL_DEPTH_TEST);
glEnable(GL_MULTISAMPLE_ARB);
```

Configuración de las luces

Las luces son una parte importante dentro de una escena en OpenGL ya que dependiendo de los parámetros de estas los elementos renderizados en la escena se pueden ver de forma adecuada o no. Esta configuración está dada por:

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, widgets->conf->noLight);
glLightfv(GL_LIGHT0, GL_AMBIENT, widgets->conf->lowLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, widgets->conf->brightLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, widgets->conf->brightLight);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

**Evento “configure_event”**

Este evento tiene asociado una *llamada de retorno* con el mismo nombre y cada vez que la ventana de la interfaz gráfica sufre una modificación, como pudiera ser un re-dimensionamiento, el evento es lanzado y la *llamada de retorno* es llamada.

Reiniciando el sistema de coordenadas

Es el primer paso que se lleva a cabo para poder hacer la renderización dentro del área de dibujo, este reinicio está dado por:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Seleccionando el “clipping volume (volumen de recorte)

El “volumen de recorte” es la forma en que se observará el sistema de coordenadas en el área de dibujo, existen dos opciones para visualizar el sistema de coordenadas: en perspectiva u ortogonal.

La vista que se utilizara para la renderización de los elementos es la vista en perspectiva y está dado por:

```
gluPerspective(widgets->screen.fovy, widgets->screen.aspect, 1.0f, 500.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

**Evento “expose”**

La *llamada de retorno* asociada al evento “expose” es la encargada de hacer el renderizado de todos los elementos que conforman la escena dentro del área de dibujo. Cada vez que la *llamada de retorno* del evento “expose” es ejecutada realiza cuatro acciones diferentes: borrado de todos los elementos en el área de dibujo, renderiza los elementos de forma invertida (para dar un efecto de reflexión, renderiza el piso sobre los elementos invertidos y finalmente renderiza los elementos de forma normal sobre el área de dibujo.

Borrado de elementos

Este proceso es importante ya que cada vez que se cambia alguna propiedad de un elemento del vector principal o del vector de límites hay que borrar el renderizado anterior y volver a hacerlo con las nuevas propiedades del elemento para dar el efecto de animación.

Este proceso está dado por las siguientes sentencias:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);

glPushMatrix();

  // Reset the coordinate system before modifying

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  // Set the clipping volume

  gluPerspective(widgets->screen.fovy, widgets->screen.aspect,
                 1.0f, 500.0f);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

glPopMatrix();
```

Renderizando los elementos invertidos

Para dar un efecto de reflexión a los elementos que conforman la escena de OpenGL se utilizan las siguientes sentencias:

```
glLightfv(GL_LIGHT0, GL_POSITION, widgets->conf->lightPosMirror);

glPushMatrix();

  glFrontFace(GL_CW);
  glScalef(1.0f, -1.0f, 1.0f);
  glTranslatef(0.0f, -35.0f, 0.0f);
  widgets->mirror = TRUE;

  DrawWorld(widgets);

glPopMatrix();
```

La primer sentencia invierte la posición de la luz, de modo que al dibujar los elementos estos sean iluminados por la misma y de este modo se vean en el área de dibujo. Las siguientes sentencias invierten el sistema de coordenadas de modo que al dibujar los elementos de la escena estos tengan ese efecto de reflexión

La función DrawWorld() es la que se encarga de dibujar cada uno de los elementos en el área de dibujo; esta función será analizada más adelante.

Renderizando el piso

En esta parte el piso es dibujado con una transparencia encima de los elementos con el efecto de reflexión, y está dado por:

```
glDisable(GL_LIGHTING);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

DrawGround(widgets->conf);

glDisable(GL_BLEND);
glEnable(GL_LIGHTING);
```

Las primeras tres sentencias deshabilitan las luces, habilitan el "*blend*" (*mezclado*) que es el que permite tener ese efecto de transparencia y a su vez lo configuran. Posteriormente el piso se renderiza en el área de dibujo y finalmente se deshabilita el *mezclado* y se habilitan las luces.

Renderizando los elementos normalmente

Finalmente se hace el renderizado de los elementos de forma normal, para esto primero hay que restaurar las luces que estaban invertidas, mediante la sentencia:

```
glLightfv(GL_LIGHT0, GL_POSITION, widgets->conf->lightPos);
```

y después se hace el renderizado de los elementos mediante las sentencias:

```
glPushMatrix();

  widgets->mirror = FALSE;

  DrawWorld(widgets);

glPopMatrix();
```

**Función DrawWorld()**

Como se mencionó con anterioridad esta función es la que se encarga de renderizar todos y cada uno de los elementos que conforman la escena en OpenGL. Consiste de tres partes principales que son: el renderizado de fuentes, el renderizado de elementos y el renderizado de límites.

Renderizando las fuentes

Para renderizar las fuentes dentro del área de dibujo se hace uso del vector principal y se utilizan las coordenadas de cada elemento, las cuales están almacenadas en el vector `positionf[]`, y también se toma valor (entero) de cada elemento para renderizarlo en forma de fuente en el área de dibujo.

Para esto se usa un ciclo `for()` que recorra todo el vector principal.

