# UDrawingPaper

Manual Técnico

# UDrawingPaper

Lenguaje Utilizado: Java Versión 17
Sistema Operativo: Windows 7 en adelante
Requisitos Adicionales: Graphviz 2

Resumen

UDrawingPaper es una aplicación de escritorio dedicada a diseñar dibujos en un estilo pixelart. La aplicación permite diversas formas de crear una imagen en función de las capas que este la conforman (similar a lo que un programa de diseño pueda manejar con las capas). Así mismo, es posible el manejo de álbumes y la visualización de la estructura que almacena la información.

El manejo de toda la aplicación depende del uso y manejo de distintas estructuras de datos. La interfaz de usuario simplemente toma la información almacenada y la muestra en pantalla.

## Árbol B:

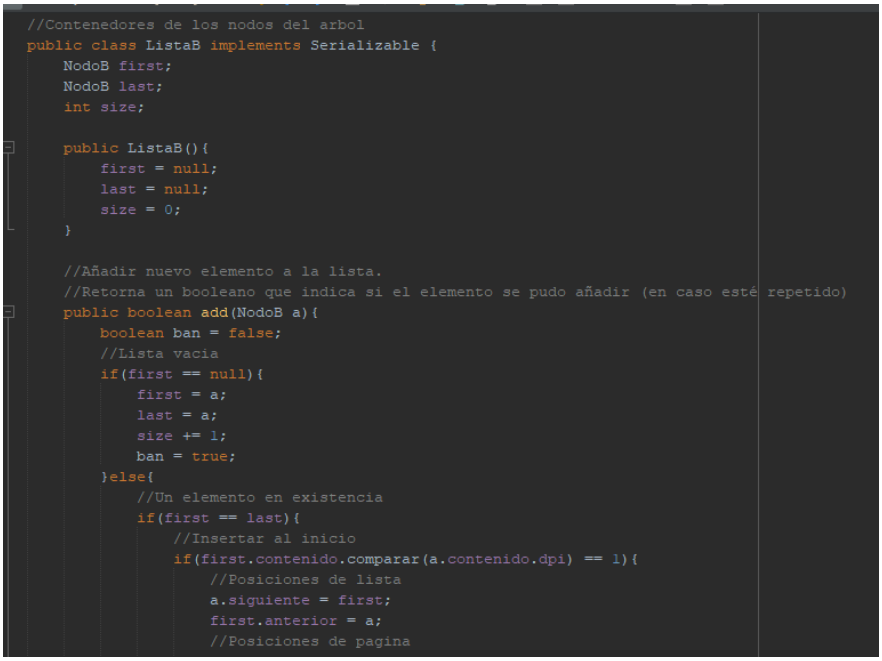
El árbol B es el esqueleto de toda la base de datos. Dentro de ella se encuentran los objetos de tipo cliente, los cuales son el nexo entre todas las estructuras poseyendo cada uno un árbol AVL para las imágenes, un ABB para las capas y una lista doble enlazada para el manejo de los álbumes.

La base del árbol es el nodo B. El nodo B almacena dentro de si cada uno de los clientes con los que se vaya a trabajar. Funciona también como pilar de la lista que conforma cada pagina y almacena los punteros que apuntan hacia otras páginas.

```
//Contenedor base del arbol
public class NodoB implements Serializable {
    public Cliente contenido;  //Contenido del nodo
    NodoB siguiente; //Sirve para la lista
    NodoB anterior; //Sirve para la lista
    PaginaB izquierda; //Nodo del arbol
    PaginaB derecha; //Nodo del arbol

    public NodoB(Cliente contenido){
        this.contenido = contenido;
        siguiente = null;
        anterior = null;
        izquierda = null;
        derecha = null;
    }
}
```

