

Homework Assignment 4

General information:

Responsible teaching assistant: Ofir Yaish.

Students' Questions regarding the assignment will be answered only in the HW4 forum.

If necessary, office hours are on Tuesday, 16:30-17:30. Please send an email to coordinate prior (yaishof@post.bgu.ac.il).

Pay Attention:

- Please read the entire assignment before starting to solve it.
- You are provided with three template files: Company.py, CompanyNode.py, and CompanyTree.py, for three classes required to implement: Company, CompanyNode, and CompanyTree, respectively.
- You are not allowed to change the names of the Requested files.
- In this assignment, you cannot assume the input is valid for every question unless it is written explicitly.
- You can import the copy library and import the class you implemented for other files (modules).
- Do not use other external libraries to solve this exercise.
- As mentioned, some tests will be visible for your convenience, and others will not.
- You must code the assignment by yourself. A Similarity test will be performed automatically, and similar codes will be automatically graded as 0.
- You can add auxiliary functions and methods (including class and static methods).
- You can assume that the instance attributes of a class object will not be modified outside the class. This means that any modification will be performed using instance methods.
- Good luck!

Task description

You are requested to create a system that simulates Companies on their financial side. You will need to implement **Company**, **CompanyNode**, and **CompanyTree** Class:

- The **Company** Class will represent a single Company.
- The **CompanyNode** Class represents a single company with relations with other companies (i.e., it can be a parent company with multiple subsidiary companies or a subsidiary company by itself).
- The **CompanyTree** Class represents a Tree of companies with relations between them. It has one company as root, the Holding company, and it can have subsidiary companies.

Note:

1. A holding company (the root of the Company Tree) can have no subsidiary companies.
2. A subsidiary company can have multiple subsidiary companies as well.

For example, the following Figure can describe a Company Tree of the Alphabet Company:

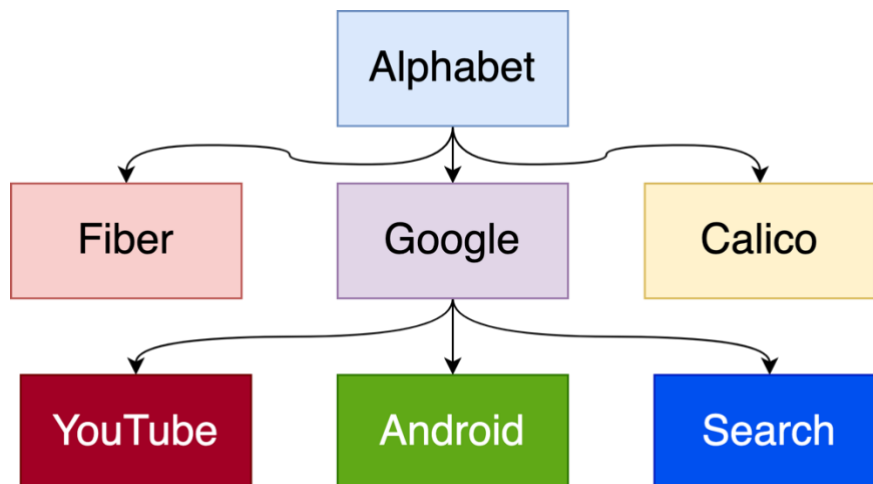


Figure 1

The Company Class

The Company Class represents a company. Those are the following attributes that you are required to define in the class:

Instance attributes:

name – of type *String* and represents the company name. The company name is comprised of words (in English) and spaces that spread the words (single space between two words). A word is a sequence of at least two English letters. In addition, The Company name must start with an upper-case letter. For example, “NVIDIA Corporation” is a valid company name.

stocks_num – of type *int* and represents the number of stocks the company has. The number must be positive.

stock_price – of type *int* or *float* and represents the price of a single stock. The price must be positive.

comp_type – of type *string* and represents the company type. The string must meet the same conditions as the string of the name attribute.

Class attributes:

_comparison_type – of type *string* and represents the criterion on which the compression operators of the class are based. There are 3 optional Criteria:

1. "net value" – the criterion is based on the market cap of the company. The market cap is defined as **stocks_num * stock_price** of the company. This is the default criteria.
2. "stock num" – the criterion is based on the **stocks_num** of the company.
3. "stock price" – the criterion is based on the **stock_price** of the company.