```
for(i = 0; i < widgets->i; i++)
```

Posteriormente hay que trasladar el sistema de coordenadas a la posición almacenada en cada elemento del vector:

```
glTranslatef(widgets->vector[i].positionf[0], 0.0f,
            -widgets->vector[i].positionf[2]);
```

Después se hace otro ligero traslado a la izquierda, para que cuando las fuentes sean renderizadas queden en el centro del elemento:

```
glTranslatef(-2.5f, 1.0f, 0.0f);
```

Y finalmente se hace el renderizado de las fuentes mediante la sentencia:

```
do_display(widgets->vector[i].val, widgets->conf, widgets->font);
```

Renderizando los elementos

Al igual que con las fuentes, para el renderizado de los elementos se utilizará un ciclo `for()` que recorra el vector principal de principio a fin:

```
for(i = 0; i < widgets->i; i++)
```

Después se debe seleccionar que color se empleará para el renderizado de los elementos, si el color RGB o el color HLV:

```
if(widgets->conf->isHLV[0])
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
               widgets->vector[i].color->hlvColor);
else
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
               widgets->vector[i].color->rgbColor);
```

Ahora se debe trasladar el sistema de coordenadas a las posiciones almacenadas por cada uno de los elementos del vector principal:

```
glTranslatef(widgets->vector[i].positionf[0],
             widgets->vector[i].positionf[2], 0.0f);
```

Y finalmente se renderiza la figura geométrica, que en este caso será un cono:

```
gdk_gl_draw_cone(TRUE, widgets->vector[i].width,
                       widgets->vector[i].height + 10, 40, 40);
```

**Timeouts (tiempos de espera)**

Los *tiempos de espera* son funciones que cada cierto intervalo de tiempo generan eventos *expose*, y este tipo de función es la que se utilizó en la sección **Manipulación gráfica de las estructuras tipo árbol.**

El *tiempo de espera* que se agregó fue:

