LibStructureGraphic 1.0 User Manual

Project Home: http://code.google.com/p/structure-graphic/
Manual Written by **Barak Itkin <lightningismyname@gmail.com>**

What is it?

StructureGraphic is a library written in java, for the purpose of visualizing data structures. It is used mainly (but not only) for debugging of complex data structures, in which text-based debugging would not be ideal.

This library is written in Java, and is meant to be used with code in Java. Support for other languages is not currently supported.

When can I use it?

StructureGraphic Version 1.0 (the current version) allows you to draw **any data structure that can be represented as a Directed Tree Graph** (A tree with directed edges). This includes simple data structures such as lists, and also more complex structures such as search trees (Red-Black, B-Tree, …).

Version 2.0 (in development, planned for the near future) will also allow drawing cyclic graphs and graphs which are not connected.

How do I use it?

Basically, you need to tell the library some basic details about your data structure, and then you simply call a “paint” method whenever you want to draw the graph.

Drawing Trees (Version 1.0 and later)

There are currently 3 ways to adapt your data structure to be used with StructureGraphic:

- Implementing the DSTreeNode interface
- Annotating the node objects of your data structure with special annotations. **This is the preferred method**, but it's not always possible to use it.
- Convert a text based representation to a tree – to be used **if you already have a method to print the tree to the console or a file**.

In the first two methods, once your data structure is adapted to the library, you just need to **pass the root to a special method that will draw the tree**. In the third method, you simply paste the text to a method that will convert it to a tree that can be passed to the paint method mentioned above.

To import all the classes/interfaces/annotations mentioned in this part, simply add the following line to your code:

```
import org.StructureGraphic.v1.*;
```

After your data structure implements one of the two techniques, or is parsed from text by the parser, to show it just pass the root of the tree data structure to DSutils.show. That function receives two more parameters – the width and height in which nodes should be drawn – you'll have to play with these…

Implementing the DSTreeNode Interface

The DSTreeNode interface is very simple to implement – it basically says that for every node in a tree, we need to know who its children are, and what the value we should draw inside this node is. Below, you can see the full documentation of the interface – and as you'll see, it's very straight forward…

DSTreeNode.java

```
/**
 * Specify the basic methods that a node in a Tree-Like data structure should
 * have.
 *
 * Assumption - The graph is a directed tree:
 * - No two nodes share the same child
 * - There are no circles in the data structure
 * @author Barak Itkin
 * @since Version 1.0
 */
public interface DSTreeNode {

    /**
     * Return the children of the given node inside an array.
     * THE ORDER OF THE NODES MATTER!
     *
     * @return An ordered array containing the children of this node. THIS VALUE
     *         MUST NOT BE NULL! RETURN AN ARRAY OF LENGTH 0 TO SPECIFY THAT
     *         THERE ARE NO CHILDREN!
     */
```

```
    public DSTreeNode[] DSgetChildren();

    /**
     * Get the value of the node. The value will be treated as a string, using
     * the object's toString() method. This is good for all the primitive types
     * (strings, numbers, booleans and chars) and also for more complex objects
     * which have their own toString() method.
     * If the value returned is null, it will be converted to the string "null"
     *
     * @return The node's value
     */
    public Object DSgetValue();

    /**
     * Return the color for drawing this node (Useful for Red Black Trees).
     * Return null for specifying the default color
     *
     * @return the color to be used for this node
     */
    public java.awt.Color DSgetColor();
}
```

Example

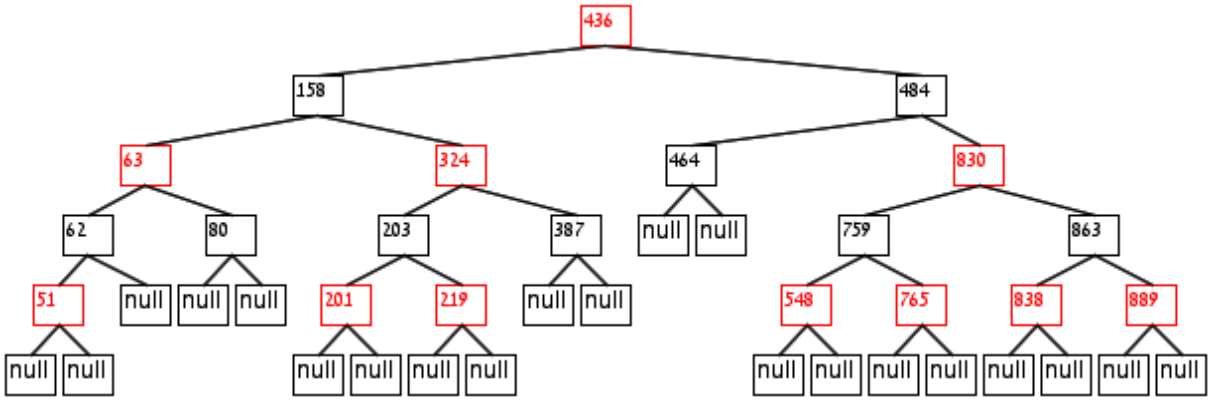
Here is a code from a working red black tree:

RBTree.java

```
import java.awt.Color;

public class RBTree {

    public RBNode head;

    public class RBNode implements DSTreeNode {

        private RBNode left;
        private RBNode right;
        private int value; //node's value
        private boolean black; //a boolean parameter for node's color

        public DSTreeNode[] DSgetChildren() {
            return new DSTreeNode[]{left,right};
        }

        public Object DSgetValue() {
            return this.value;
        }

        public Color DSgetColor() {
            return this.black ? Color.BLACK : Color.RED;
        }
```

```
        ...
    }
    ...
}
```

Now we will simply pass the head field of the RBTree to the paint method to get something like this:



Note the very important fact – **children which are null, are drawn as null nodes!** This is because our implementation simply returns an array of children without checking if they are null or not. This is good, because if we would draw only one child in that case, the visualization wouldn't show that it's the left/right (we would know this by the value, but it's annoying to check that).