Note:

1. You are not allowed to change the attributes names or add additional ones.
2. The **_comparison_type** with one underscore defines a protected attribute. You can read about protected attributes in Python [here](#).

You are required to implement the following methods in Company Class:

Method	Description
<code>__init__(self, name, stocks_num, stock_price, comp_type)</code>	The initializer receives the company name, number of stocks, price of a single stock, and company type. The method initializes the new Company instance with suitable values. If one of the inputs does not follow the attribute constraints, the program should raise <code>ValueError</code> (you are free to choose any error message). Note: You are not allowed to assume any assumption on the inputs.
<code>net_worth(self)</code>	The method computes and returns the market cap of the company. The market cap is defined as <code>stocks_num * stock_price</code>
<code>set_name(self, name)</code>	The method updates the company name if the new name follows the name attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code> . Otherwise, return <code>False</code> .
<code>set_stocks_num(self, stocks_num)</code>	The method updates the company number of stocks if the new number follows the <code>stocks_num</code> attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code> . Otherwise, return <code>False</code> . Note: The method is not changing the company's

	market cap. Therefore, you might be required to change the price of a single stock.
<code>set_stock_price(self, stock_price)</code>	<p>The method updates the price of a single stock if the new price follows the <code>stock_price</code> attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code>. Otherwise, return <code>False</code>. Note: The method <u>might change the company's market cap</u>. The number of stocks the company holds after the update must be the maximal one such that the new company market cap is not higher than the previous market cap (before the update).</p> <p>Note: If the new price is higher than the market cap (before the update), return the Boolean value <code>False</code> without updating the company.</p> <p>Remember: the number of stocks must be a positive integer.</p>
<code>set_comp_type(self, comp_type)</code>	The method updates the company type if the new type follows the <code>comp_type</code> attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code> . Otherwise, return <code>False</code> .
<code>update_net_worth(self, net_worth)</code>	The method updates the company market cap such that the number of stocks remains the same and the single stock price changes. The <code>net_worth</code> input value must be a positive integer or float. If the method succeeds in updating, then return the Boolean value <code>True</code> . Otherwise, return <code>False</code> .
<code>add_stocks(self, number)</code>	<p>The method receives an integer number and adds this number to the current number of stocks the company holds.</p> <p>Note: This number can be negative as well. In this case, you are required to reduce the number of stocks.</p> <p>Note: The updated number of stocks must be positive. You can assume that number is an integer. If the method succeeds in updating, then return the Boolean value <code>True</code>. Otherwise, return <code>False</code>.</p>
<code>__repr__(self)</code> <code>__str__(self)</code>	<p>The methods return a string that contains the description of the company. For example, for the company "NVIDIA Corporation" of type "High tech" that holds 1000 stocks of price 20.284, the returned string would be:</p> <p>"(NVIDIA Corporation 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)"</p> <p>Note: This is a one-line string without "\n" at its end. After each comma, there is one space.</p>



<code>change_comparison_type(cls, comparison_type)</code>	A class method that updates the criterion on which the comparison operators are based. The new <code>comparison_type</code> must follow the <code>_comparison_type</code> class attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code> . Otherwise, return <code>False</code> .
<code>__lt__(self, other)</code> <code>__gt__(self, other)</code> <code>__eq__(self, other)</code> <code>__ne__(self, other)</code> <code>__ge__(self, other)</code> <code>__le__(self, other)</code>	<p>You are required to perform operator overloading for all the comparison operators (i.e., <code>></code>, <code><</code>, <code>==</code>, <code>!=</code>, <code>>=</code>, <code><=</code>) such that comparison is based on the criterion defined by the class attribute <code>_comparison_type</code>.</p> <p>Note: if “other” is not of type <i>Company</i>, return the Boolean value <code>False</code>.</p>
<code>__add__(self, other)</code>	<p>You are required to perform operator overloading to the addition operator (i.e., <code>+</code>) such that the addition operator will merge the companies on which the operation is performed. The right operand will be merged to the left operand (i.e., the right-side company will be merged to the left one). This method will return a <u>new</u> instance of the merged company. The merged company has:</p> <ul style="list-style-type: none"> • The same name and type as the left operand. • The number of stocks is the sum of the number of stocks of the two companies. • The market cap is the sum of the market cap of the two companies. The single stock price is determined accordingly. <p>Note: You are not allowed to change existing companies.</p> <p>Note: You can assume that the “other” argument type is <i>Company</i>.</p>

Note: You can implement any function or method (including static and class methods) that might help you.

