The BST Tutorial

> *This document will guide you through the creation of a BST formatted story. You are welcomed to suggest additions and modifications here.*

Table of Contents

Read me!

> *You are reading the **version 4** of this document, **revision 0**. Each revision aims to provide a better reading and learning experience while keeping up to date with the latest features.*
>
> *Latest changes :*
> ***Version 4** adds Module Documentation, and introduces the UIB documentation.*
> *Information was added for the supertools tag and the clone action.*
> ***Revision 0** is the initial revision.*
>
> *This document was made for the version **0.3-RC1** of OpenBST. Features of the language may change between versions.*

Thank you for downloading this document. It will guide you through creating your very own BST Story.

BST stands for Branching Story Tree. It can be much more than a simple branching story, but I like keeping things simple.

The BST scripting interface is Turing complete. (It has IF and GOTO capabilities, as well as data storage and manipulation tools, which makes it Turing complete).

Enjoy!

Part 1 : Static stories

So, you're here to learn how to create BST stuff, right? Let's get started!

BST (Branching Story Tree) is a format that allows you to easily create branching stories, all based on text. You can also create small text-based games using the same format, thanks to the scripting interface provided - we will see about it later. It's powerful!

For now, let's focus on creating our first, simple story. It will not have any kind of scripting and will stay consistent : there is no dynamic content inside it. Such a story is called a Static story.

A Simple Story

> *This will cover static story creation, Text Nodes and basic usage of the OpenBST player. After reading it, you will be able to create static branching stories without any problem. Like a boss.*

The syntax used by the BST is very simple. Create a text file, any editor will do. (Note : Use Notepad++ instead of the regular Notepad application on Windows. The file must be using UTF-8. If you use the Notepad application on Windows, make sure you select Unicode when saving your file.)

The extension does not matter, but you cannot use LibreOffice or Micro$oft Office : you *have* to use a simple text editor. For instance, you can use Notepad or Notepad++ on Windows, or Gedit, Pluma or Mousepad on Linux, but there are many more available.

Nodes

In BST, every part of your story is called a "node". Let's take an example :

Jack is at the entrance of his house. He can either go upstairs, go to his kitchen, go to his bedroom or go to the toilets.

Here, every part of the house will be represented by a node : the entrance is a node, the kitchen is a node, the bedroom is a node… From the entrance node, you will be able to go to the other nodes using options. Options are pretty self explanatory : the reader will be able to go to the next node using any of the options provided by the current node.

Let's create our first nodes. There are multiple types of nodes, but here we will just create a simple Text Node. We will see the other nodes later on, as they are quite advanced.

```
1:This is a first node
```

As you can see, a Text Node is a very simple node that only contains text and options. Save the file, open OpenBST, select "Open File" and choose the file we just saved. You will see the text "This is a first node" shown, along with other information at the bottom. We will

see about them later on. Don’t worry about the buttons at the top, we will also see about them later on.

What happened?

In BST, every node, regardless of its type, has a unique ID; the ID of the node we created here is 1.

We declared our node by using this simple syntax :

```
<id>:Text of the node
```

Every node is declared like this. Your BST story will always start on the node that has an ID of 1 by default.

But what if we want to create a node that has multiple lines of text? No problem! Just add your text like you normally would with any other editor. Here is an example :

```
1:This is

a

multi-line


node ;)
```

Now that we have created our first node, let's create another one. We just repeat the same pattern on a new line.

```
1:This is my first node
Very impressive right?

... okay maybe not.

2:This is my second node

Still not impressive I guess
```

As you can see, the same syntax is used to declare the node, only the content and the node's ID have changed. Again, EVERY node must have a UNIQUE ID.

If you play the file again, you will notice that nothing has changed. Where is our second node?

The program cannot guess where the reader can go next : you have to tell it where we can go next using *options*.

Options

Options tell the program what are the next nodes that the user can choose from. Do note that options are a feature that is specific to Text Nodes, as other types of nodes use different mechanisms.

Let's take our previous example and add an option so that we can go from node 1 to node 2.

```
1:This is my first node
Very impressive right?

... okay maybe not.
:Let's make it more impressive|2

2:This is my second node

Still not impressive I guess
```

You can see that we added a line at the end of the first node. Launching the file with OpenBST will show you that you can have the option to go to node 2, and, if you click on it, the second node will be displayed.

You can of course have multiple options. Note that options cannot have multiple lines of text.