La siguiente clase es la lista. Cada uno de los nodos del árbol está compuesto por una lista. Si bien esto pudo haberse ahorrado manejado todo desde la rama, la ventaja que se vio en manejar la lista es la separación de los métodos que se manejan en el registro. El único método que maneja la lista es añadir un nuevo nodo. Debido a que la lista maneja un orden ascendente, el método añadir se fragmenta en tres, dependiendo si la inserción la hace al inicio, en medio o al final de la lista.



```
//Contenedores de los nodos del arbol
public class ListaB implements Serializable {
    NodoB first;
    NodoB last;
    int size;

    public ListaB(){
        first = null;
        last = null;
        size = 0;
    }

    //Añadir nuevo elemento a la lista.
    //Retorna un booleano que indica si el elemento se pudo añadir (en caso esté repetido)
    public boolean add(NodoB a){
        boolean ban = false;
        //Lista vacia
        if(first == null){
            first = a;
            last = a;
            size += 1;
            ban = true;
        }else{
            //Un elemento en existencia
            if(first == last){
                //Insertar al inicio
                if(first.contenido.comparar(a.contenido.dpi) == 1){
                    //Posiciones de lista
                    a.siguiente = first;
                    first.anterior = a;
                    //Posiciones de pagina
```

El contendedor de cada lista es la página. Como se dijo antes, la pagina pudo contener a la lista, sin embargo, los métodos que maneja la pagina son de una lógica y concepto diferente que, por facilidad, es mas conveniente manjar aparte. La pagina solo mantiene una tarea esencial: separarse cuando llegue al límite. La naturaleza del árbol B dice que al llegar a cierto numero de valores en la lista debe de separarse. Para este caso, cada que llegue a cinco se separa en dos páginas más.

El proceso es bastante simple. La regla dice que el valor intermedio de la pagina debe de subir a la lista padre y los términos que quedan se convierten en pagina derecha y pagina izquierda. El método dividir separa por nodos la lista, pone la del medio como nueva raíz; los dos primeros se unen en una lista para crear una nueva página y lo mismo con los últimos dos del lado derecho.

```java
//Las paginas funcionan como nodos del arbol
public class PaginaB implements Serializable{
    ListaB lista;   //Lista de claves
    int tamaño;

    public PaginaB(){
        lista = new ListaB();
        tamaño = 0;
    }
    //Dividir la pagina en caso de haber superado el limite permitido
    //Regresa el nodo de la nueva raiz
    public NodoB divide(PaginaB a){
        NodoB aux = a.lista.first;
        //Moverse al centro de la lista
        for(int i = 0; i < 2; i++){
            aux = aux.siguiente;
        }

        //Separar la lista por partes, reiniciar sus posiciones y crear nuevas paginas
        //Lista1
        NodoB L11 = a.lista.first;
        NodoB L12= L11.siguiente;
        L11.anterior = L11.siguiente = L12.anterior = L12.siguiente = null;
        PaginaB left = new PaginaB();
        left.addP(L11);
        left.addP(L12);
```

```java
    //Tiene dos opciones. En el primer caso, regresa la lista actualizada
    //En el segundo, divide las paginas y regresa el primer nodo de la nueva raiz
    public Object addP(NodoB a){
        Object val = null;
        if(lista.add(a)){
            tamaño = lista.size;
            //Validacion de tamaños
            if(tamaño == 5){
                val = divide(this);          //Nodo de la nueva raiz
            }else{
                if(tamaño < 5){
                    val = this;              //Pagina modificada
                }
            }
        }

        return val;
    }
```