The CompanyNode Class

The **CompanyNode** Class represents a single company that might have relations with other companies (i.e., it can be a parent company with multiple subsidiary companies or a subsidiary company by itself).

- The **CompanyNode** Class inherits from the **Company** Class.
- We will interchangeably use the terms “node” and “**CompanyNode**”.
- We will interchangeably use the terms “children” and “subsidiary companies”.
- A node connected to all lower-level nodes is called an “ancestor”. The connected lower-level nodes are “descendants” of the ancestor node.

- The "**Company Node rule**", that each CompanyNode must obey: a CompanyNode must be larger or equal to its subsidiary companies (i.e., its children) where the comparison is based on one of the four criteria: "net value", "stock num", "stock price", and "total sum". The first three have the exact definition as in the Company Class. The "total sum" is the sum of the company's market cap and all the companies lower to it in the hierarchy (i.e., its subsidiary companies, their subsidiary companies, and so on – or in other words, its descendants).
- Note that since this rule should be valid for each node, each parent node must be higher or equal to its children.
- **Note:** You can assume that the comparison criterion between CompanyNode instances is predetermined before creating any CompanyNode instance and will not change after such ones are created.

Those are the following attributes that you are required to define in the class (in addition to ones the class inherits from the Company Class):

Instance attributes:

__children – of type *List* and represents a list of all the children of the company. The type of elements in the list must be of CompanyNode. Note that this is a private attribute.

__parent – of type *CompanyNode* or *None* and represents the mother company of the current CompanyNode Class. A nonvalue represents a CompanyNode without a mother company. Note that this is a private attribute.

Class attributes:

_comparison_type – of type *string* and represents the criterion on which the compression operators of the class are based. The 4 criteria were defined above. The default criterion is the criterion of the parent class Company. Note that this attribute overrides the parent attribute.

You are required to implement the following methods in CompanyNode Class:

Note: Think recursive, it will ease your work!

Method	Description
<code>__init__(self, name, stocks_num, stock_price, comp_type)</code>	Initialize the Company instance attributes the same way as the Company Class initializer. In addition, the <code>__children</code> and <code>__parent</code> instance attributes are initialized to an empty list and <i>None</i> , respectively.
<code>get_parent(self)</code>	The method returns the <code>__parent</code> instance attribute.
<code>get_children(self)</code>	The method returns the <code>__children</code> instance attribute.
<code>def __len__(self)</code>	The method returns the number of children the CompanyNode instance has.

<code>is_leaf(self)</code>	The method returns the Boolean value True if the <i>CompanyNode</i> instance does not have children. Otherwise, return False.
<code>add_child(self, child)</code>	<p>The method adds a child to the <i>CompanyNode</i> instance. The input argument <i>child</i> must be of type <i>CompanyNode</i>. The child is added to the end of the list of children (i.e., the <code>__children</code> list).</p> <ul style="list-style-type: none"> You can add a child only if the “Company Node rule” is satisfied. You can assume that before the addition, the child has no relation to the current <i>CompanyNode</i>, and that it satisfied the “Company Node rule”. If the method succeeds in adding the child, return the Boolean value True. Otherwise, return False. Note that you are required to set the <code>__parent</code> attribute of the child accordingly.
<code>total_net_worth(self)</code>	The method returns the sum of the <i>CompanyNode</i> market cap and its descendants’ market caps. For example, if we apply this method to the Alphabet company (Figure 1), the method will return the sum of the market cap of all the companies in the Alphabet tree.
<code>test_node_order_validity(self)</code>	<p>The method checks whether the current node (“self”) satisfies the “Company Node rule” with respect to his ancestors and descendants. The method returns the Boolean value True if the conditions are met and False otherwise.</p> <p>Examples (for the company tree in Figure 1):</p> <ol style="list-style-type: none"> If we apply the method to the “Google” node, then we need to check that “Google” is larger or equal to “YouTube”, “Search”, and “Maps”. In addition, we need to check that “Google” is lower or equal to “Alphabet”. If we apply the method to the “YouTube” node, we need to check that “YouTube” is lower or equal to “Google” and that “Google” is lower or equal to “Alphabet”.
<code>is_ancestor(self, other)</code>	The method checks whether the current node (“self”) is an ancestor to “other”. You can assume that input argument “other” is of type <i>CompanyNode</i> .