```
widgets->sync.timeout_ID = gdk_threads_add_timeout (10, timeout, widgets);
```

Este *tiempo de espera* tiene asociado una función que es la que se estará ejecutando cada cierta cantidad de tiempo, ene este caso 10 milisegundos y gracias a esto cada vez que se hagan modificaciones sobre el vector principal el área de dibujo se estará actualizando

constantemente permitiendo ver la animación de los elementos de los algoritmos de ordenamiento.

# Ejecutando la aplicación

## Ejecución desde la terminal

Como primer paso se debe abrir una terminal en el sistema operativo, posteriormente hay que moverse a la carpeta principal del proyecto, la cual es: `Arboles/Código Fuente/src`. Posteriormente hay que ejecutar el proyecto mediante el siguiente comando (véase la Figura 10):

$./Proyecto

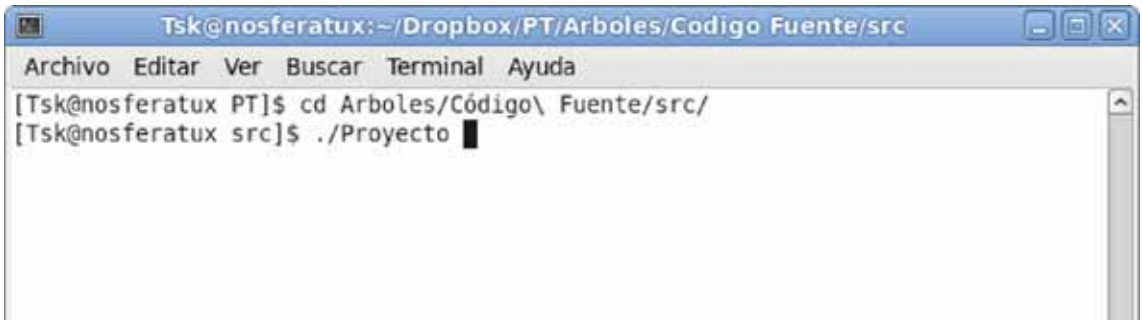



*Figura 10: Ejecución desde la terminal*

La ejecución de este comando mostrará en pantalla la ventana principal de la aplicación (véase la Figura 11).

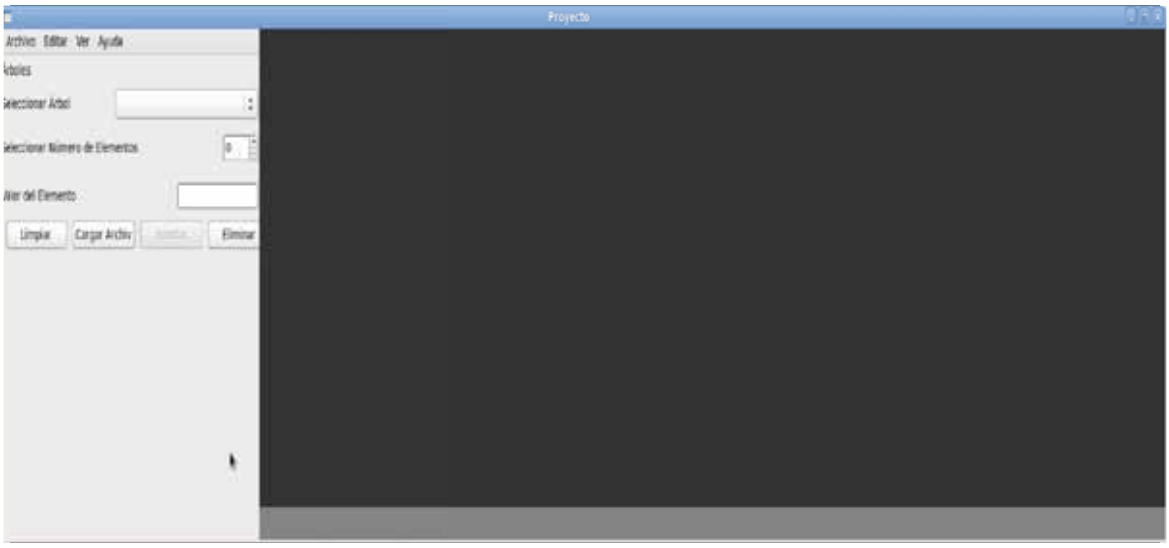



*Figura 11: Ventana principal*

## Seleccionando estructura tipo árbol

En en menú “Ver” se hace un clic sobre la opción arboles (véase la  Figura 12).

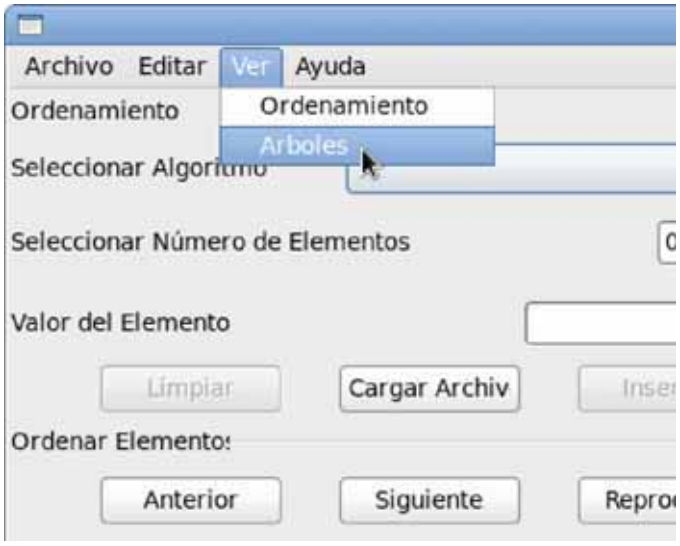



*Figura 12: Selección de estructuras tipo árbol*

Una vez que se ha seleccionado el tipo de árbol  que será ejecutado  procedemos a insertar elementos.

## Insertar elementos en el árbol

Tecleamos el valor del elemento en la entrada de texto, se presiona  el botón insertar y en ese momento se visualiza ya la animación en la pantalla, siempre se inicia con el elemento bajando desde la parte superior del área de dibujo(véase Figura 13).

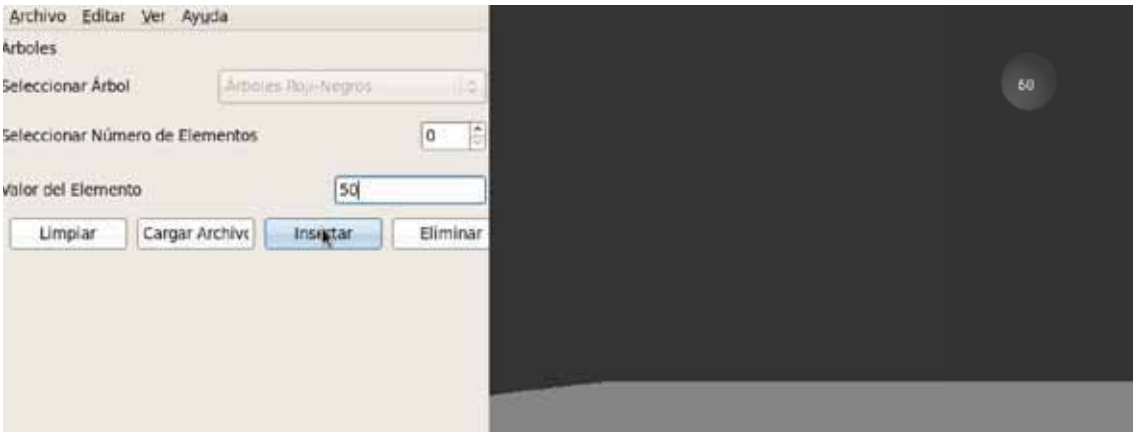



*Figura 13: Insertando elementos en la interfaz gráfica*

Para insertar mas elementos se realizan los mismos pasos(véase la Figura 14).

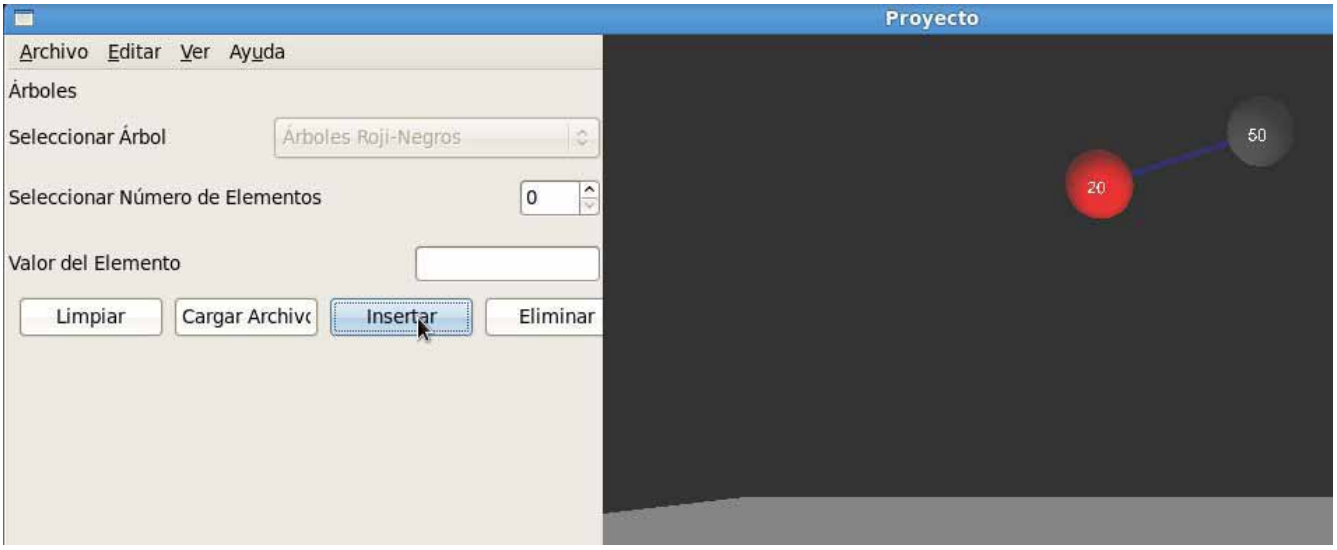



*Figura 14: Insertando elementos en la interfaz gráfica b*

## Eliminación de elementos en el árbol

Para eliminar un elemento del árbol, tecleamos el valor de dicho elemento en la entrada de texto, y posteriormente presionamos el botón Eliminar (véase Figura 15)

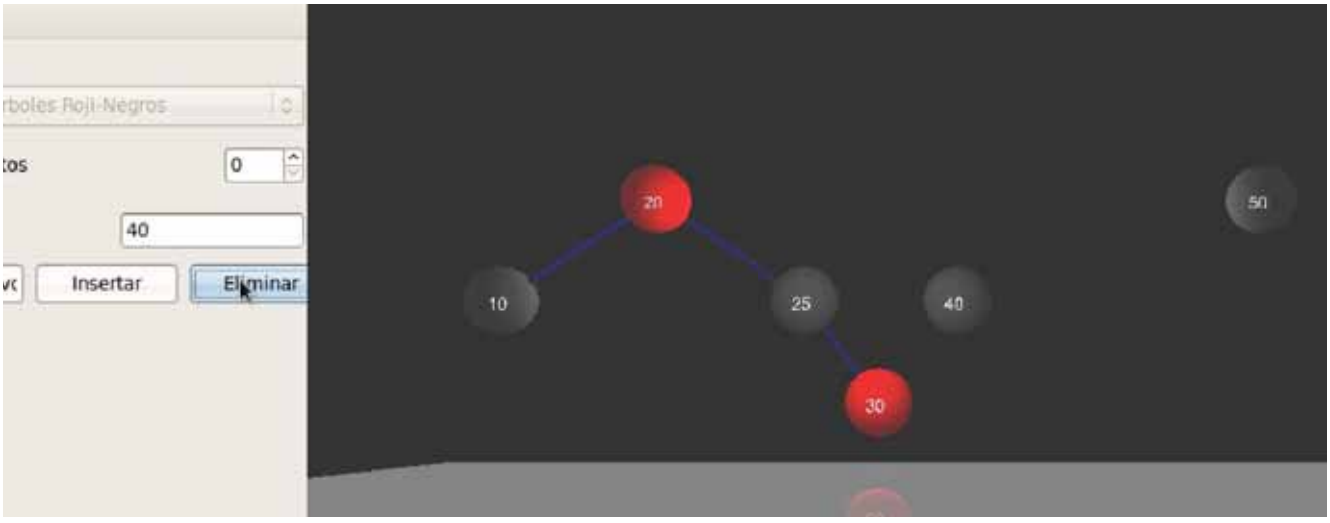



*Figura 15: Eliminar elementos del árbol.*

## Limpiando los elementos de la interfaz gráfica

Para limpiar los elementos que son mostrados dentro de la interfaz gráfica se debe de hacer un clic sobre el *botón* “Limpiar”, esta acción borrará los elementos mostrados en la interfaz gráfica y permitirá agregar nuevos elementos por cualquiera de los métodos antes mencionados (véase la Figura 16).

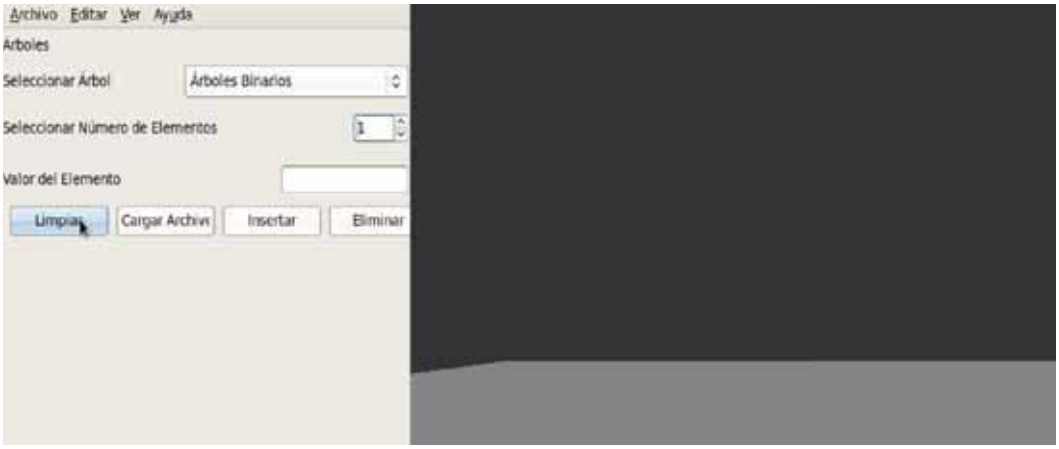



*Figura 16: Limpiando los elementos de la interfaz gráfica*