Using the @DS annotations

This method assumes that each node in the tree is represented by a unique object, where its value is stored in one or more fields of the object (if it has no value, no such field is needed), and it's children are stored in one or more fields (if it has no children, no such field is needed).

- @DSChildren(DSChildren.DSChildField.ITERABLE)
  Use this on a field which stores the children inside an object which implements the java.util.Iterable interface (for example, lists).
- @DSChildren(DSChildren.DSChildField. ARRAY)
  Use this on a field which stores the children inside an array of some type.
- @DSChildren(DSChildren.DSChildField. SINGLE)
  Use this on a field which specifies an object which is a child
- @DSColor
  Use this on a field from type java.awt.Color to specify the color that should be used for drawing this node. Usually unused, except for Red-Black trees.
- @DSValue
  Mark any value that should be drawn inside the rectangle representing this node. This will be drawn as FIELD_NAME: FIELD_VALUE.

Special notes

- **All the annotated fields must be public!**
- When not even one value is specified, the value will be the empty string. When more than one value is specified, all the values will be displayed.
- When specifying more than one @DSChildren field, no specific order is guaranteed! Note however that inside a single field of children, the internal order will be kept for iterables and arrays.
- When not even one child is specified, the node will have no children. If the node has a child field, which is null, **null will be painted as a part of the tree as a child!**
- When no color is specified, the default color (black) will be used.

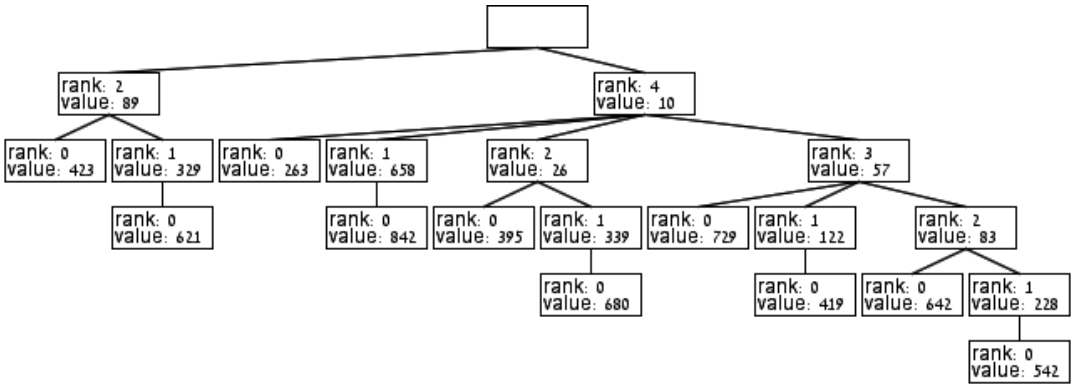
Example

Here is a real working code from a binomial heap:

Heap.java

```
public class Heap {

    @DSChildren(DSChildren.DSChildField.ITERABLE)
    public NodeList forest;

    ...

    public class HeapNode {

        @DSValue
        public int rank;

        @DSValue
        public int value;

        @DSChildren(DSChildren.DSChildField.ITERABLE)
        public NodeList children;

        ...

    }

    public static class NodeList implements Iterable<HeapNode> {

        ...

   }
}
```

The result will look like this:



Using the text parser

The text parser is the quickest and dirtiest way to use the library, and it is meant for people who already have old style debug printing implemented. It will take a string in which every object is in a single line, and the "depth" of the child is indicated by the amount of leading **spaces**.

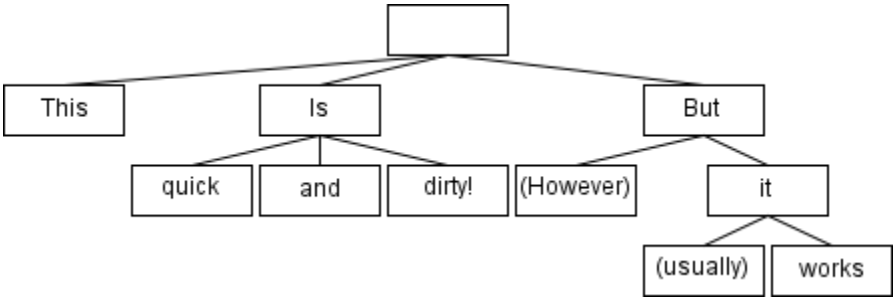
To use it, just do:

```
DSTreeParser.parseSimpleMultiline(m)
```

Where m is a string (or a java.io.File containing a string) formatted like in the example below. This method creates a tree from your text representation, and you can pass this tree to any of the tree painting methods.

Example

```
This
Is
  quick
  and
  dirty!
But
  (However)
  it
    (usually)
    works
```



Note how this method created a common ancestor to group all the roots.