The syntax we just saw is :

```
:Option text that will be shown on the button|<id of the node this
will lead to>
```

Let's create a slightly more complex example :

```
1:First node of awesomeness
:Lame|2
:Cool|3

2:u so rude, i cri everitim

3:Thank you!
:Actually no, it's lame|2
```

You can see that the node 2 has no options : such a node is called a final node. It represents one of the endings of your story, as the user cannot go anywhere else from there.

> *As you may have noticed, OpenBST gives you a small set of options for every final node, even if you did not request any. Said option are :*
> *1) A non-selectable "The End." button*

> *2) A non-selectable button that indicates the ID of the final node the user just landed on*
> *3) An option to exit OpenBST*
>
> *4) An option to restart the entire script.*
>
> *These are the buttons that we also saw before adding any option, since the nodes were then Final Nodes.*

Congratulations, you just created a branching story! That's all you need to know for creating your BST file!

Before we leave, do know that you can put comments inside BST files. Comments are lines that will be completely ignored by OpenBST, as if they never existed. Here is an example

```
1:An example
# that does not show this line
but shows this one
```

You'll see that only "An example but shows this one" will be displayed on screen, but that "that does not show this line" is not shown anywhere. Comments are lines that start with a `#`. They can be literally ANYWHERE inside the file ; they just need to be on a separate line. Do not worry if your BST file contains other occurences of the `#` symbol, as comments only count if the *line* starts with it.

Now that we are all set and you know how to create basic nodes, let's have a look at how you can have more customized nodes and stories using tags.

Markup, colors, title, author, all that good stuff

> *This will cover tagging in all its forms, as used in static stories*

Tags are simple ways to indicate to OpenBST that there is something special about the node, the story or an option.

There are two types of tags in BST :

- Top-level tags. Those have an impact on the entire story. They use the following syntax and are declared before any node has started : `tagName=Tag value`
- Secondary level tags. They only have an impact on nodes and options. For text nodes, they are declared after the content and before the options. For options, they are declared right after the option on a new line, before the next option.

  They use the following syntax : `::tagName=Tag value`

The following tags can be used and are recognized by OpenBST :

- `markup` : This tag can be used as a top level tag to specify the markup language in the entire document, or in a node to either override the top-level tag's value or specify a markup language that only is used in a specific node. The following values

are recognized (case is ignored, so "markdown" and "MaRkDoWn" have the same effect)

    - `none` : Indicates that no markup language is used
    - `html` : The text is formatted in HTML. Do note that the implementation provided by OpenBST is extremely basic.
    - `markdown` : The text is formatted using Markdown
    - `md` : Same as "markdown"

- `author` : The name of the author(s). Top level tag only. Used for some parts in OpenBST, and it is in general better to know who made the story.
- `title` : The title of the story. Top level tag only. Used in OpenBST to provide better information.
- `color` : The color you want to use in a text node or an option. Secondart level tag, used in Nodes and Options. The following colors are available : `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` and `YELLOW`. You can also use any hex-formatted color (`#FFFFFF` for example. This will result in the text being white.)
- `nsfw` : This should be set to “true” if your story is considered “not safe for work”. You know, mature stuff. Top-level only.
- `initialnode` : Stories start at node 1 by default ; if, for some reason, you not to start at another node, you can specify which using this tag. For example, if we want to start at, say, node 15, we would add this tag : `initialnode=15`
- `supertools` : In OpenBST, you have a toolbar that gives useful tools, but those can be a little bit overpowered and may be used for cheating. You can use this tag to restrict which tools can be accessed.
    - `all` : Show all the tools. This is the default value.
    - `hidecheat` : Hide the most powerful and cheating prone tools : Node ID indicator, Jump To Node button, and Variable Watcher
    - `savstate` : Hide everything except save states
    - `savestatenoio` : Show save states only, but exclude the Import and Export functions.
    - `none` : Hide everything.

    Do note that the Close button cannot be hidden. This tag will have different effects on other players.

Here is an example that uses most of these tags.

```
title=An Awesome Example
author=BST Team
markup=markdown

1:An awesome **pink** example!
::color=PINK
:I prefer red!|2
::color=RED
:I prefer cyan!|3
::color=CYAN

2:There you go, you red-lover
::color=RED

3: There you go, you cyan-lover
::color=CYAN
```

And there you have it! Next, we will see how to create scripts, dynamic options, Logical Nodes and Virtual Nodes!