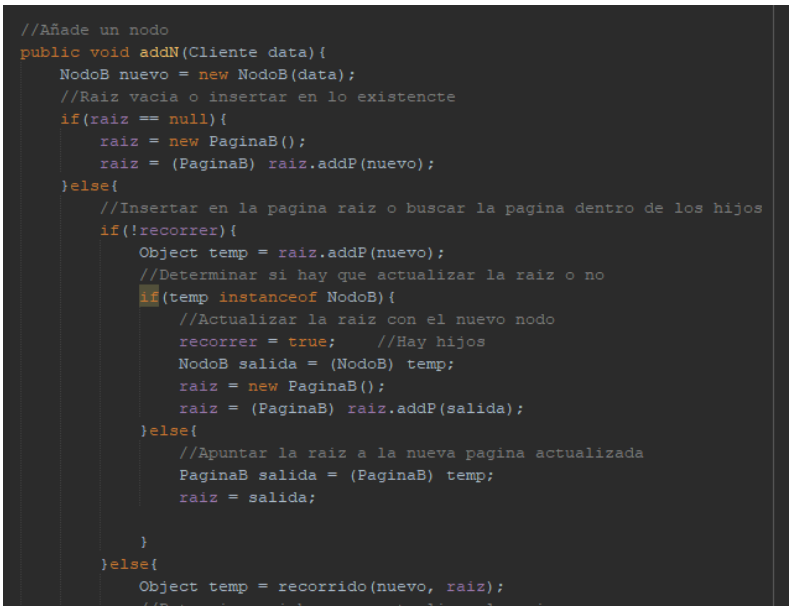
La clase Árbol es el armazón de la estructura. Mencionada, su tarea es simplemente manejar los punteros que vayan retornando los métodos. Algo que no se menciono de la pagina es que su método de división puede regresar dos cosas: un nodo o una o página. Cada vez que se regresa un nodo se asume que es un nuevo padre del árbol con todas las conexiones: Este solo se añade junto a la lista de padres que se encuentra por encima. En caso de retornar una página, lo que regresa es la pagina con el nuevo valor agregado.

```
*/
public class ArbolB implements Serializable {
    public PaginaB raiz;
    boolean recorrer;       //Indica si hay que recorrer o no el arbol

    public ArbolB(){
        raiz = null;
        recorrer = false;
    }

    //Metodo recursivo para insertar el nodo
    public Object recorrido(NodoB a, PaginaB origen){
        //Bandera que determina si es hoja.
        int ban = 0;
        //Revisa que el primer nodo tenga conexiones
        if(origen.lista.first.izquierda == null){
            ban = 1;
        }

        //Caso base
        if(ban == 1){
            //Añadir normalmente el nodo a la pagina
            return origen.addP(a);
        }else{
            //Menor a la raiz. Se mueve a la lista izquierda.
            if(origen.lista.first.contenido.comparar(a.contenido.dpi) == 1){
                Object r = recorrido(a, origen.lista.first.izquierda);
                //Determinar si hay que actualizar la hoja o no
                if(r instanceof NodoB){
                    //Insertar a la lista el nodo
```



```
//Añade un nodo
public void addN(Cliente data){
    NodoB nuevo = new NodoB(data);
    //Raiz vacia o insertar en lo existencte
    if(raiz == null){
        raiz = new PaginaB();
        raiz = (PaginaB) raiz.addP(nuevo);
    }else{
        //Insertar en la pagina raiz o buscar la pagina dentro de los hijos
        if(!recorrer){
            Object temp = raiz.addP(nuevo);
            //Determinar si hay que actualizar la raiz o no
            if(temp instanceof NodoB){
                //Actualizar la raiz con el nuevo nodo
                recorrer = true;    //Hay hijos
                NodoB salida = (NodoB) temp;
                raiz = new PaginaB();
                raiz = (PaginaB) raiz.addP(salida);
            }else{
                //Apuntar la raiz a la nueva pagina actualizada
                PaginaB salida = (PaginaB) temp;
                raiz = salida;

            }
        }else{
            Object temp = recorrido(nuevo, raiz);
```

Otros métodos relevantes para mencionar son el método de búsqueda y el método niveles. El método de búsqueda, como su nombre lo indica, es el encargado de regresar el puntero con el cliente en concreto que se indica como parámetro (usado en el login). Simplemente recorre el árbol similar a como lo haría uno binario. El método de niveles se encarga de hacer un recorrido por niveles al árbol. Para eso, se utiliza la ayuda de una cola. Toda esa información es utilizada para crear un listado con los clientes que conforman la estructura.

```
//Metodo recursivo para insertar el nodo
public NodoB buscar(String val, PaginaB origen){
    //Bandera que determina si es hoja.
    int ban = 0;
    //Revisa que el primer nodo tenga conexiones
    if(origen.lista.first.izquierda == null){
        ban = 1;
    }

    NodoB salida = null;

    //Caso base
    if(ban == 1){
        NodoB aux = origen.lista.first;
        int bn = 0;
        while(bn == 0){
            if(aux.contenido.comparar(val) == 0){
                salida = aux;
                bn = 1;
            }else{
                if(aux.siguiente == null){
                    salida = null;
                    bn = 1;
                }else{
                    if(aux.contenido.comparar(val) == -1){
                        aux = aux.siguiente;
                    }else{
                        salida = null;
                        bn = 1;
                    }
```

## Árbol AVL:

El árbol AVL se encarga de almacenar el objeto imagen. Las imágenes tienen en su posición un identificador, el cual sirve para almacenar y manejar el árbol y un subárbol ABB de capas que maneja las imágenes almacenadas en memoria.

