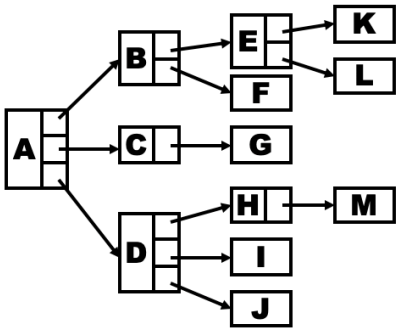
Lab 9 Report Template

1.Given a tree, say in linked list, construct another tree using FirstChild-NextSibling representation.

1.1 linked list tree:

The linked list tree is organized in this way:



1.1.1 coding in Python:

```python
class linked_node(object):

    def __init__(self, value=None):
        self.value = value
        self.next = []


class linked_list_tree(object):

    def __init__(self, value=None):
        self.root = linked_node(value)

    def add_as_child(self, father, *child):
        current_level = [self.root]
        next_level = []
        while current_level != []:
            for node in current_level:
                if node.value == father:
                    node.next += list(map(linked_node, child))
                    return
                else:
                    next_level += node.next
            current_level = next_level
            next_level = []
        print("do not find father")

    def show(self, node=None):
        if not node:
            node = self.root
        print(node.value,end=" ")
        nodes = node.next
        for no in nodes:
            self.show(no)
```

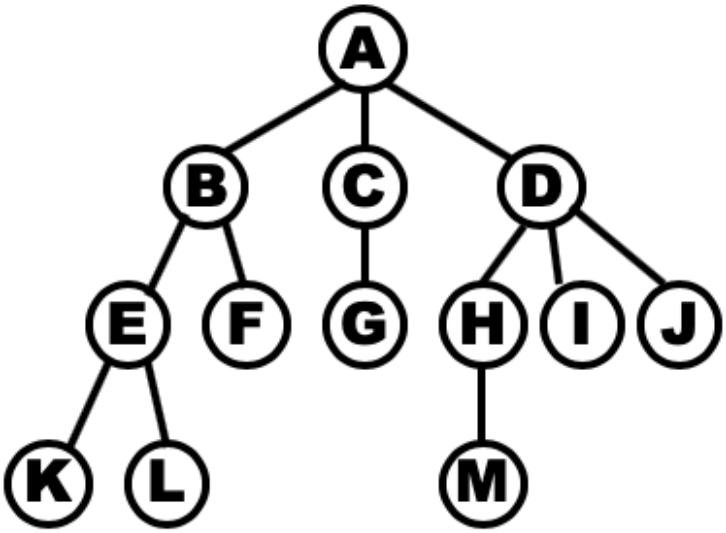
1.1.2 details:

Every node in the tree contain a value and a list called next, which contain the next nodes of this node. When I try to add some value into the tree, I find a special value and generate a node as one of the next nodes of this node. Use the method show, to get the preorder traversal of this tree.

1.1.3 test:

```
Tree = linked_list_tree("A")
Tree.add_as_child("A", "B", "C", "D")
Tree.add_as_child("B", "E", "F")
Tree.add_as_child("E", "K", "L")
Tree.add_as_child("C", "G")
Tree.add_as_child("D", "H", "I", "J")
Tree.add_as_child("H", "M")
Tree.show()
```

1.1.4 The tree:

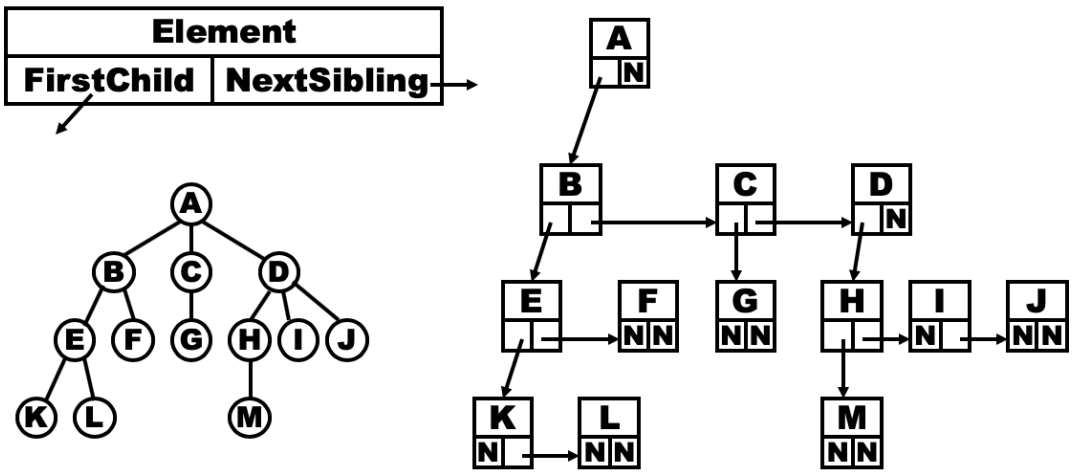


1.1.5 Output:

```
A B E K L F C G D H M I J
```

In fact, this is the preorder traversal of the tree.