<code>change_comparison_type(cls, comparison_type)</code>	<p>A class method that updates the criterion on which the comparison operators are based. The new <code>comparison_type</code> must follow the <code>_comparison_type</code> class attribute constraints. If the method succeeds in updating, then return the Boolean value <code>True</code>. Otherwise, return <code>False</code>.</p> <p>Note: <code>Comparison_type</code> now has 4 options.</p> <p>Note: You can assume that the comparison criterion between <code>CompanyNode</code> instances is predetermined before creating any <code>CompanyNode</code> instance and will not change after such ones are created.</p> <p>Note: Note that this method overrides the parent method.</p>
<code>__repr__(self):</code>	<p>This method overrides the parent method. The method returns a string that contains the description of the company and its descendants. The format of the string for a <code>CompanyNode</code> instance with 3 children is:</p> <p><code>"[node_string, [child_1_string, child_2_string, child_3_string]]"</code></p> <p><code>node_string</code> is the string returned from the <code>repr</code> method of the parent class <code>Company</code>, and each <code>child_i_string</code> is the string returned by the <code>repr</code> method of the <code>CompanyNode</code>. Note that the definition of the format is recursive.</p> <p>Note: There is one space after each comma in the format above.</p> <p>Note: This method overrides the parent method.</p>
<code>__add__(self, other)</code>	<p>This method overrides the addition operator of the parent class <code>Company</code>. This method returns a <u>new</u> <code>CompanyNode</code> instance of the merged company. Since we are merging two <code>CompanyNode</code> instances, then we modify the operation:</p> <ul style="list-style-type: none"> • The name, type, single stock price, and number of stocks values of the merged <code>CompanyNode</code> will be the same as defined in the parent class operator. • If “other” is an ancestor of self, raise <code>ValueError</code> (you are free to choose any error message).

	<ul style="list-style-type: none"> • The descendants of “other” are merged into “self”. Practically, this means that the children of “other” are appended to the “self” children list (to the end, in the same order they were ordered in “other”). • You can assume that the trees that “other” and “self” are part of were obeying the “Company Node rule”. • If the new merged companyNode instance is not obeying the “Company Node rule”, raise ValueError (you are free to choose any error message). • The objects in any relation to merged CompanyNode should be deeply copied. By any relation, the meaning is that if we build the tree that the merged CompanyNode is part of, then each node in the tree should be deeply copied. • Note: “other” and “self” can be in the same tree. • Note: You can assume that all the company names are unique. • Note: You are not allowed to change existing companies. • Note: Pay careful attention to detaching and attaching nodes correctly. • Note: You can assume that the other argument type is <i>CompanyNode</i>.
<pre>set_stock_price(self, stock_price) set_stocks_num(self, stocks_num) update_net_worth(self, net_worth) add_stocks(self, number)</pre>	<p>Override (if necessary) the methods such that their functionality remains the same, but the “Company Node rule” is considered. If the rule is not preserved, then return the Boolean value False, and do not update the company.</p>
<pre>__lt__(self, other) __gt__(self, other) __eq__(self, other) __ne__(self, other) __ge__(self, other)</pre>	<p>Override (if necessary) the methods such that they consider the fourth criterion added.</p>

<code>__le__(self, other)</code>	
----------------------------------	--

Note: You can implement any function or method (including static and class methods) that might help you. In some cases, you might even have to do so.

Note: recall that can call the parent class methods.