El nodo AVL es la base de la estructura. Al ser un árbol binario, maneja dos punteros para moverse por sus hijos izquierdo o derecho y un nodo padre utilizado en cierta parte de las rotaciones.

```
 */
public class NodoAVL implements Serializable{
    String ID; //ID del nodo - Se usa al graficar
    public Imagen content; //contenido del nodo
    public NodoAVL hijo1; //Hijo izquierdo del arbol
    public NodoAVL hijo2; //Hijo derecho del arbol
    public NodoAVL padre; //Padre del noco en cuestion. Si no tiene es la raiz.
    Boolean hoja; //Determina si el arbol es una hoja o no
    int altura;

    public NodoAVL(Imagen contenido){
        this.ID = "";
        this.content = contenido;
        this.hijo1 = null;
        this.hijo2 = null;
        this.padre = null;
        this.hoja = true;
        this.altura = 0;
    }
}
```

De forma simple, el árbol AVL funciona como un árbol ABB. Lo que lo hace complejo es el método del balanceo. Gracias a la recursividad que posee el análisis del árbol, el enfoque usado para agregar es ir de arriba hacia abajo buscando la rama donde debe de insertarse el nodo. De ahí hacia arriba, el árbol se encarga de ir calculando alturas y comprobando si debe de realizar una rotación.

Las rotaciones no son mas que movimientos de nodos. Existen las simples y las compuestas que realizan las simples dos veces.

```
 */
public class ArbolAVL implements Serializable{
    public NodoAVL raiz;
    int tamaño;

    public ArbolAVL(){
        this.raiz = null;
        tamaño = 0;
    }

    //Altura del nodo. Regresa la altura del nodo
    public int altura(NodoAVL nodo){
        if(nodo != null){
            return nodo.altura;
        }else{
            return -1;
        }
    }

    //Profundidad del nodo. Realiza una compración simple entre dos valores y regresa el más grande.
    public int mayor(int a1, int a2){
        if(a1 >= a2){
            return a1;
        }else{
            return a2;
        }
    }
```

```
    //Rotación simple por la izquerda
    public NodoAVL rotacionl(NodoAVL nodo){
        NodoAVL aux = nodo.hijo1;
        nodo.hijo1 = aux.hijo2;
        aux.hijo2 = nodo;
        //Calculo de las nuevas alturas
        nodo.altura = mayor(altura(nodo.hijo2), altura(nodo.hijo1)) + 1;

        aux.altura = mayor(nodo.altura , altura(nodo.hijo1)) + 1;
        return aux;
    }
    //Rotación simple por la derecha
    public NodoAVL rotacionr(NodoAVL nodo){
        NodoAVL aux = nodo.hijo2;
        nodo.hijo2= aux.hijo1;
        aux.hijo1 = nodo;
        //Calculo de las nuevas alturas
        nodo.altura = mayor(altura(nodo.hijo1), altura(nodo.hijo2)) + 1;
        aux.altura = mayor(nodo.altura , altura(nodo.hijo2)) + 1;
        return aux;
    }

    //Rotación Izquierda - Derecha. COmbinar rotaciones nada más.
    public NodoAVL rotacionlr(NodoAVL nodo){
        nodo.hijo1 = rotacionr(nodo.hijo1);
        NodoAVL aux = rotacionl(nodo);
        return aux;
    }
```

La otra función del árbol es borrar. La teoría indica que se debe de borrar el valor mas grande de la rama izquierda o la menor de la rama derecha.  Se opto para la rama derecha. Para hallarlo es que existe el método grande. Este simplemente se sigue moviendo por el lado derecho del árbol hasta que encuentra un tope. Se busca el puntero que de con ese nodo y se borra.

