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. Esta actualización de la aplicación está orientada al proceso de distribución de las imágenes, así como el manejo de los registros y todas las transacciones llevadas por la aplicación a través del uso de Blockchain.

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.

Tabla Hash:

La tabla hash es la encargada de almacenar la información relacionada con los mensajeros. De forma simple, la tabla hash no es mas que un arreglo cualquiera. Lo que hace que difiera y sea diferente a este es el posicionamiento y forma de almacenar la información de la estructura, así como el cambio en el tamaño que pueda experimentar.

El pilar de la tabla hash, y lo que almacena, es el nodo hash. El nodo hash es el encargado de guardar cual es la llave que identifica al contenido, así como el contenido en si mismo. También es un auxiliar dentro de la traficación pues sirve durante el recorrido para saber que se debe o no graficar, así como la conexión y manejo dentro de los nodos.

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package Estructuras;

import java.io.Serializable;
import java.util.UUID;

/**
 *
 * @author JJONK19
 */
public class NodoHash implements Serializable {
    public Mensajero content;
    public int key;
    public String ID;          //ID del nodo. Tiene uso en el gráfico.

    public NodoHash(Mensajero _content, int _key){
        content = _content;
        key = _key;
        ID = "\""+UUID.randomUUID().toString() + "\"";
    }

}
```

El método más importante de esta es el método add, el cual añade un nuevo dato a la tabla hash, pero antes, cabe explicar los otros métodos que sirven de auxiliar a éste.

El método uso tiene la tarea de calcular cual es el porcentaje de uso que esta manejando la tabla hash. Utiliza la variable uso, la cual indica el numero de espacios llenos dentro del array, y tamaño que indica el tamaño actual del array.

Hash es la encargada de devolver el índice al que va a ir dirigido la entrada. Por requerimientos, el valor a aplicar el hash es el DPI. El DPI es un numero de trece dígitos por lo que se almacena como un Big Integer y desde ahí se aplica la función indicada dentro del enunciado. Ese valor luego es casteado a String y devuelto como un int.

La función colisión es una opción a la función hash cuando esta tiende a fallar. Al igual que la anterior recibe un BigInteger que evalúa dentro de la función, convierte el resultado a String y lo retorna como int.

```java
//Retorna el porcentaje de uso en la tabla
public double uso(){
    return (double)uso/tamaño;
}

//Retorna el indice de la tabla
public int hash(BigInteger a){
    BigInteger tam = new BigInteger(Integer.toString(tamaño));
    BigInteger hash = a.mod(tam);
    String temp = hash.toString();
    int salida = Integer.parseInt(temp);
    return salida;
}

//Maneja las colisiones
public int colision(BigInteger a, int colision){
    BigInteger col = new BigInteger(Integer.toString(colision));
    BigInteger siete = new BigInteger(Integer.toString(7));
    BigInteger uno = new BigInteger(Integer.toString(1));
    BigInteger hash = (a.mod(siete).add(uno)).multiply(col) ;
    String temp = hash.toString();
    int salida = Integer.parseInt(temp);
    return salida;
}
```

La función primo se encarga de retornar el valor del primo siguiente al valor de entrada que reciba. El procedimiento que realiza es ir dividiendo el numero a la mitad e iterar un ciclo donde se calcule el residuo entre los valores que sean menores a este. En caso detecto que el residuo es cero, no cumple. Esta tarea se va realizando, sumando uno al valor de entrada y repetir con cada número hasta que se detecte un numero primo. Esto se usa en el re-hash ya que el valor que debe ir tomando la tabla conforme va creciendo debe de ser el primo próximo al valor actual.

```java
    //Retorna el primo siguiente al tamaño actual
    //Basado = https://www.javatpoint.com/prime-number-program-in-java
    public int primo(int a){
    int i = 0;
    int m = 0;
    int flag = 0;
    int n = a + 1;
    boolean ban = true;

    while(ban){
        flag = 0;
        m = n / 2;
        if(n==0||n==1){
            n++;
        }else{
            for(i = 2;i <= m; i++){
              if(n % i == 0){
                n++;
                flag = 1;
               break;
              }
            }
            if(flag == 0){
                ban = false;
            }
        }
    }
        return n;
    }
```

El método re-hash se encarga de reorganizar la tabla hash cada vez que se supera el porcentaje de uso permitido (0.75). Lo que hace es ejecutar el método primo y encontrar el siguiente tamaño de la tabla. Esto lo ejecuta hasta que encuentre un tamaño que permita mantener el uso bajo el límite. Una vez hecho eso, se guarda el array actual en una variable y se reinicia la tabla. Se recorre el temporal, se les asigna un nuevo hash y se almacena dentro de la tabla.

```
//Actualiza el tamaño de la tabla así como sus posiciones
public void rehash(){
    //Actualizar el tamaño
        while(uso() >= 0.75){
            tamaño = primo(tamaño);
        }
        NodoHash[] bt = bucket;
        bucket = new NodoHash[tamaño];

    //Recorrer de nuevo la lista y aplicar rehash
    for(int i = 0; i < bt.length; i++){
        if(bt[i] != null){
            Mensajero a = bt[i].content;
            addR(a);
        }
    }

}
```

Una vez explicado eso, el método add funciona así: Recibe el objeto de entrada y extrae el DPI. Luego lo usa para generar el hash. Si el hash supera el tamaño, se le envía a la función colisión para obtener el nuevo hash. Si el valor esta ocupado se sigue iterando hasta que encuentre un valor disponible o supera el tamaño que en cuyo caso, no lo agrega. Este mismo procedimiento se repite cuando el tamaño no se supere y la posición se encuentre ocupada.

```
//Añade un Valor a la tabla
public void add(Mensajero a){
    //Añadir al bucket
    int posicion = hash(a.dpi);
    //Revisar si el indice no se sale de la tabla
    if(posicion >= (bucket.length)){
            boolean ban = true;
            int i = 1;
            //Iterar la creacion de un nuevo hash hasta que entre
            while(ban){
                posicion = colision(a.dpi, i);
                if(posicion >= (bucket.length)){
                    ban = false;
                    System.out.println(a.dpi + " genera un hash superior al tamaño " + posicion + " NTamaño:" +
                }else{
                    NodoHash check = bucket[posicion];
                    if(check == null){
                        //Añadir a la posicion
                        bucket[posicion] = new NodoHash(a, posicion);
                        uso++;
                         ban = false;
                    }
                }
                i++;
            }
    }else{
        NodoHash check = bucket[posicion];
```

Lista Simple:

La lista simple es la base para el manejo de la mayoría de las bases de datos que utiliza la aplicación. Para el caso de los clientes el funcionamiento de esta es tradicional y no varia en nada, almacenando cada entrada de manera secuencial.

El otro uso importante que tiene es en la lista adyacente la cual no es mas que una lista de listas. Lo único que varia en su funcionamiento es que almacena en su nodo una lista. Cabe mencionar, que los métodos de traficación son diferentes, pero solo se modifica la forma en que se declara y conecta cada cosa que traiga dentro.

El método add lo que hace es añadir un nuevo valor a la lista. Crea el nodo y si la lista esta vacía, lo añade como cabecera. De lo contrario comienza a recorrer cada nodo hasta que encuentre uno donde tenga la posición next vacía para añadirlo.

```
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++;
    }
}
```

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 dé serializa conforme se van abriendo o cerrando ventanas.

```java
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();
}
```

```java
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();
    }
}
```