The CompanyTree Class

The CompanyTree Class represents a Tree of companies that has a relations. Those are the following attributes that you are required to define in the class:

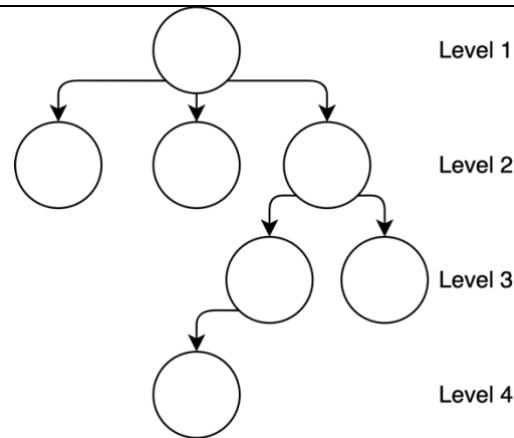
Instance attributes:

__root – of type *CompanyNode* or *None* and represents the root of the company tree (i.e., the holder company). Note that this is a private attribute.

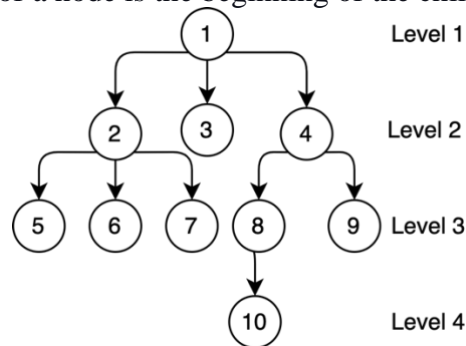
You are required to implement the following methods in CompanyTree Class:

Note: Think recursive, it will ease your work!

Method	Description
<code>__init__(self, root=None)</code>	The initializer receives the root of the company tree. The method initializes the new CompanyTree with the root value. If the root is None, then the CompanyTree is an empty tree. Otherwise, the root must be of type <i>CompanyNode</i> . In addition, it must be a parent company that does not have any holder. If the root does not follow the attribute constraints, the program should raise ValueError (you are free to choose any error message).
<code>set_root(self, root)</code>	The method updates the root of the company tree (i.e., <code>__root</code>) if the new root follows the attribute constraints we defined in the initializer. If the method succeeds in updating, then return the Boolean value True. Otherwise, return False. Note: root can be None.
<code>get_root(self)</code>	The method returns the <code>__root</code> instance attribute.
<code>__str__(self)</code>	The method returns a string that contains the description of the company Tree. The returned string will contain each level of the tree elements in different rows (in ascending order). In a tree, each step from top to bottom is called a tree's level. The level count starts with 1 (the root) and increases by 1 at each level or step. For example, this is a 4-level tree:



The inner order in each level is determined by the children's order of the prior level nodes. For example, in this 4-level tree (assuming the most left child of a node is the beginning of the children list):



The 1-level order: node 1.

The 2-level order: node 2, node 3, node 4.

The 3-level order: node 5, node 6, node 7, node 8, node 9.

The 4-level order is: node 10.

The strings of the nodes in the same level are separated by “ * ” (one space, an asterisk, and one space).

Example:

The returned string of tree above will be:

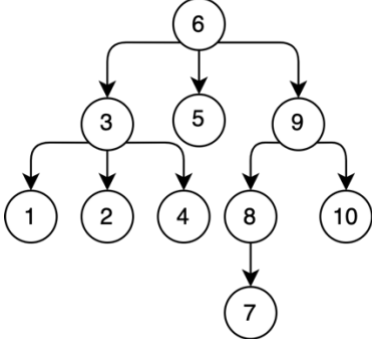
```

str(node 1) + "\n" + str(node 2) + " * " + str(node 3) +
" * " + str(node 4) + "\n" + str(node 5) + " * " +
str(node 6) + " * " + str(node 7) + " * " + str(node 8) +
" * " + str(node 9) + "\n" + str(node 10)
  
```

Note: The string representation of each CompanyNode is the str method and not the repr method.

`__iter__(self)`
`__next__(self)`

You are required to implement the iterator design pattern you saw in class. Using your implementation, you should be able to traverse the tree elements in order. You can read about the in-order traversal in a