Luego de borrar hay que revisar que el árbol se encuentre equilibrado por lo que se va de abajo hasta arriba actualizando los pesos y rotando los nodos en caso no cumplan con el equilibrio.

```
//Eliminar Nodo
public void borrar(NodoAVL nodo){
    if(nodo != null){
        if(nodo.hijo1 == null && nodo.hijo2 == null){   //Es una hoja
            if(nodo.padre.hijo1 != null){
                if(nodo.padre.hijo1.equals(nodo)){
                    nodo.padre.hijo1 = null;
                    actualizar(this.raiz);
                    balance(this.raiz, null);
                }else{
                    nodo.padre.hijo2 = null;
                    actualizar(this.raiz);
                    balance(this.raiz, null);
                }
            }else{
                nodo.padre.hijo2 = null;
                actualizar(this.raiz);
                balance(this.raiz, null);
            }

            nodo = null;

        }else{
            if(nodo.hijo1 == null){         //Solo tiene un hijo
                nodo.content = nodo.hijo2.content;
                nodo.hijo2 = null;
                actualizar(this.raiz);
                balance(this.raiz, null);
```

```
    //Conseguir la posicion del más grande de los pequeños
    public Object grande(NodoAVL nodo, NodoAVL pa){
        NodoAVL aux = nodo;
        NodoAVL padre= pa;
        int pos = 1; //1 para hijo izquierdo y 2 para hijo derecho
        int ban = 0;     //Estado Inicial
        Object content = null;
        if(nodo.hijo1 == null && nodo.hijo2 == null){
            content = nodo.content;
            pa.hijo1 = null;
        }else{
            while(ban == 0){
                if(aux.hijo2 != null){
                    padre = aux;
                    pos = 2;
                    aux = aux.hijo2;
                }else{
                    content = aux.content;
                    if(pos == 1){
                       padre.hijo1 = aux.hijo1;
                    }else{
                       padre.hijo2 = aux.hijo2;
                        }
                    ban = 1;
                }
            }
        }

        return content;
```

## Árbol ABB:

El árbol binario de búsqueda se encarga de almacenar la información relacionada con las capas (matrices). Su unidad básica es el NodoABB. El nodo ABB solo posee punteros hacia sus hijos. Maneja un espacio para el nombre y uno para su contenido.

```
public class NodoABB implements Serializable{
    String ID; //ID del nodo - Se usa al graficar
    public Capa content; //contenido del nodo
    public NodoABB hijo1; //Hijo izquierdo del arbol
    public NodoABB hijo2; //Hijo derecho del arbol
    Boolean hoja; //Determina si el arbol es una hoja o no
    int altura;

    public NodoABB(Capa contenido){
        this.ID = "";
        this.content = contenido;
        this.hijo1 = null;
        this.hijo2 = null;
        this.hoja = true;
    }
}
```

El método de añadir del árbol ABB es bastante simple. Su funcionamiento es ir comparando su valor con el valor de la raíz y se va moviendo por las ramas hasta que encuentre que uno de los padres tiene un hijo vacío en la rama correspondiente.

```
//Metodo Añadir
public void add(Capa agregar, NodoABB revisar){  //Siempre que se vaya a usar, se envia la raiz para que empie
        if(raiz == null){
            NodoABB nuevo = new NodoABB(agregar);
            raiz = nuevo;
            tamaño++;
        }else{
        if(revisar.content.id != agregar.id){
            if(revisar.content.id > agregar.id){
                if(revisar.hijo1 == null){
                    NodoABB nuevo = new NodoABB(agregar);
                    revisar.hijo1 = nuevo;
                    tamaño++;
                }else{
                    add(agregar, revisar.hijo1);
                }
            }else{
                if(revisar.hijo2 == null){
                    NodoABB nuevo = new NodoABB(agregar);
                    revisar.hijo2 = nuevo;
                    tamaño++;
                }else{
                    add(agregar, revisar.hijo2);
                }
            }
        }
    }
}
```