# Conclusiones

En conclusión haber realizado este proyecto implicó poner en práctica muchas habilidades adquiridas a través de la carrera, primeramente el desarrollar bajo un ambiente Linux no resulto tan sencillo se requirió investigación adicional para instalar aplicaciones y preparar el ambiente propicio para el desarrollo de esta aplicación.

Ademas de esto se requirió documentarse en el uso de la herramienta propia para la implementación de las animaciones OpenGL, que aunque resultan sencillas a la vista, necesitaron un conocimiento no solo de conceptos de programación, en algunos casos requirieron incluso conocimientos matemáticos.

La implementa de las estructuras lógicas fue una tarea mas que se tenia que resolver, aunque es cierto que se puede disponer de códigos en la red, había que adecuar estos algoritmos a necesidades propias del proyecto.

Y principalmente se consiguió cumplir el objetivo de este proyecto, e cual era describir el comportamiento de una estructura de datos tipo árbol, en algunas de sus variantes las cuales fueron implementadas en esta aplicación, con ayuda de la API OpenGL se implementaron pequeñas rutinas de animación que obedeciendo a algoritmos de arboles binarios, AVL, etc. Permitieron observar detalles que en ocasiones resultan transparentes para el estudio de estas estructuras.

# Manual de instalación

## Objetivo

El objetivo del presente documento es mostrarle al usuario los requerimientos de hardware y software necesarios para poder ejecutar el proyecto de una forma correcta. De igual forma proporciona una guía para instalar las dependencias necesarias para poder compilar el proyecto.

## Requisitos de hardware

Los requisitos mínimos de hardware son los siguientes:

- Procesador 1.8 Ghz
- Memoria física 512 MB
- Tarjeta de video 32 MB

## Requisitos de software

El proyecto a sido desarrollado y probado sobre las distribuciones de Linux: Fedora 12 versión 32 bits y Fedora 20 en versión 64 bits.

Para la correcta compilación y ejecución del proyecto es necesario tener instalados los siguientes paquetes:

- *glib* y *glib-devel*
- *gtk+* y *gtk+-devel*
- *gtkglext-devel* y *gtkglext-libs*
- *glib2* y *glib2-devel*
- *ftgl* y *ftgl-devel*
- *mesa-libGL*, *mesa-ligGL-devel*, *mesa-libGLU* y *mesa-libGLU-devel*
- *gtk2* y *gtk2-devel*

## Instalación de paquetes

Para la instalación de los paquetes necesarios para la compilación y ejecución del proyecto se utilizará la herramienta ***yum*** que viene integrada en las distintas versiones de la distribución de Fedora. Para otras distribuciones de Linux se deberán utilizar las herramientas pertinentes para la búsqueda e instalación de paquetes, como es el caso de ***apt-get*** y/o ***aptitude*** para las distribuciones Debian y Ubuntu.

Para poder instalar cualquier paquete se debe hacer con la cuenta de super usuario (***su***) o en su caso con una cuenta con permisos de super usuario (***sudo***).

Si de desea instalar como super usuario se debe abrir una **terminal** y teclear:

```
$su
```

Se introduce la contraseña de super usuario y el prompter deberá de cambiar a:

```
#
```

Posteriormente se ejecuta el comando:

```
#yum install glib glib-devel gtk+ gtk+-devel gtkglext-devel gtkglext-libs glib2 glib2-devel ftgl  ftgl-devel mesa-libGL mesa-ligGL-devel mesa-libGLU mesa-libGLU-devel gtk2 gtk2-devel
```

En el caso de que yum pida instalar algunas dependencias se deben de aceptar las mismas y continuar con la instalación.

Si la instalación se hace con un usuario con permisos de super usuario entonces se ejecuta el siguiente comando:

```
#sudo yum install glib glib-devel gtk+ gtk+-devel gtkglext-devel gtkglext-libs glib2 glib2-devel ftgl  ftgl-devel mesa-libGL mesa-ligGL-devel mesa-libGLU mesa-libGLU-devel gtk2 gtk2-devel
```

Inmediatamente solicitará la contraseña del usuario (no la de super usuario), se introduce y se continúa con la instalación como en el paso anterior.

## Compilación

Para poder compilar el proyecto se debe acceder desde una terminal al directorio *src/*. Este directorio se encuentra en la carpeta principal del proyecto: Arboles. Para tal efecto se teclea desde la terminal:

```
$cd Arboles/Código Fuente/src
```

Posteriormente se compila el proyecto mediante el comando *make*:

```
$make
```

Se debe esperar a que el comando termine su ejecución y visualizar que no muestre ningún error en el proceso. En dado caso que el proceso de compilación arroje un error se debe verificar que todos los paquetes listados en la parte de requerimientos de software se encuentren correctamente instalados, para este efecto se puede usar el siguiente comando:

```
$yum list <nombre_paquete>
```

El resultado de este comando mostrará si el paquete se encuentra correctamente instalado, por ejemplo:

```
$ yum list gtk+
Loaded plugins: langpacks, presto, refresh-packagekit
Installed Packages
gtk+.i686                          1:1.2.10-71.fc15            @fedora
```

### Ejecución

Para ejecutar la aplicación mediante una terminal se accede al directorio Arboles/Código Fuente/src