Part 2 : Dynamic stories (Scripting)

> *This will cover all there is to know about dynamic stories, except Virtual and Logical nodes*

Okay! Now that we're done talking about static stories, let's begin working on what makes BST awesome : Scripting!

Since scripting is such a vast subject, we'll go step by step. First, let's create a very simple story that uses some basic scripting.

Basic scripting : options scripts, actions and variables

> *This will cover Script in Options (SiO), variables and variable inclusion in Text and Virtual Nodes.*

I have to admit something. I lied to you. The syntax for options is a little bit more complicated than what I showed. Do know that the one you saw earlier works perfectly, but there is another syntax that is slightly more advanced.

```
1:Example node
:Option 1|2|{incr:var}

2:Another example node
```

Try to run it! You'll see that you have an option that does not look different from what we saw in the first part of this tutorial. It's normal : scripts are silent. They only speak when an error occurs. The syntax is the same as the previous one, with the addition of `|{incr:var}`.

Here, the | is used to separate the ID of the next node with the rest of the option, which is a script.

A script is a succession of instructions. An instruction has a function name and a description. Here, the instruction is `{incr:var}`, with the function name being `incr` and the description being `var`. The `incr` function is a function that increments a variable.

A variable is a pretty much like a box. The box has a name on it, and we can put *something* inside it. Most of the instructions operate on a variable : here the `incr` function increments the variable `var` by one, where "var" is the name of the box, and the content of the box gets incremented by one. In BST, variables can contain either integers (both negative and positive) or strings (like "Abcdefg123 456 abba").

You can put multiple instructions in an option. For example, we could have put `:Option 1|2|{incr:a}{incr:b}{incr:c}`, and this would have incremented, in order, the variables "a", "b" and "c". Sometimes, you will have to be careful about the order of your instructions, so just make sure the order is logical.

So that's it : when we choose the option "Option 1", the variable "var" gets incremented by 1. But can we actually see the content of the variable var in our story?

We can! It's not compulsory, but sometimes, you'll need to show the variable to the user. This can be useful when the variable is something like an health point counter, where the user would be interested in seeing how many points he has left.

Let's build an example that has a single node which shows the content of the variable "a" and an option that increments said variable.

```
1:a = ${a}
:a+1|1|{incr:a}
```

Also, I have not said it to you, but yes, you can say that the next node is the same node : here the option 1 will continue on the node 1. It's perfectly possible, but is useless when doing Static stories, as it only makes an infite loop. Here, it will be very useful, since we get to see the result of incrementing the variable a.

As you may have seen, the syntax for including a variable's value inside a Text Node (or a Virtual Node, we will see about them later) is `${var}`, where "var" is the name of your variable.

Go on and run this story in OpenBST to see the result live! "a" starts at 0, then gets incremented by 1 every single time you click on the button!

And that's it! You just created your first dynamic story! Before we go deeper, here is a list of useful functions :

- `incr:<variable>` : Increments the variable named "<variable>" by 1. Only works with numbers

- `decr:<variable>` : Decrements the variable named “<variable>” by 1. Only works with numbers.
- `set:<variable>,<value>` : Sets the variable “<variable>” to “<value>”. The value can be an integer or a string.
- `clone:<variable to clone>,<variable to set>` : Clones the value of “<variable to clone>” to “<variable to set>”. This is useful for duplicating
- `<operation>:<putIn>,<a>,<b>` : Sets the variable “<putIn>” to the result of “<operation>” with “<a>” and “<b>”. The values can be normal numbers or variable names.
  “<operation>” can be :
  - `add` for an addition (a+b)
  - `sub` for a substraction (a-b)
  - `mul` for a multiplication (a*b)
  - `div` for an division (a/b)
- `exit:` : Makes the program crash instantly. No other information is given, the program just quits.
- `input:<variable>,Text of the dialog prompt` : Shows a dialog box with a message (here, “Text of the dialog prompt”) that asks for an input. Only works with strings.

> *These script functions that do something are internally called “Script Actions”.*

So, now you know how to create *some* scripting stuff. But all that is quite limited : what if I want an option to go to a different node is a variable is, say, equal to one?

Next node definers (NND), conditional options

A very powerful tool for this is a Next Node Definer. By the way, you already know how to use one!

