README.md

Segment Intersection Sweep Line Algorithm

1. Introduction

1.1 Binary Search Tree

Code: tree.py

Test script: test.py

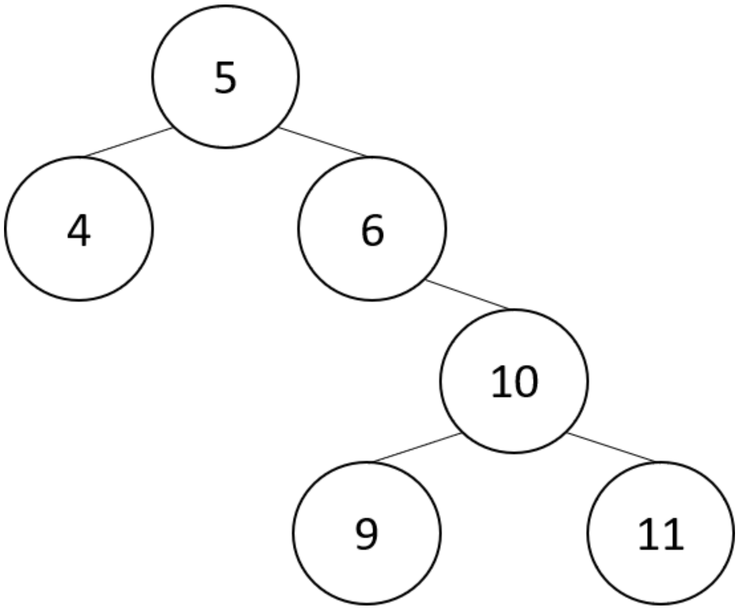
1.1.1 Simple BST

A binary search tree is a data structure that groups nodes. The first node inserted becomes the root, and then the next value inserted will be added in the left (if the value is smaller that the root/parent node) or in the right (if the value is smaller that the root/parent node) child node of the node that is a leaf (no children).

Using the code, an example tree would be the following:

```
>>> from tree import BST
>>> tree = BST()
>>> tree.insert(5)
>>> tree.insert(4)
>>> tree.insert(6)
>>> tree.insert(10)
>>> tree.insert(9)
>>> tree.insert(11)
>>> tree.printBST()
4
5
6
9
10
11
```

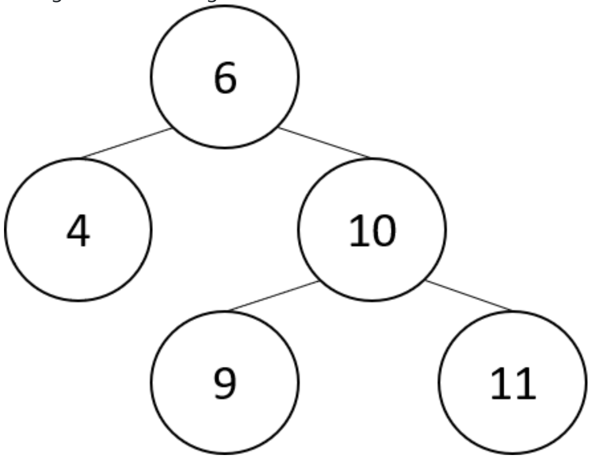
Which gives us the following tree:



If, for example, we delete the root node as below:

```
>>> tree.delete_value(5)
>>> tree.printBST()
4
6
9
10
11
```

Giving us the following tree now:



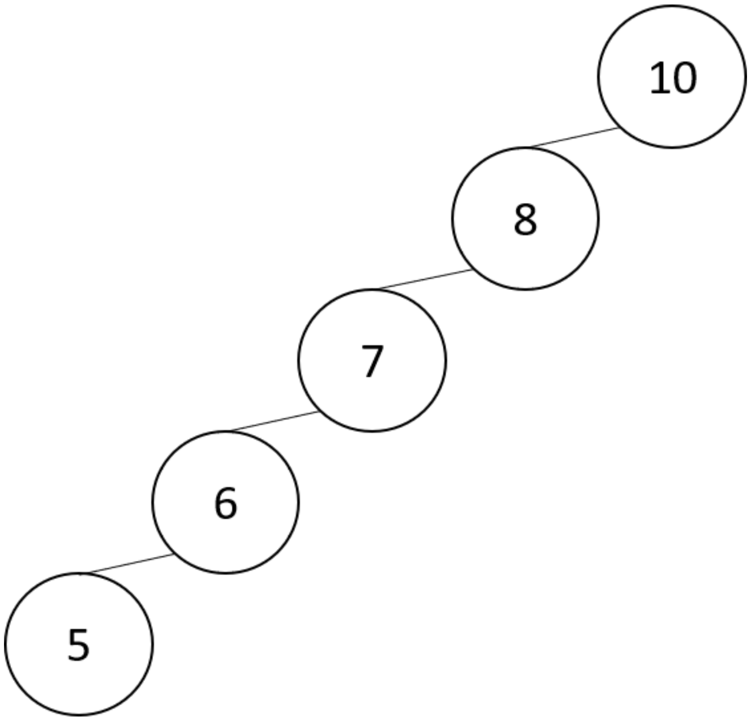
Note: the printBST() function prints an inorder traversed tree, so if we see the values in order, our tree works fine.