1.2 FirstChild-NextSibling tree:



1.2.1: code in Python:

```python
class FCNS_node(object):

    def __init__(self, value, first_child=None, next_sibling=None):
        self.value = value
        self.first_child = first_child
        self.next_sibling = next_sibling
```

```python
class FCNS_tree(object):

    def __init__(self):
        self.root = None

    def set_root(self, value, first_child=None, next_sibling=None):
        self.root = FCNS_node(value, first_child, next_sibling)

    def get_root(self):
        return self.root
```

```python
def add_as_first_child(self, node, value):
    current_level = [self.root]
    next_level = []
    while current_level:

        for current_node in current_level:
            if current_node.value == node:
                if not current_node.first_child:
                    current_node.first_child = FCNS_node(value)
                    return
                else:
                    print("this men has the first child")
            elif current_node.first_child:
                    point = current_node.first_child
                    while point.next_sibling:
                        next_level.append(point)
                        point = point.next_sibling
                    next_level.append(point)

        current_level = next_level
        next_level = []
    print("Do not find the node")
```

```
def add_as_next_sibling(self, node, value):
    current_level = [self.root]
    next_level = []
    while current_level:

        for current_node in current_level:
            if current_node.value == node:
                if not current_node.next_sibling:
                    current_node.next_sibling = FCNS_node(value)
                    return
                else:
                    print("this men has the next sibling")
            elif current_node.first_child:
                point = current_node.first_child
                while point.next_sibling:
                    next_level.append(point)
                    point = point.next_sibling
                next_level.append(point)

        current_level = next_level
        next_level = []
```

```
def show(self):
    current_level = [self.root]
    next_level = []
    while current_level:
        print()

        for current_node in current_level:
            print(current_node.value, end=" ")
            if current_node.first_child:
                point = current_node.first_child
                while point.next_sibling:
                    next_level.append(point)
                    point = point.next_sibling
                next_level.append(point)

        current_level = next_level
        next_level = []
    print()
```

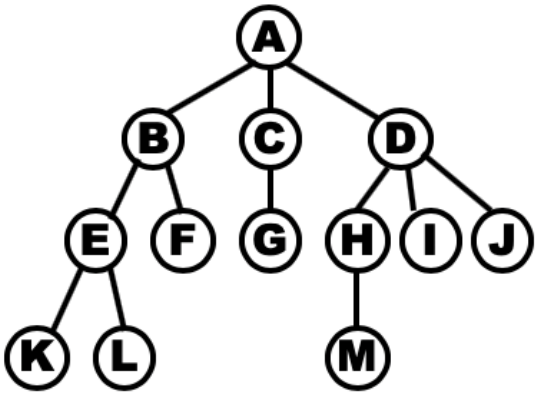
1.2.2

Every node in this tree has a first child and a next sibling, which all can be none. Every tree has a specific root node. There are two methods can be used to build a tree.

1.2.3 Test:

```
Tree = FCNS_tree()
Tree.set_root("A")
Tree.add_as_first_child("A", "B")
Tree.add_as_next_sibling("B", "C")
Tree.add_as_next_sibling("C", "D")
Tree.add_as_first_child("B", "E")
Tree.add_as_next_sibling("E", "F")
Tree.add_as_first_child("E", "K")
Tree.add_as_next_sibling("K", "L")
Tree.add_as_first_child("C", "G")
Tree.add_as_first_child("D", "H")
Tree.add_as_next_sibling("H", "I")
Tree.add_as_next_sibling("I", "J")
Tree.add_as_first_child("H", "M")
Tree.show()
```

1.2.4 the tree:



1.2.4 Output:

```
A
B C D
E F G H I J
K L M
```

2.Implement inorder, preoder, postorder and levelorder tree traversal algorithms.

2.1: background:

I used the FirstChild-NextSibling tree to finish the inorder, preoder, postorder and levelorder tree traversal algorithms.