Okay, once again I lied to you. Let’s take a very basic option.

```
:I am an option|2
```

The 2 is actually something called a Next Node Definer, also called NND because that’s much shorter. Here, we use a Static NND, which never changes. There is another type of NND called “if-NND”. An if-NND is a next node definer that determines what the next node is depending on the validity of a statement.

In short, an if-NND decides what the next node will be depending on if a condition is true or not.

But how can we define said conditions? We only saw functions that *do* things, not functions that *check* if something is true!

For this, we use a different type of functions called Checkers. Let’s see how they work in an if-NND :

```
1:This is an example.

a = ${a}
:a+1|1|{incr:a}
:Go|2,3[equ:a,3]

2:a was equal to 3

3:a was not equal to 3
```

This is fairly simple. On our first node, we have a text that indicates the value of the variable “a”. Then, we have an option to increment “a”, and a Go option. The Go option will move us to node 2 is “a” is equal to 3, or node 3 if “a” is not equal to 3.

The syntax of an if-NND is :

```
<nodeIfTrue>,<nodeIfFalse>[checker]
```

Checkers follow the same syntax as the functions we saw previously, except that the symbols `{` and `}` are replaced by `[` and `]`.

What if we want our option to be available only when we decide it can?

We use conditional options! Conditional options are options that have a checker. If the checker returns true, the option will be available, if it returns false, the option won’t be shown.

Let’s take our previous example and say that we cannot get over 3. For this, we just have to make our first option, the one which increments “a”, be available only if a is less than 3.

```
1:This is an example.

a = ${a}
:a+1|1|[less:a,3]{incr:a}
:Go|2,3[equ:a,3]

2:a was equal to 3

3:a was not equal to 3
```

You’ll see that you cannot go over 3 using this example.

The option has changed a little bit :

```
:a+1|1|[less:a,3]{incr:a}
```

You can see that we added a checker right before the incr function. The position of the checker does not matter, but there can only be *one* checker.

You can of course combine the if-NND and the use of checkers or script functions inside an option.

Here is a list of useful checkers :

- `equ:<var1>,<var2>` : Checks equality between <var1> and <var2>. While <var1> HAS to be a variable, <var2> can either be a variable or a value.
- `greater:<var1>,<var2>` : Checks if <var1> is greater than <var2>. While <var1> HAS to be a variable, <var2> can either be a variable or a value.
- `greaterequ:<var1>,<var2>` : Checks if <var1> is greater than or equal to <var2>. While <var1> HAS to be a variable, <var2> can either be a variable or a value.
- `less:<var1>,<var2>` : Checks if <var1> is less than <var2>. While <var1> HAS to be a variable, <var2> can either be a variable or a value.
- `lessequ:<var1>,<var2>` : Checks if <var1> is less than or equal to <var2>. While <var1> HAS to be a variable, <var2> can either be a variable or a value.

Okay. Now we know how to do a lot of things, and we can make our stories dynamic. Next, we'll see what are the two other types of nodes : Virtual Nodes and Logical Nodes.

Part 3 : Deeper inside scripting

> *This will cover Virtual and Logical nodes*

So, we are now able to do some stuff on one option; but let's be honest, while that is pretty useful for only one or two increments, it can get really messy when we get a lot of functions.

Enter the world of Logical nodes!

Logical Nodes

Logical nodes are scripts : pretty much what you used inside the options for scripting except way more powerful.

Simple Logical Nodes & if-NNDs in Logical Nodes

Let's have a look at a very simple logical node.

```
1:&
add:a,10
incr:a
incr:a
:2

2:Value of a is ${a}
:Do it again!|1
```

As you can see, a logical node is nothing more than a script in a dedicated node rather than all squashed in an option.

The first node of the example is a logical node, as indicated by the special beginning. Every node type have this beginning, with the node's id, a colon (`:`) and a special character that defines the type of the node. Here is a list of all the special characters :

- `&` : Logical node
- `>` : Virtual node (more details about them later)
- No special character : Text node

Most types support having text right after that character : this is the case for Text Nodes as they do not have any special characters, and Virtual Nodes. Again, more details about them later.

Every line of a Logical Node is a function. Empty lines are ignored. They are executed in the order they are written : here, we first set the value of a to 10, we increment it twice and decrement it once.

A line that starts with a colon is a special line in Logical Nodes. Here, the user cannot choose where to go, it's up to the logical node to decide which node is next. Lines that start with a colon are Next Node Definers inside these Logical Nodes, meaning that when this line is reached, it will be used to see where to go next. You can use both static NNDs (as seen in the example) or if-NNDs, or any other NND. We also say that the next node's ID is "returned", that the Logical Node "returns" the next node's ID.