	<p>binary tree here. Since our tree is a nonbinary tree, we will define the in-order as follows:</p> <ol style="list-style-type: none"> 1. Recursively traverse the current node's left subtrees by their order. 2. Visit the current node. 3. Recursively traverse the current node's right subtrees by their order. <p>Where:</p> <ul style="list-style-type: none"> • The <u>current node's left subtrees</u> are defined as the first half of the current node's children (if there are an odd number of children, then the middle child is considered in the first half). • The <u>current node's right subtrees</u> are defined as the rest of the current node's children. <p>Example: The node numbers describe the in-order of the following tree:</p>  <pre> graph TD 6((6)) --> 3((3)) 6 --> 5((5)) 6 --> 9((9)) 3 --> 1((1)) 3 --> 2((2)) 3 --> 4((4)) 9 --> 8((8)) 9 --> 10((10)) 8 --> 7((7)) </pre> <p>Note: You can add instance attributes dynamically (i.e., not in the initializer).</p>
<p><code>insert_node(self, node)</code></p>	<p>This method inserts a new node into the CompanyTree. Before the insertion, the new node must be:</p> <ol style="list-style-type: none"> 1. Of type <i>CompanyNode</i>. 2. A node without a parent node. 3. A node that obeys the “Company Node rule”. 4. A node that does not exist in the CompanyTree. <p>If all the preconditions are satisfied, the method will insert the new node as a child of the first node in the in-order traversal so that the insertion will not affect the “Company Node rule”. The new node will be appended to the end of the children list.</p> <p>If the method succeeds in updating, then return the Boolean value True. Otherwise, return False.</p> <p>Note: You can assume that all the company names are unique. Note: The new node might have children.</p>

<code>remove_node(self, name)</code>	<p>The method removes the CompanyNode named name from the tree and returns it. If no node has this name, the method returns None. You can assume that the method does not receive the name of the root (i.e., the method will not remove the root).</p> <p><u>If the node to remove is not the root and has children</u>, append its children to the parent of the removed node (append to the end of the children list). You are required <u>to empty</u> the children list of the removed node.</p> <p>Note: You can assume that all the company names are unique. Note: Pay careful attention to detaching and attaching nodes correctly. Note: The returned node must be the removed node without any of its old relations (i.e., the parent attribute needs to be None, and the children list needs to be empty).</p>
--------------------------------------	---

Running examples:

We provide you a Main file that contains some running examples. The code is provided here as well:

```
from Company import Company
from CompanyNode import CompanyNode
from CompantTree import CompanyTree

def main():
    """
    Note: This file contains only partial tests
    """
    print("#####Company tests#####")
    print("1. create Company instance:")
    c1 = Company("Google", 1000, 20.284, "High Tech")
    print(c1)

    print("2. test net_worth method:")
    print(c1.net_worth())

    print("3. test set_name method:")
    print(c1.set_name("Google2"))
    print(c1.set_name("Google Two"))
    print(c1)

    print("4. test set_stocks_num method:")
    print(c1.set_stocks_num(2000))
    print(c1)
```

```
print("5. test set_stock_price method:")
print(c1.set_stock_price(25))
print(c1)

print("6. test set_comp_type method:")
print(c1.set_comp_type("General"))
print(c1)

print("7. test update_net_worth method:")
print(c1.update_net_worth(0))
print(c1)
print(c1.update_net_worth(2027.5))
print(c1)

print("8. test add_stocks method:")
print(c1.add_stocks(-850))
print(c1)
print(c1.add_stocks(8000))
print(c1)

print("9. test Operator Overloading:")
c2 = Company("Lenovo", 1000, 5, "High Tech")
print(c2)
print(c1 > c2)
print(c1 < c2)
print(c1 == c2)
print(c1 + c2)

print("#####CompanyNode tests#####")
print("1. create CompanyNode instances and connect them, and test repr:")
c0_node = CompanyNode("BGU Zero", 1000, 20.284, "High Tech")
c1_node = CompanyNode("BGU One", 1100, 20.284, "High Tech")
c2_node = CompanyNode("BGU Two", 2000, 20.284, "High Tech")
print(c2_node.add_child(c1_node))
print(c1_node.add_child(c0_node))
print(repr(c2_node))

print("2. test len:")
print(len(c2_node))

print("3. test total_net_worth:")
print(c2_node.total_net_worth())

print("4. test is_ancestor:")
print(c2_node.is_ancestor(c1_node))
print(c1_node.is_ancestor(c2_node))

print("5. add operator:")
c3_node = c2_node + c0_node
print(repr(c3_node))

print("#####CompanyTree tests#####")
c0_node = CompanyNode("BGU Zero", 1000, 20.284, "High Tech")
```

```
c1_node = CompanyNode("BGU One", 1100, 20.284, "High Tech")
c11_node = CompanyNode("BGU One One", 1100, 20.284, "High Tech")
c12_node = CompanyNode("BGU One Two", 1100, 20.284, "High Tech")
c13_node = CompanyNode("BGU One Three", 1100, 20.284, "High Tech")
c2_node = CompanyNode("BGU Two", 2000, 20.284, "High Tech")
c2_node.add_child(c1_node)
c2_node.add_child(c11_node)
c2_node.add_child(c12_node)
c2_node.add_child(c13_node)
c1_node.add_child(c0_node)
comp_tree = CompanyTree(root=c2_node)
print("1. test str:")
print(comp_tree)
print("2. test iterator design pattern implementation:")
for node in comp_tree:
    print(node)

print("3. test insert_node method:")
c_new_node = CompanyNode("BGU added", 500, 20.284, "High Tech")
print(comp_tree.insert_node(c_new_node))
print(comp_tree)

print("4. test remove_node method:")
print(repr(comp_tree.remove_node("BGU Zero")))
print(comp_tree)

if __name__ == "__main__":
    main()
```