```
$cd Arboles/Código Fuente/src
```

Y se ejecuta el siguiente comando:

$./Proyecto

Este comando desplegará la interfaz gráfica del proyecto.

## Manual de usuario

### Objetivo

El presente documento tiene como objetivo principal permitir al usuario familiarizarse con la interfaz gráfica del proyecto, así como dar explicación de cada una de las partes que conforman a la interfaz gráfica. Cabe señalar que el alcance de este manual se enfoca solamente en la sección de árboles.

### Ejecutando la aplicación

Para ejecutar  la aplicación mediante una terminal se accede al directorio Arboles/Código Fuente/src

$cd Arboles/Código Fuente/src

Y se ejecuta el siguiente comando:

$./Proyecto

Este comando lanzara la aplicación del proyecto.

## Ventana principal

En la Figura 17 se puede apreciar la ventana principal que se muestra al ejecutar el proyecto.

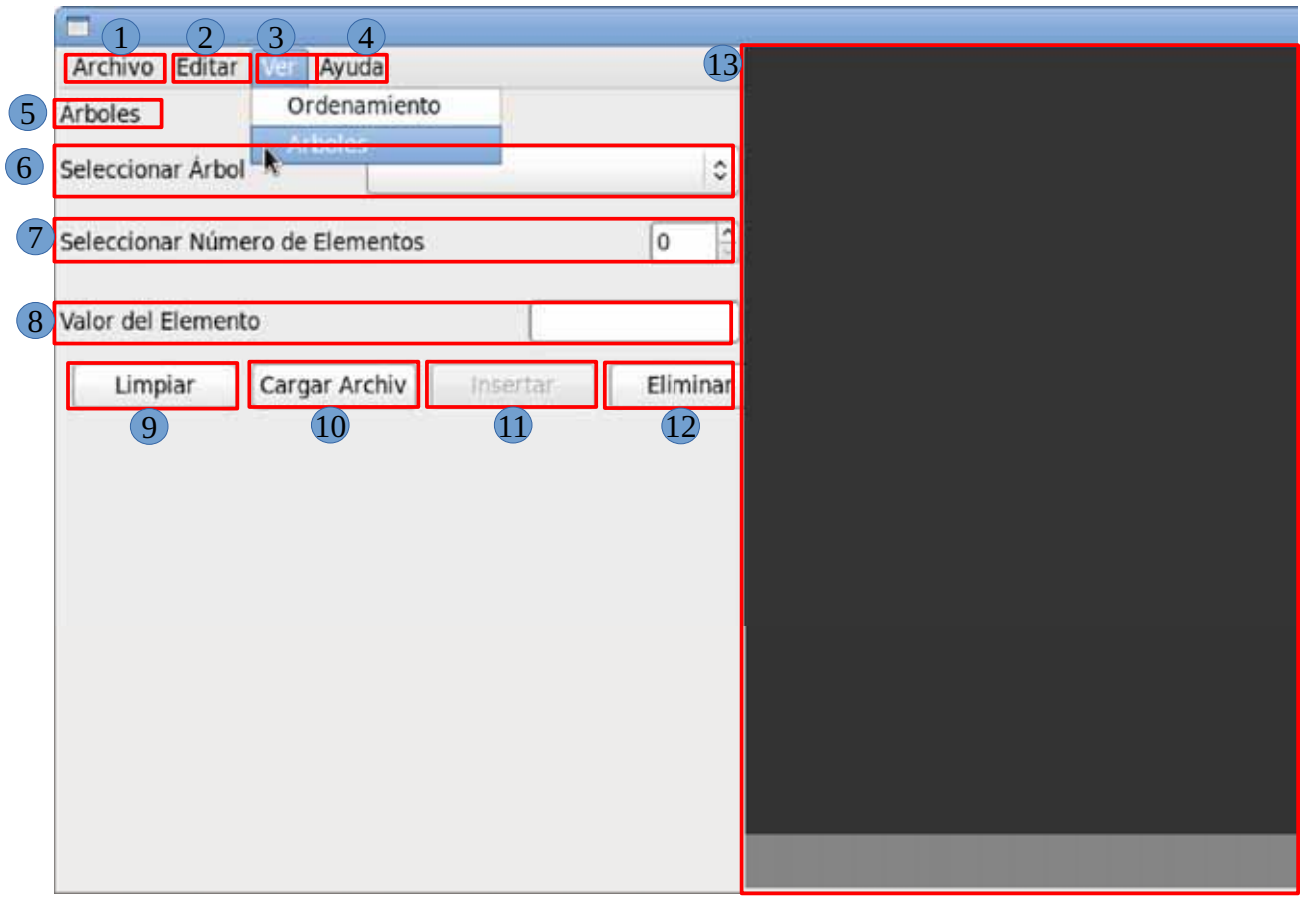



*Figura 17: Ventana Principal*

Esta ventana consta de las siguientes partes:

**1**, **2** y **4** son menús que no tienen contenido alguno y estan reservados para futuras mejoras en la interfaz gráfica.

**3** El menú *Ver* permite seleccionar entre algoritmos de ordenamiento y estructura tipo árbol.

**5** Es la sección reservada para las estructuras tipo árbol.

**6** Es un combo que permite seleccionar uno de los 4 tipos de arboles que se encuentran disponibles para la aplicación.

**7** Permite Seleccionar el numero de elementos que serán insertados en la estructura.

**8** Es un campo de texto en el que se coloca el valor del elemento que sera insertado. Este campos solo acepta valores numéricos enteros.

**9** El botón *Limpiar* permite borrar los elemento insertados en la esctructura, y limpia el área de dibujo.

**10** El botón cargar archivo permite introducir elementos previamente almacenados en un archivo.

**11** El botón Insertar inserta en la interfaz gráfica el elemento especificado en **8**.

**12** Ele botón Eliminar elimina un elemento de la estructura.

**13** Es el área donde se visualizaran los elementos insertados en la interfaz gráfica.

## Menú *Ver*

Al pulsar el menú Ver se pueden visualizar las opciones de Ordenamiento y Árboles que la interfaz gráfica contiene, estos se puede observar de forma clara en la Figura 18.