1.1.2 Balanced BST

Using this idea, if we simply create the following:

```
>>> from tree import BST
>>> tree2 = BST()
>>> tree2.insert(10)
>>> tree2.insert(8)
>>> tree2.insert(7)
>>> tree2.insert(6)
>>> tree2.insert(5)
>>> in_array = BST.inorder(tree2.root)
[5, 6, 7, 8, 10] # same output as printBST(), inorder
```

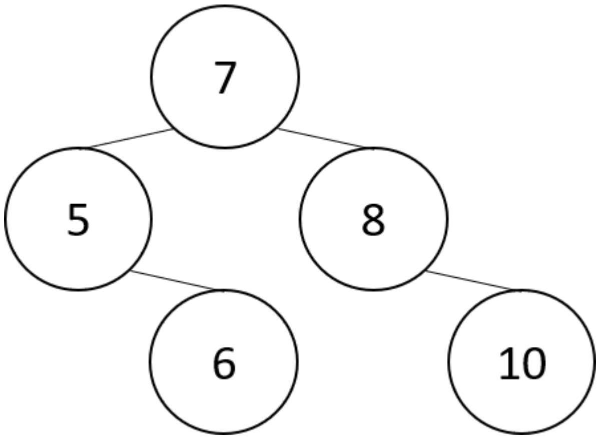
We have this tree:



Which, in theory is **unbalanced**: a balanced BST is one where their left and right subtree differ in height by at most 1. In the above tree, the left subtree has height is 4 and the right subtree has height 0. To balance it, we do the following:

```
>>> bTreeRoot = BST.getBalancedBST(in_array, 0, len(in_array) - 1)
>>> pre_array = BST.preorder(bTreeRoot)
[7, 5, 6, 8, 10] # preorder
```

And now we have a balanced BST.



1.2 Point Binary Search Tree

Code: ptree.py

Test script: ptree-test.py

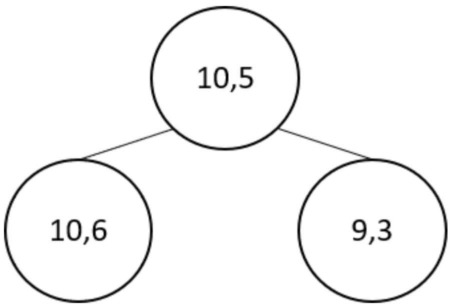
Now we changed the basic data structure to store Point objects instead of simple numbers. BST uses **comparisons** which we need to override for Point objects so that the three stores left and right values (points) following the rule below.

```
(p1 < p2) -> if p1.y > p2.y or if p1.y == p2.y and p1.x < p2.x
```

Now if we test it as follows:

```
>>> from ptree import BST
>>> from glibrary import Point, Vector, Line
>>> ptree = BST()
>>> ptree.insert(Point(10, 5))
>>> ptree.insert(Point(10, 6))
>>> ptree.insert(Point(9, 3))
>>> in_array = BST.inorder(ptree.root)
>>> print(in_array)
[(10, 6), (10, 5), (9, 3)] # now 'sorted' order means bigger y's in front
```

Which gives us the tree below.



which we can also balance.

```
>>> bTreeRoot = BST.getBalancedBST(in_array, 0, len(in_array) - 1)
>>> pre_array = BST.preorder(bTreeRoot)
>>> print(pre_array)
[(10, 5), (10, 6), (9, 3)] # preorder
```

Which in reality ends up being the same result because ptree is already balanced.

