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 2** of this document, **revision 3**. Each revision aims to provide a better reading and learning experience while keeping up to date with the latest features.*
>
> *Latest changes :*
> ***Version 2** introduces the second part dedicated to scripting, along with various visual improvements.*
> ***Revision 3** fixes missing Source Code styling for some examples.*
>
> *This document was made for the version **0.1-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, and launch 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.

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 you leave this first part, do know that you do not have to relaunch OpenBST every single time you want to reload the file : you can easily do that by right-clicking the text of the current node in OpenBST, and selecting the option that starts with Reload. This will instantly close and reopen OpenBST.

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, YELLOW. You can also use any hex-formatted color (#FFFFFF for example. This will result in the text being white.)

Here is an example that uses all 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*

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.
- `<operation>:<putIn>,<a>,<b>` : Sets the variable "<putIn>" to the result of "<operation>" with "<a>" and "<b>". "<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.