Let's take another example.

```
1:Press an option to start
:a = 1|10|{set:a,1}
:a = 2|10|{set:a,2}
:a = 3|10|{set:a,3}

2:a was 1

3:a was 2

4:a was 3

10:&
:2,-1[equ:a,1]
:3,-1[equ:a,2]
:4,1[equ:a,3]

# We return to 1 if for some reason a isn't 1, 2 or 3 (this shouldn't
happen normally)
```

In this example, we use a special feature of if-NNDs that is only available in Logical Nodes. By specifying -1 instead of an actual node, the program will just continue going through the Logical Node.

Here, once we reach node 10, we first check if a = 1 ; if yes, we go to node 2, if no, we continue. The same thing happens with the next NND ; the last one is just here to make sure we have a node to go to.

If the node does not specify a node, OpenBST will show an error.

If statements

It is possible to have "if statements" inside a Logical Node. We do so by using the following syntax :

```
[checker]?{ifyes}{ifyes}...:{ifno}{ifno}...
```

We just use a checker as a condition, some instructions that will be executed if the checker says "true", a colon (`:`) and some instructions that will be executed if the checker says "false". Here is an example that tells if a number is positive, negative or equal to 0 :

```
1:Press START to start
:START|2

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}:
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:The number ${a} is negative
4:The number ${a} is positive
5:The number ${a} is 0
```

If the number is strictly negative, we set b to 1. If it is not striclty negative, which means that it is either 0 or positive, we set b to 2. Now, if the number is 0, we set b to 3. After that, we go to the node 3 if b is 1, or continue if it has another value. At this point, its value can be either 2 or 3, so we just need one if-NND : if b is 2, we go to node 4 ; if it's not 2, it means that it has to be 3, we can jump to the node 5 right away.

Play around by changing the line `set:a,10` and you will see that the node after you press START will be correct.

As you can see, it is not necessary to have instructions executed if the checker returns "false". However, a colon is still necessary. This would work :

```
[equ:a,0]?{set:b,3}:
```

But this wouldn't work :

```
[equ:a,0]?{set:b,3}
```

And that's all you need to know about Logical Nodes!

As you may have noticed, there are quite a lot of repetitions with the nodes 3, 4 and 5 in the example. There is a type of node specifically made to contain text that will be included at multiple places : Virtual nodes.

Virtual nodes

You can see virtual nodes as small pieces of paper that you can glue anywhere inside Text Nodes. Here is an example of a virtual node that shows the text “Hello” on top of every node :

```
1:${>10}
My name is Wilbert
:Nope, it’s Robert|2

2:${>10}
My name is Robert
:Nope, it’s Albert|3

3:${>10}
My name is Albert
:Nope, it’s Wilbert|1

10:>Hello!
```

For the rest, virtual nodes are exactly like Text Nodes, except that they do not have options, and that the program will give an error if you try to go to one of them : they are not made for that. They are built to be included in other nodes. Virtual nodes can also contain other virtual nodes or variables ; this is perfectly possible :

```
1:Virtual${>2}!

2:>node${>3}

3:>ception
```

As you can see, to include the text of a virtual node, you just need to do the same as if you wanted to include a variable, but use the node’s ID instead of the variable’s name and put a `>` between the `{` and the ID.

Virtual nodes + Logical nodes = <3

You can also include a logical node inside a node ; you just need to use `${&id}` instead of `${>id}` . The way it works is that the next node that the logical nodes returns will be used as a virtual node. For example, this works :

```
1:Text : ${&2}

2:&
:3
```

```
3:>Hello World
```

This does the same thing as if you put `${>3}` instead of `${&2}` as the Logical Node will always return the node 3.