2. Data Structures Needed

2.1 Event Binary Search Tree

Code: etree.py

Test script: etree-test.py

This is one of the **actual** data structures that we will use for the Segment Algorithm, we will call it Q formally. The difference now is simply that instead of storing **Point Objects**, the BS tree will store **Event Objects**, but the rule of insertion will still be the same.

```
(p1 < p2) -> if p1.y > p2.y or if p1.y == p2.y and p1.x < p2.x
```

An **Event Object** is simply a group of 3 values:

- `point` : Point Object

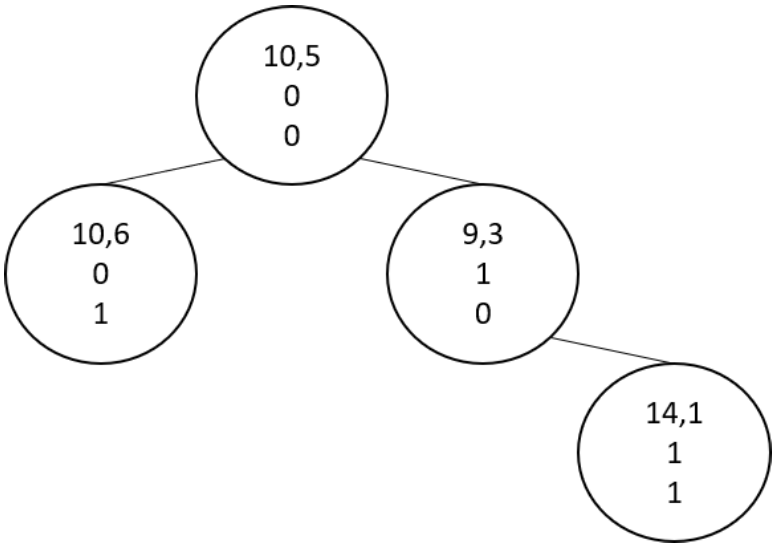
- `seg` : Integer that determines the index of the segment that contains this point.
- `pos` : Integer that determines the index position of the point in its segment (0 starts the segment).

Every **Event** instance will represent the `value` of a Node inside the tree.

This tree will act as a **Queue of Events** but will be faster because of the BST nature when searching. Now if we construct an example as the following:

```
>>> from etree import *
>>> from glibrary import Point, Vector, Line
>>> etree = Q()
>>> e1 = Event(Point(10, 5), 0, 0)
>>> etree.insert(e1)
>>> e2 = Event(Point(10, 6), 0, 1)
>>> etree.insert(e2)
>>> etree.insert(Event(Point(9, 3), 1, 0))
>>> etree.insert(Event(Point(14, 1), 1, 1))
>>> in_array = Q.inorder(etree.root)
>>> print(in_array)
[E[Point: (10, 6) Seg: 0 Pos: 1], E[Point: (10, 5) Seg: 0 Pos: 0], E[Point: (9, 3) Seg: 1 Pos: 0], E[Point: (14, 1) Seg: 1 Pos: 1
```

Which involves two line segments: s0 that goes from (10, 5) to (10, 6), and s1 that goes from (9, 3) to (14, 1). This event tree will look like the diagram below.



You delete nodes from an etree by **Point Object** values or by **Node Object**.

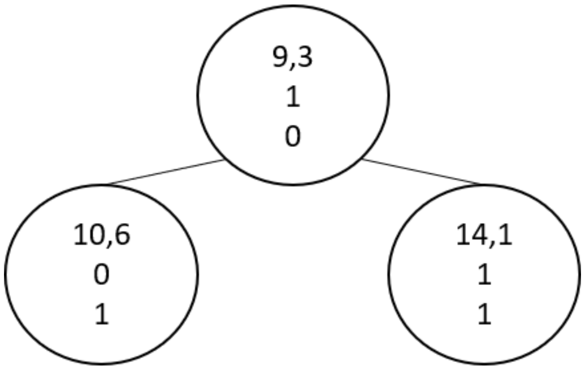
Delete by Point

```
>>> etree.deleteValue(Point(14, 1))
>>> in_array = Q.inorder(etree.root)
>>> print(in_array)
[E[Point: (10, 6) Seg: 0 Pos: 1], E[Point: (10, 5) Seg: 0 Pos: 0], E[Point: (9, 3) Seg: 1 Pos: 0]]
```

Delete by Node

```
>>> etree.deleteNode(etree.root)
>>> in_array = Q.inorder(etree.root)
>>> print(in_array)
[E[Point: (10, 6) Seg: 0 Pos: 1], E[Point: (9, 3) Seg: 1 Pos: 0], E[Point: (14, 1) Seg: 1 Pos: 1]]
```

Giving us the tree below.