Al seleccionar la opción Ordenamiento o Árboles la interfaz se comportara de forma distinta en las secciones **5** información diferente.

El alcance del presente manual es para describir el funcionamiento de las estructuras tipo árbol, por lo cual se deberá seleccionar *Arboles*.

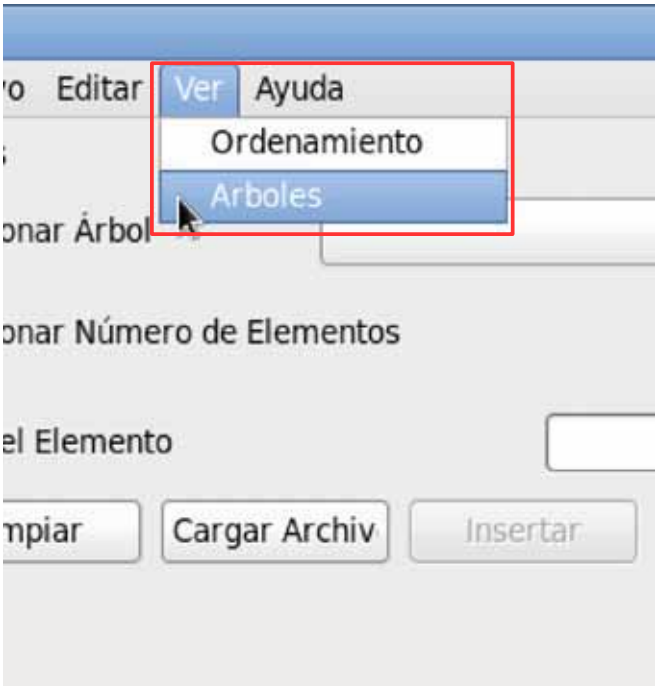



*Figura 18: Menú Ver*

## Seleccionando estructura tipo árbol

La aplicación cuenta con 4 variaciones de árboles, al hacer clic sobre el combo de la sección *`Seleccionar Árbol`* se desplegara una lista con los 4 tipos de arboles disponibles. En la Figura 19 se muestra la lista de las estructuras de la interfaz gráfica, y para seleccionar alguno de ellos solo basta con hacer un clic sobre él.

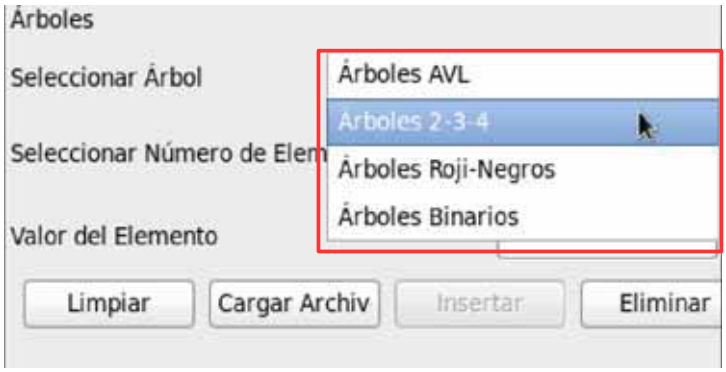



*Figura 19: Lista de estructuras tipo árbol*

## Introduciendo elementos

En el campo de texto correspondiente a la sección *Valor del Elemento* se debe teclear un valor numérico y entero, dar clic en el botón Insertar y automáticamente se ejecutara la animación de inserción en el árbol,  como se muestra en la Figura 20.

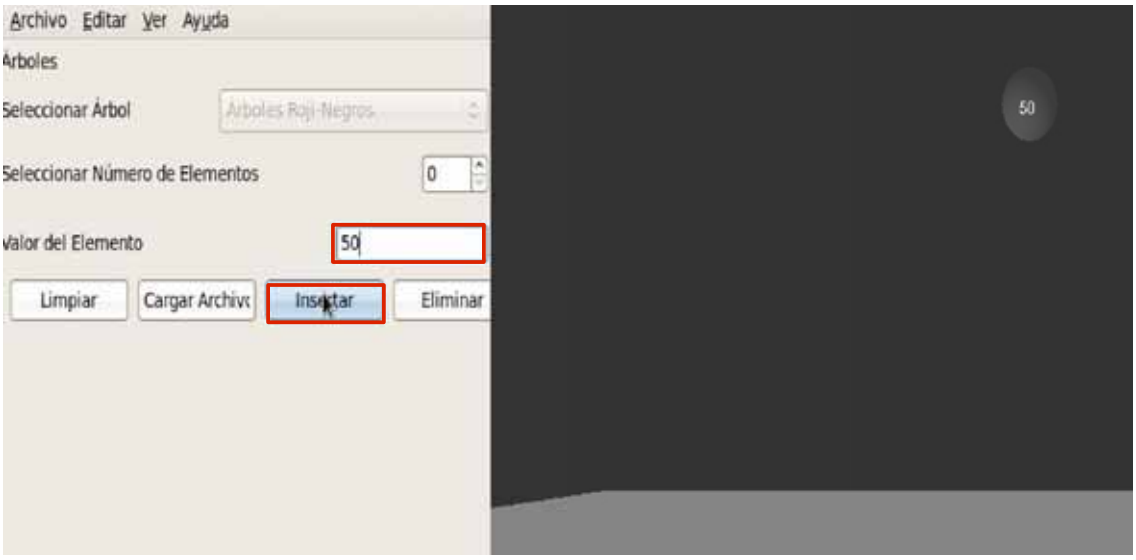



*Figura 20: Insertar elemento*

## Eliminando elementos

En el campo de texto *Valor del Elemento* indicamos el valor del elemento que se desea eliminar de la estructura, y posteriormente dar clic en el botón Eliminar, como se muestra en la Figura 21.

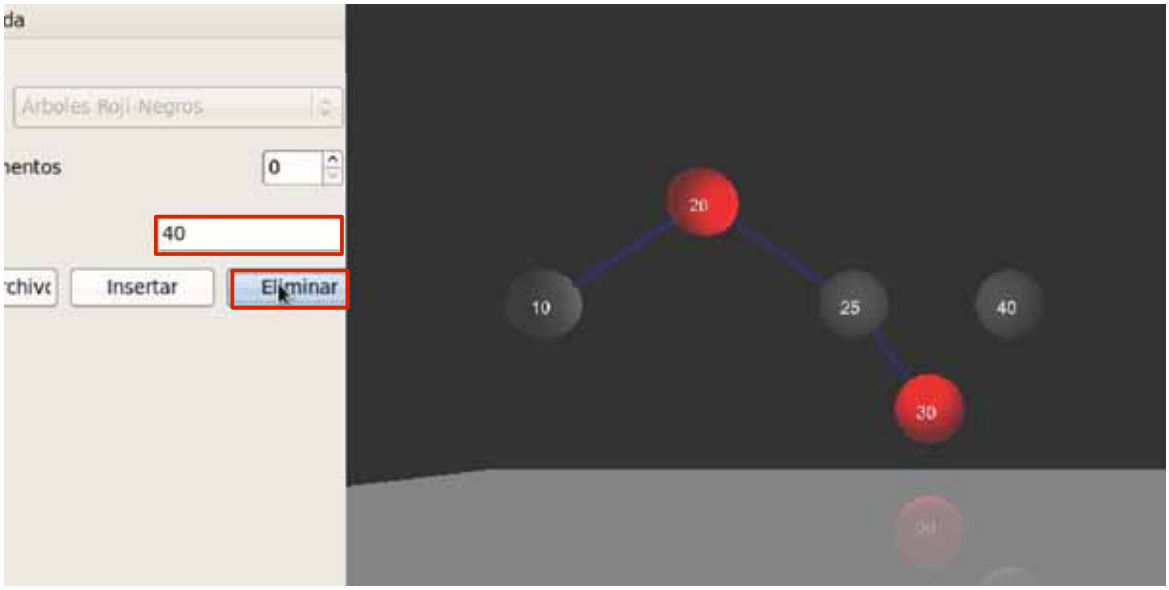



*Figura 21: Eliminar elemento*

## Limpiando Datos

El botón Limpiar esta habilitado en cualquier momento de la ejecución de la animación en el momento que se desee cambiar de estructura o se quiera iniciar un nuevo ejercicio este botón esta disponible, al hacer clic el área de animación es limpiada y se eliminan los datos en la estructura de datos, y se habilita el Combo de la sección *`Seleccionar Árbol`*, se puede ver la Figura 22 .

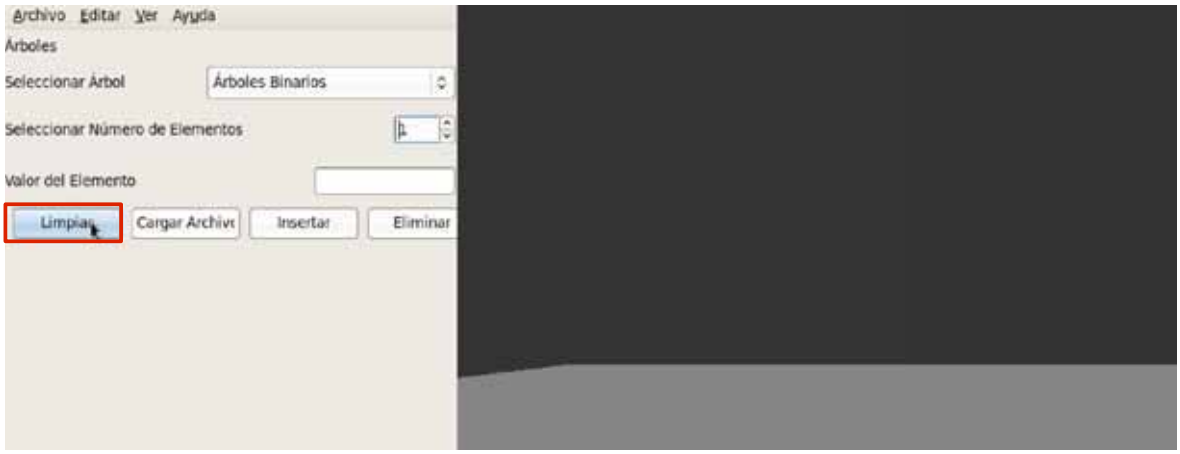



*Figura 22: Limpiar Datos*

# Bibliografía


[1] Simulaciones en Java de algoritmos de ordenamiento
http://www.mac.cie.uva.es/~arratia/cursos/UVA/sortingdemos.html. Consultada el 8 de marzo de 2009.

[2] Sorting Algorithms
http://cg.scs.carleton.ca/~morin/misc/sortalg/. Consultada el 8 de marzo de 2009.

[3] Algoritmos de ordenación
http://infnato.blogspot.com/2008/08/algoritmos-de-ordenacin.html. Consultada el 8 de marzo de 2009.