Tanto el árbol AVL como este poseen una serie de recorridos preestablecidos. Estos son el reorden, inrden y postorden. Dichos recorridos se basan en ir descendiendo por el árbol hasta que ya no pueda en cierto orden.

```java
//Metodo para preorden. Siempre que se vaya a declarar, se tiene que mandar raiz co
public String preorder(NodoABB inicio){
    String t = "";
    if(raiz != null){
        t += inicio.content.id + " " ;
        if(inicio.hijo1 != null){
            t += preorder(inicio.hijo1);
        }
        if(inicio.hijo2 != null){
            t += preorder(inicio.hijo2);
        }
    }
    return t;
}
public String preorderI(NodoABB inicio){
    String t = "";
    if(raiz != null){
        t += inicio.content.id + "," ;
        if(inicio.hijo1 != null){
            t += preorderI(inicio.hijo1);
        }
        if(inicio.hijo2 != null){
            t += preorderI(inicio.hijo2);
        }
    }
    return t;
}
```

## Matriz:

La matriz es la representación gráfica del pixelart generado por la aplicación. La base de toda la estructura es un nodo matriz. Un nodo matriz es un contenedor que guarda el hexadecimal del color que se quiere guardar y posee punteros en las cuatro direcciones cardinales.

```java
import java.io.Serializable;

/**
 *
 * @author JJONK19
 */
public class NodoMatriz implements Serializable {
    NodoMatriz arriba;
    NodoMatriz abajo;
    NodoMatriz izquierda;
    NodoMatriz derecha;
    int x; //Posicion en x de la matriz
    int y; //Posicion en y de la matriz
    Object contenido;
    String ID; //Usado para graficar
    String Nombre;

    //Nodo para construir nodos
    public NodoMatriz(int x, int y, Object contenido){
        arriba = null;
        abajo = null;
        izquierda = null;
        derecha = null;
        this.x = x;
        this.y = y;
        this.contenido = contenido;
        this.Nombre = "";
    }
```

La matriz en si solo posee un nodo raíz. La matriz posee dos llistas de cabeceras (una vertical y otra horizontal). Cada que se quiere añadir un nuevo nodo, se recorren ambas caberas y se busca si las coordenadas ingresadas existen en la lista. De no hacerlo, la aplicación comienza a recorrer las listas hasta que encuentra el espacio correspondiente. Luego ingresa todas las cabeceras que habían de por medio para mantener la sucesión.

En caso existir, lo que hace es ir a la cabecera y desde ahí recorrer esa lista para ingresarlo de tal manera que se conserve su orden ascendente. Eso mismo aplica para la otra cabecera. En caso de existir la posición, simplemente remplaza el contenido del nodo.

```
//Buscar si la columna ya existe en la lista de columnas
//Regresa null si no lo encuentra
public NodoMatriz searchy(int y){
    NodoMatriz aux = raiz;
    while(aux != null){
        if(aux.y == y){
            break;
        }else{
            aux = aux.derecha;
        }
    }

    return aux;
}

//Buscar si la fila ya existe en la lista de filas
//Regresa null si no lo encuentra
public NodoMatriz searchx(int x){
    NodoMatriz aux = raiz;
    while(aux != null){
        if(aux.x == x){
            break;
        }else{
            aux = aux.abajo;
        }
    }

    return aux;
}
```

```
}
//Añadir una posicion a la matriz
public void add(int x, int y, Object contenido){
    NodoMatriz tempx = searchx(x);
    NodoMatriz tempy = searchy(y);
    NodoMatriz temp = new NodoMatriz(x, y, contenido);

    if(tempx == null && tempy == null){
        nfil(x);    //Añadir la fila que no está
        ncol(y);    //añadir la columna que no esta
        tempx = searchx(x);
        tempy = searchy(y);
        tempx.derecha = temp;
        temp.izquierda = tempx;
        tempy.abajo = temp;
        temp.arriba = tempy;
    }else{
        if(tempx != null && tempy == null){
            //Añadir en y
            ncol(y);    //añadir la columna que no esta
            tempy = searchy(y);
            tempy.abajo = temp;
            temp.arriba = tempy;

            //Añadir en x. Es copiar el metodo para añadir columna.
            NodoMatriz aux = tempx;
            while(aux != null){
                if(y > aux.y){
                    //Añadir al final
                    if(aux.derecha == null){
```

## Lista Doble Circular:

En esta lista se almacenan los álbumes generados de los archivos de prueba. Funcionan como estructuras lineales que se mueven a la derecha o la izquierda. La cualidad especial de la lista es que guarda listas simples dentro de sus nodos.