2.2 Line Status Binary Search Tree

Code: ttree.py

Test script: etree-test.py

This is the **other specific data structure** that will be used for the Segment Algorithm, and we will call it T formally.

T will also be a BST, but in this case instead of storing **Event Objects** as node values, it will store **Segment Objects** as node values. The rule for inserting a segment in T will be a bit more complex, so the *insert()* function will need the parameters:

- `t1` : point representing the active event (P in the algorithm).
- `s1` : segment 1, contains start and end point objects.

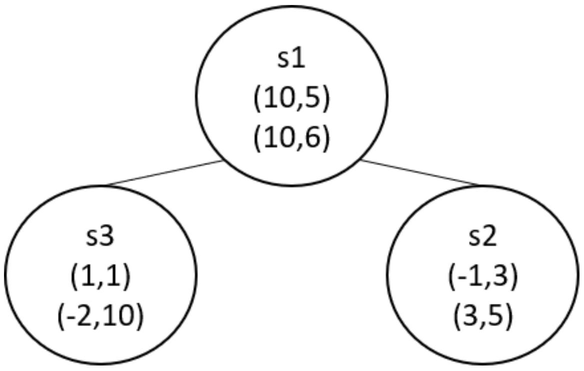
The comparison rule that T will do to insert `s1` will work as follows: in order for the T tree to know whether `s1 < si` , we need to build three lines: the first will be a **horizontal line** `tline` that passes at level t1.Y value and is actually the *sweep line* of the algorithm; the second, `line1` that passes through s1.start and s1.end points; and `line2` , which passes through si.start and si.end points. Then, we compute `hit1` which will be the **point of intersection** between `line1` and `tline` , also we compute `hit2` which will be the **point of intersection** between `line2` and `tline` . Finally, s1 is less than si if `hit1` .X < `hit2` .X.

> **In other words, let s1 and si be two segments, s1 < si if the x coordinate of s1 line intersection with the sweep line at t1 (or P) is less than the x coordinate of si line intersection with the sweep line at t1 (or P).**

The implementation of this T tree is used like follows.

```
>>> from ttree import *
>>> tline = T()
>>> t1 = Point(5,15)
>>> s1 = Segment(Point(10,5), Point(10,6))
>>> s2 = Segment(Point(-1,3), Point(3,5))
>>> s3 = Segment(Point(1,1), Point(-2,10))
>>> tline.insert(s1, t1)
>>> tline.insert(s2, t1)
>>> tline.insert(s3, t1)
>>> in_array = T.inorder(tline.root)
>>> print(in_array)
[S[start:(1, 1) end:(-2, 10)], S[start:(10, 5) end:(10, 6)], S[start:(-1, 3) end:(3, 5)]]
```

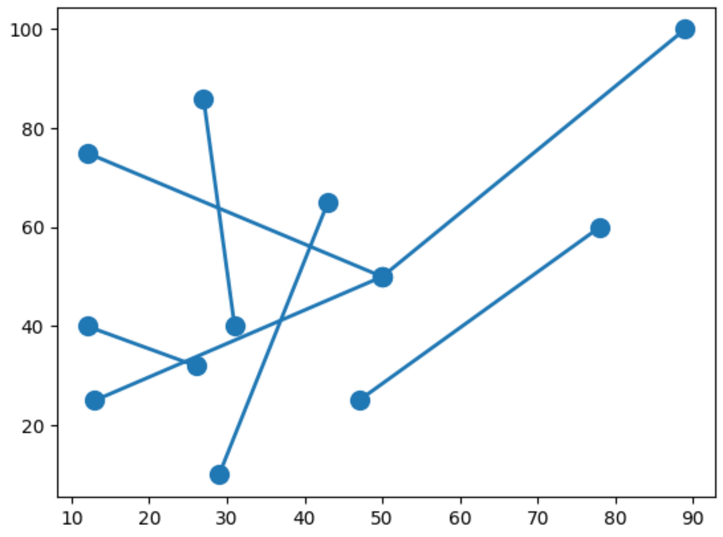
Which visually is the tree below.



3. Implementation

3.1 Input

The input from the file 0.in is plotted as follows.



Handy Links

- BST Basics
- BST Balanced
- BST to balanced
- Traversal inorder
- Tree Traversals
- Bentley-Ottmann Algorithm
- Successors and Predecessors in BST