2.2: inorder:

```
void inorder ( tree_ptr tree )
{ if ( tree ) {
      inorder ( tree->Left );
      visit ( tree->Element );
      inorder ( tree->Right );
  }
}
```

2.2.1: coding in Python:

```python
def __inorder(self, node):
    son = node.first_child
    if son:
        self.__inorder(son)
        output_in.append(node.value)
        while son.next_sibling:
            son = son.next_sibling
            self.__inorder(son)
    else:
        output_in.append(node.value)

def inoder_traversal(self):
    global output_in
    output_in = []
    self.__inorder(self.root)
    return output_in
```

2.3: preorder:

```
void  preorder ( tree_ptr  tree )
{  if  ( tree )  {
       visit ( tree );
       for (each child C of tree )
          preorder ( C );
   }
}
```

2.3.1 coding in Python:

```python
def __preorder(self, node):
    output_pre.append(node.value)
    son = node.first_child
    if son:
        while son.next_sibling:
            self.__preorder(son)
            son = son.next_sibling
        self.__preorder(son)

def preorder_traversal(self):
    global output_pre
    output_pre = []
    self.__preorder(self.root)
    return output_pre
```

2.4: postorder:

```
void  postorder ( tree_ptr  tree )
{  if  ( tree )  {
       for (each child C of tree )
          postorder ( C );
       visit ( tree );
   }
}
```

2.4.1 coding in Python:

```python
def __postorder(self, node):
    son = node.first_child
    if son:
        while son.next_sibling:
            self.__postorder(son)
            son = son.next_sibling
        self.__postorder(son)
    output_post.append(node.value)

def postorder_traversal(self):
    global output_post
    output_post = []
    self.__postorder(self.root)
    return output_post
```

2.5: levelorder:

```
void  levelorder ( tree_ptr  tree )
{   enqueue ( tree );
    while (queue is not empty) {
       visit ( T = dequeue ( ) );
       for (each child C of T )
          enqueue ( C );
    }
}
```

2.5.1 coding in Python:

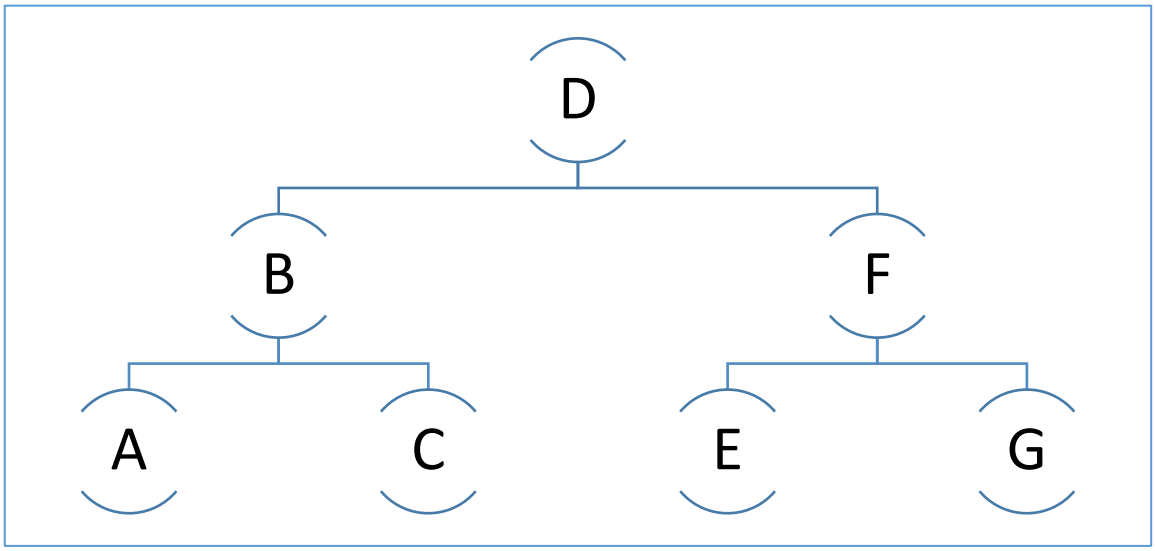
```python
def levelorder_traversal(self):
    global output_level
    output_level = []
    self.__levelorder([self.root])
    return output_level

def __levelorder(self, nodes):
    if nodes == []:
        return
    sons = []
    for node in nodes:
        output_level.append(node.value)
        son = node.first_child
        while son:
            sons.append(son)
            son = son.next_sibling
    self.__levelorder(sons)
```

2.6: Test:

```
tree = FCNS_tree()
tree.set_root("D")
tree.add_as_first_child("D", "B")
tree.add_as_next_sibling("B", "F")
tree.add_as_first_child("B", "A")
tree.add_as_first_child("F", "E")
tree.add_as_next_sibling("A", "C")
tree.add_as_next_sibling("E", "G")
tree.show()
print("levelorder: "+str(tree.levelorder_traversal()))
print("preorder"+str(tree.preorder_traversal()))
print("inorder"+str(tree.inoder_traversal()))
print("postorder"+str(tree.postorder_traversal()))
```

2.7: The tree:



2.8:Output:

```
D
B F
A C E G
levelorder: ['D', 'B', 'F', 'A', 'C', 'E', 'G']
preorder['D', 'B', 'A', 'C', 'F', 'E', 'G']
inorder['A', 'B', 'C', 'D', 'E', 'F', 'G']
postorder['A', 'C', 'B', 'E', 'G', 'F', 'D']
```