Executing the code will output:

(Some lines are split into multiple lines due to the PDF width limit. We provided the output as a spread file as well)

#####Company tests#####

1. create Company instance:
(Google, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)
2. test net_worth method:
20284.0
3. test set_name method:
False
True
(Google Two, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)
4. test set_stocks_num method:
True
(Google Two, 2000 stocks, Price: 10.142, High Tech, Net Worth: 20284.0)
5. test set_stock_price method:
True
(Google Two, 811 stocks, Price: 25, High Tech, Net Worth: 20275)
6. test set_comp_type method:
True
(Google Two, 811 stocks, Price: 25, General, Net Worth: 20275)
7. test update_net_worth method:



False
(Google Two, 811 stocks, Price: 25, General, Net Worth: 20275)
True
(Google Two, 811 stocks, Price: 2.5, General, Net Worth: 2027.5)
8. test add_stocks method:
False
(Google Two, 811 stocks, Price: 2.5, General, Net Worth: 2027.5)
True
(Google Two, 8811 stocks, Price: 2.5, General, Net Worth: 22027.5)
9. test Operator Overloading:
(Lenovo, 1000 stocks, Price: 5, High Tech, Net Worth: 5000)
True
False
False
(Google Two, 9811 stocks, Price: 2.7548160228315157, General, Net Worth: 27027.5)
#####CompanyNode tests#####
1. create CompanyNode instances and connect them, and test repr:
True
True
[(BGU Two, 2000 stocks, Price: 20.284, High Tech, Net Worth: 40568.0), [(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998), [(BGU Zero, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0), []]]]]
2. test len:
1
3. test total_net_worth:
83164.4
4. test is_ancestor:
True
False
5. add operator:
[(BGU Two, 3000 stocks, Price: 20.284, High Tech, Net Worth: 60852.0), [(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998), []]]
#####CompanyTree tests#####
1. test str:
(BGU Two, 2000 stocks, Price: 20.284, High Tech, Net Worth: 40568.0)
(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One Two, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One Three, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)
(BGU Zero, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)
2. test iterator design pattern implementation:
(BGU Zero, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)
(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)
(BGU One One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)
(BGU Two, 2000 stocks, Price: 20.284, High Tech, Net Worth: 40568.0)
(BGU One Two, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)
(BGU One Three, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)
3. test insert_node method:
True
(BGU Two, 2000 stocks, Price: 20.284, High Tech, Net Worth: 40568.0)
(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One Two, 1100 stocks, Price: 20.284,



High Tech, Net Worth: 22312.399999999998) * (BGU One Three, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)

(BGU Zero, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0)

(BGU added, 500 stocks, Price: 20.284, High Tech, Net Worth: 10142.0)

4. test remove_node method:

[(BGU Zero, 1000 stocks, Price: 20.284, High Tech, Net Worth: 20284.0), []]

(BGU Two, 2000 stocks, Price: 20.284, High Tech, Net Worth: 40568.0)

(BGU One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One One, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One Two, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998) * (BGU One Three, 1100 stocks, Price: 20.284, High Tech, Net Worth: 22312.399999999998)

(BGU added, 500 stocks, Price: 20.284, High Tech, Net Worth: 10142.0)