Now that we have all the power of Logical Nodes and Virtual nodes, we can simplify this example :

```
1:Press START to start
:START|2

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}:
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:The number ${a} is negative
4:The number ${a} is positive
5:The number ${a} is 0
```

… into something that does not contain any repetition :

```
1:Press START to start
:START|10

10:The number ${a} is ${&2}

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}:
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:>negative
4:>positive
5:>0
```

And that's it! You now know everything about Logical and Virtual nodes! Here are some more examples if you need some :

```
1:${&2}, World!
:Hello|1|{set:a,0}
:Goodbye|1|{set:a,1}
:Farewell|1|{set:a,2}
:Screw you|1|{set:a,3}

2:&
:3,-1[equ:a,0]
```

```
:4,-1[equ:a,1]
:5,6[equ:a,2]

3:>Hello
4:>Goodbye
5:>Farewell
6:>Screw you
```

In this example, you can tell the world “Hello”, “Goodbye”, “Farewell” or “Screw you”. We can also optimize it by making the options conditional, so that the user cannot select the text that is already selected for the node ; for this, we just need to add a “not” checker to every node :

```
1:${&2}, World!
:Hello|1|[not:a,0]{set:a,0}
:Goodbye|1|[not:a,1]{set:a,1}
:Farewell|1|[not:a,2]{set:a,2}
:Screw you|1|[not:a,3]{set:a,3}

2:&
:3,-1[equ:a,0]
:4,-1[equ:a,1]
:5,6[equ:a,2]

3:>Hello
4:>Goodbye
5:>Farewell
6:>Screw you
```

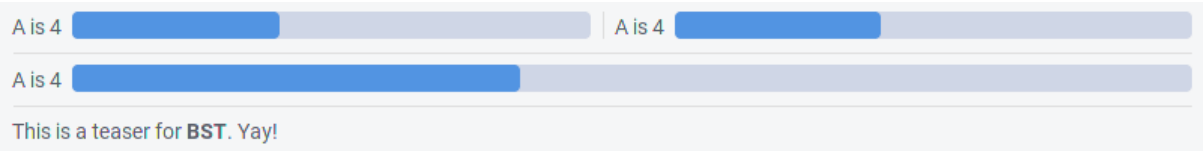
Modules documentation

> ***A foreword on modules :***
>
> *Modules are very useful parts of BST that were separated from the rest of the language due to their complexity or the fact that they can be tricky to add in a Player. All the modules described here work best in OpenBST – and while we will try to help other players to keep up to date, we cannot guarantee that these will work on them.*
>
> *All modules use a similar pattern for the tags, functions and checkers they use : they start using an identifier, an underscore, and the rest is pretty normal.*
>
> *Do note that modules tend to use tags to store data : as such, it is recommended that you avoid using tags that start with two underscores ("__"). Modules use the following pattern : "__modulename_..." with the rest being whatever the module needs.*

Modules are very useful. They give you ways to go further with your story, and are one of the key elements for games. This part is dedicated to documenting everything that there is to know on these powerful tools.

Module : User Info Bar (or User Interface for BST)

UIB is a set of functions and tag that allow you to create a fully customizable bar that displays various information. In this chapter, we will see how to create the following bar.



Step 1 : uib_layout

The first thing needed when using UIB is the `uib_layout`, a tag that contains all the information about the different components used with UIB.

Let's see the `uib_layout` used in the image.

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
```

All the components are separated by commas `,` (they can also be separated by `;`). Here is a complete list of all the components that exist :

- `tb` : a shortcut for `t,b` that is much more consistent and accurate visually.
- `vs` : a vertical separator

- `ln` : this indicates that the next component will be on a new line
- `hs` : a horizontal separator. Usually, this would be placed alone on a separate line.
- `t` : a text component. Its value can be formatted using the markup language specified in the top-level tag `markup`, even if you're not using a node.
- `b` : a bar component. It has the following properties :
  - `max` : the maximum value of the bar
  - `min` : the minimum value of the bar
- `gu` : a vertical gap that is a very useful replacement for `vs` in situations where a gap would be more appropriate. The letters mean "gap unrelated"

Components can have a value (for example, the value of a text component is the text, and the one of a bar component is a number) and/or properties.

The `uib_layout` tag can also contain much more advanced properties, but we'll have a look at them later. For now, this model works in most situations.

Step 2 : Initialize UIB

UIB needs to be initialized before you touch anything else. Usually, you would do this on the first node which would then be a logical node, or on any other logical node. Just remember to initialize UIB before touching anything else or else very bad things will happen.

For this example, we make the first node a logical node. The function that initializes UIB is `uib_init:`, so we only need to add it to our logical node :