[4] Java Models
http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm. Consultada el 8 de marzo de 2009.

[5] OpenGL animation Selection sort
http://www.youtube.com/watchv=LuANFAXgQEw&playnext_from=PL&feature=PlayList&p=C3A23970563C1A3A&playnext=1&index=5. Consultada el 8 de marzo de 2009.

[6] OpenGL animation Shakersort VS Quicksort
http://www.youtube.com/watchv=oiCe1x38Fhc&playnext_from=PL&feature=PlayList&p=C3A23970563C1A3A&index=9&playnext=5&playnext_from=PL. Consultada el 8 de marzo de 2009.

[7] B-Tree Example
http://www.youtube.com/watchv=coRJrcIYbF4&playnext_from=PL&feature=PlayList&p=D44AF9DE61D09F99&index=16. Consultada el 8 de marzo de 2009.

[8] Red-Black Tree Example
http://www.youtube.com/watchv=vDHFF4wjWYU&playnext_from=PL&feature=PlayList&p=D44AF9DE61D09F99&index=17. Consultada el 8 de marzo de 2009.

[9] OpenGL
http://www.opengl.org/. Consultada el 8 de marzo de 2009

[10] Universidad Complutense de Madrid
*www.**ucm**.es*. Consultada el 8 de marzo de 2009

[11] Facultad de Informática de la Universidad Complutense de Madrid, Campus virtual Herramienta para el estudio de estructuras de datos y algoritmos.
http://books.google.com.mx/booksid=DIfiD_w68C4C&pg=PA13&lpg=PA13&dq=animaciones+de+estructuras+de+datos&source=bl&ots=n00qeMDPRO&sig=FhPgukqVABIrQmdXXjiQ3hwsank&hl=es&ei=IbuxSb2AMuH8tgei4um6Bw&sa=X&oi=book_result&resnum=1&ct=result#PPA13,M1. Consultada el 8 de marzo de 2009.

[12] Propuesta de Proyecto Terminal “Aplicación gráfica para asistir al proceso de enseñanza - aprendizaje de las Estructuras de Datos” , Universidad Autónoma Metropolitana Unidad Azcapotzalco, Abraham Torres Moro estudiante de Licenciatura en Ingeniería en Computación.

[13] Data Structures & Algorithms 1.0.
http://linux.softpedia.com/get/Science-and-Engineering/Mathematics/Data-Structures-and-Algorithms-38328.shtml. Consultada el 8 de marzo de 2009.

[14] SDL.
http://www.libsdl.org/. Consultada el 8 de marzo de 2009

[15]SDL_ttf.
http://www.libsdl.org/projects/SDL_ttf/. Consultada el 8 de marzo de 2009

[16] Fedora Project.
http://fedoraproject.org/. Consultada el 8 de marzo de 2009

[17] The GTK+ Project.
http://www.gtk.org/. Consultada el 8 de marzo de 2009

[18] OpenGL Extension to GTK+.
http://gtkglext.sourceforge.net/. Consultada el 8 de marzo de 2009