Dentro de la aplicación, su funcionamiento se limita a la eliminación de imágenes. Cada vez que se birra una imagen, se recorre cada una de las listas en búsqueda de la imagen en cada lista.

```
//Metodos de la lista
//-----------------------------------------------------------------
public void add(Object _content, String _name){
    NodoListaSimple temp = new NodoListaSimple (_content, _name);
    if(isEmpty()){
        head = temp;
        no++;
    }else{
        NodoListaSimple r = head;
        while(r.next != null){
            r = r.next;
        }
        r.next = temp;
        no++;
    }
}


//Vacia la lista
public void deleteL(){
    this.head = null;
    no = 0;
}

public boolean isEmpty(){
    return this.head == null;
}

//Las posiciones empiezan desde 0. Restarle uni al idice para no te
```

## Interfaz:

El funcionamiento de la interfaz no funciona de una manera particular. La mayor parte del tiempo, el proceso consiste en obtener imágenes de las interfaces o actualizar/añadirlas. Entre las cosas que cabe destacar:

La lectura de los JSON es apoyada gracias a la librería JSONPath. Esta clase tiene la facilidad de guardar como diccionarios la información adquirida. Para ciertos casos, es conveniente porque permite castear directamente un JSONArray a un arreglo de objetos directamente.

```
private void ccapActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    aux.contenido.capas = new ArbolABB();   //Reiniciar Arbol
    JFileChooser filechooser = new JFileChooser();
    FileNameExtensionFilter exp = new FileNameExtensionFilter("Archivos JSON (*.json)", "json");
    filechooser.addChoosableFileFilter(exp);
    filechooser.setFileFilter(exp);
    if(filechooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION){
        try{
            File json = filechooser.getSelectedFile().getAbsoluteFile();
            //Separar Nombres
            List<Integer> nombre = JsonPath.parse(json).read("$.*.id_capa");

            //Separar Lista de pixeles
            List<Map> pix = JsonPath.parse(json).read("$.*.pixeles");

            //Crear capas
            for (int i =0; i < nombre.size();i++){
                //Lectura de los diccionarios
                int tempm = nombre.get(i);

                //Creacion de lista de pixeles
                JSONArray temp = (JSONArray) pix.get(i);
                List<Map> pixa = JsonPath.parse(temp).read("$.*");

                //Creacion de capa
                Capa Nuevo = new Capa(tempm, pixa);

                aux.contenido.capas.add(Nuevo, aux.contenido.capas.raiz);
```

La generación de imágenes es parte importante para corroborar el comportamiento correcto de la aplicación, sin embargo, es imposible mantener la eficiencia de la para todos los casos. Como nota, es importante mencionar que, para algunas imágenes, el proceso de generación puede ser demasiado tardado y puede que genere problemas al intentar cargar la imagen a la aplicación.

El ultimo detalle consiste en el proceso de mover información entre ventanas. La opción seleccionada para eso fue la serialización. Se maneja un árbol B general

dúrate toda la ejecución que constantemente se serializa y se deserializa conforme se van abriendo o cerrando ventanas.

```
private void formWindowClosing(java.awt.event.WindowEvent evt) {
    // TODO add your handling code here:
    //Serealizar
    try{
        //Crear data
        FileOutputStream f=new FileOutputStream("src/main/java/data.ser");
        ObjectOutputStream out=new ObjectOutputStream(f);
        data = new ArbolB();
        out.writeObject(data);
        out.flush();
        out.close();
    }catch(Exception e){

    }

    Login n = new Login();
    n.setVisible(true);
    this.dispose();
}
```

```
private void formWindowOpened(java.awt.event.WindowEvent evt) {
    // TODO add your handling code here:
    // TODO add your handling code here:
    try {
        FileInputStream abrir = new FileInputStream("src/main/java/data.ser");
        ObjectInputStream escribir = new ObjectInputStream(abrir);
        data =  (ArbolB) escribir.readObject();
        escribir.close();
        abrir.close();

        aux = data.buscar(user, data.raiz);
    } catch (IOException i) {
       data = new ArbolB();

    } catch (ClassNotFoundException c) {
        data = new ArbolB();
    }
}
```