```
1:&
uib_init:
```

> *Under the hood, this command reads the UIB tag, interprets it, and translates it into actual visual components.*

Now that UIB is initialized, we need to set all the values and properties we need…

Step 3 : Setting properties and values

This is fairly easy to do. Before we jump straight into setting all the values, you have to understand how ID attribution works in UIB.

Basically, every component that can have a value or a property is identified by the component's usual text representation (for a text component, this would be `t`) followed by a number. This number is unique to the component and is determined depending on the position of the component in the tag. For instance, if our layout tag was `t,t,tb` the first text component would have an ID of `t0`, the second one an ID of `t1`, the third one an ID of

`t2`, while the bar would have an ID of `b0`. Do note that when you use the component `tb`, you're actually using *two* components : a `t` and a `b`, so you will have to refer to them as if were using `t,b` instead of `tb`.

> *The usage of tb instead of t,b adds various visual improvements that are very important in multiple situations. Just use this notation instead of the other, really.*

Now that this is done, we can continue to setting our values. Here is an example for the first `tb` :

```
uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a
```

This may look complicated, but it really isn't. The `uib_set` function is used whenever you need to set something's value. The syntax is :

```
uib_set:<component id>,Value (that can contain commas)
```

The value depends on what you want to set :

- For a text component, the value can be text (`Hello World!`), a virtual node (`>nodeid`, here it is `>5`) or a logical node (`&nodeid`)
- For a bar component, this can be a number (`1` `2` `38` `920`) or a variable (`variablename`, here we use the variable `a`)

When you use a variable, a logical node or a virtual node, the value will be automagically updated whenever needed. It is not recommended to use a logical node that changes variables though, as it may be called at random times and result in strange behavior.

Here, we set the value of `t0` to the virtual node `5`, and the value of `b0` to the variable `a`.

Setting properties is done through the following syntax :

```
uib_setprop:<component id>,<property name>,<value>
```

The value here can be text or numbers, but it *cannot* be a variable or a node of any kind.

We repeat this pattern for all the other components and create a virtual node 5. Our BST file now looks like this :

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
markup=markdown

1:&
uib_init:

uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a
```

```
uib_set:t1,>5
uib_setprop:b1,max,10
uib_set:b1,a

uib_set:t2,>5
uib_setprop:b2,max,10
uib_set:b2,a

uib_set:t3,This is a teaser for **BST**. Yay!

# Warning : this file will crash, as the logical node doesn't lead to
anywhere!

5:>A is ${a}
```

The `markup` tag was added so that the text `t3` can make use of it.

Step 4 : Making the bar visible

By default, UIB is invisible. We just have to make it visible by using :

```
uib_setvisible:true
```

We can of course replace true by false if we want to hide UIB.

Now, we just have to add a node that allows us to control the variable `a`, and we're done! Here is the complete example :

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
markup=markdown

1:&
uib_init:

uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a

uib_set:t1,>5
uib_setprop:b1,max,10
uib_set:b1,a

uib_set:t2,>5
uib_setprop:b2,max,10
uib_set:b2,a

uib_set:t3,This is a teaser for **BST**. Yay!

uib_setvisible:true
:2

2:Testtt ${a}
```

```
:a++|2|[less:a,10]{incr:a}
:a--|2|[greater:a,0]{decr:a}

5:>A is ${a}
```

And that's it! What we saw is just very basic layout.

Step 5 : Using advanced features

UIB supports a grid mode. Each component will be in a separate cell (except if you use the specific advanced options), and you can specify special properties for each column. Here is an example :

```
uib_grid=[][grow][][][grow]
uib_layout=tb,gu,tb,nl,tb,gu,tb
```

As you can see, each column is represented by square brackets []. If you wish to add properties to a column, you can put them inside the brackets (multiple properties can be separated by commas.

A list of all the properties is available at the following link : http://www.miglayout.com/cheatsheet.html

Look at the section that describes column properties : everything described there can be used in UIB.

Advanced mode is another powerful tool where you can fully control what is displayed using MigLayout component. Here is an example :

```
uib_advanced=true
uib_layout=t:alignx right, growx;t:alignx left, growx
```

This is a layout with a text component aligned to the right and that fills up maximum space, and another text component aligned to the left (which is the default anyway, so specifying alignx left was not compulsory) and also fills up maximum space. The fact that we specified two growx will make the two text components have approximately the same size.

You can use all the component properties from the cheatsheet linked above. You can also use both Advanced Mode and Grid Mode. However, this tutorial does not aim to give a complete description of it all, as it would be way too much work, due to the almost infinite possibilities of UIB.

Note : this module documentation is updated quite often, so check back here for additional features! :-)