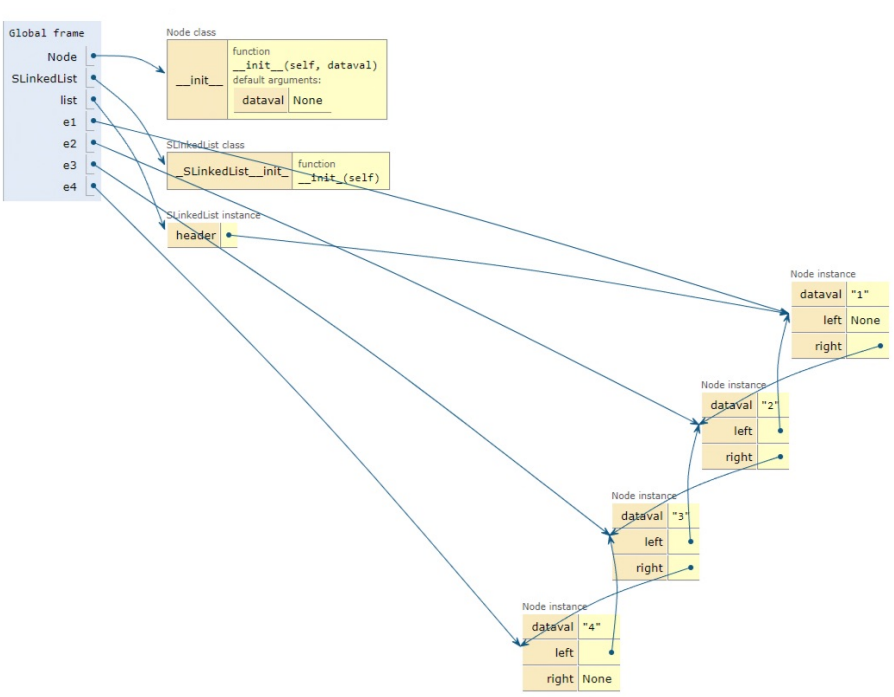
(known limitations)

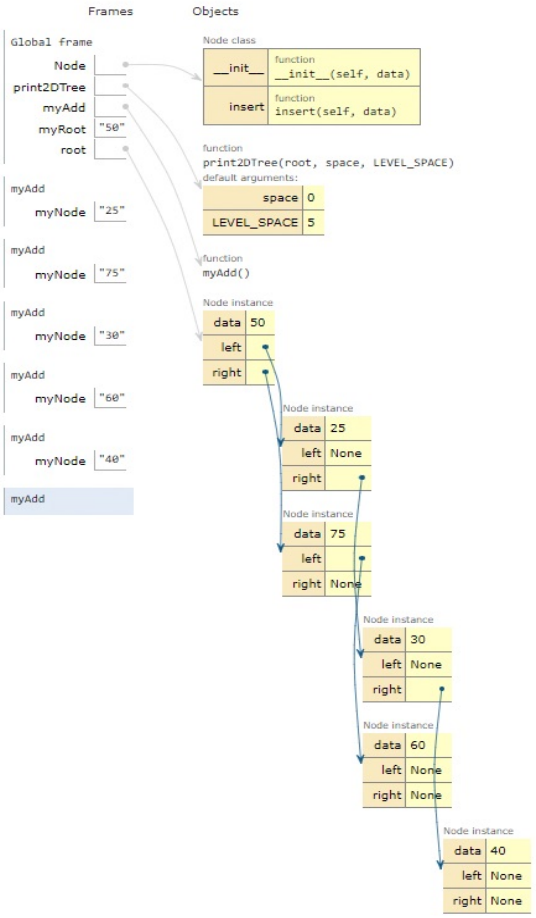
```
1  class Node:
2      def __init__(self, dataval=None):
3          self.dataval = dataval
4          self.left = None
5          self.right = None
6  class SLinkedList:
7      def __init_(self):
8          self.header = None
9  list = SLinkedList()
10 e1 = Node("1")
11 e2 = Node("2")
12 e3 = Node("3")
13 e4 = Node("4")
14 list.header = e1
15 e1.right = e2
16 e2.left = e1
17 e2.right = e3
18 e3.left = e2
19 e3.right = e4
20 e4.left = e3
```




```
class Node:
    def __init__(self,data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self,data):
        if self.data is None:
            self.data = data
        else:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                    print2DTree(root)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                    print2DTree(root)
                else:
                    self.right.insert(data)

def print2DTree(root, space=0, LEVEL_SPACE=5):
    if (root == None):
        return
    space += LEVEL_SPACE
    print2DTree(root.right, space)
    # print() # neighbor space
    for i in range(LEVEL_SPACE, space):
        print(end=" ")
    print("|" + str(root.data) + "|<")
    print2DTree(root.left, space)

def myAdd():
    while (1):
        myNode = input('Add Node:')
        if myNode.isdigit():
            root.insert(int(myNode))
            myAdd()
        else:
            print('Only INT!')

while (1):
    myRoot = input('Add Root:')
    if myRoot.isdigit():
        root = Node(int(myRoot))
        print2DTree(root)
        myAdd()
    else:
        print('Only INT!')
```

Print output (drag lower right corner to resize)

```
Add Node:40
      |75|<
            |60|<
|50|<
            |40|<
      |30|<
|25|<
```



```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
        if self.data is None:
            self.data = data
        else:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                    print2DTree(root)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                    print2DTree(root)
                else:
                    self.right.insert(data)


def lastNodeDel(root, myLastNode):
    myQueue = []
    myQueue.append(root)
    while (len(myQueue)):
        temp = myQueue.pop(0)
        if temp is myLastNode:
            temp = None
            return
        if temp.right:
            if temp.right is myLastNode:
                temp.right = None
                return
            else:
                myQueue.append(temp.right)
        if temp.left:
            if temp.left is myLastNode:
                temp.left = None
                return
            else:
                myQueue.append(temp.left)


def delNode(root, delTarget):
    if root == None:
        return None
    if root.left is None and root.right is None:
        if root.data == delTarget:
            temp = root.right
            root = None
            return temp
    delTarget_node = None
    myQueue = []
    myQueue.append(root)
    temp = None
    while (len(myQueue)):
        temp = myQueue.pop(0)
        if temp.data == delTarget:
            delTarget_node = temp
        if temp.left:
            myQueue.append(temp.left)
        if temp.right:
            myQueue.append(temp.right)
    if delTarget_node:
        lastNodeVal = temp.data
        lastNodeDel(root, temp)
        delTarget_node.data = lastNodeVal
    return root
```

```
def print2DTree(root, space=0,
LEVEL_SPACE=5):
    if (root == None):
        return
    space += LEVEL_SPACE
    print2DTree(root.right, space)
    # print() # neighbor space
    for i in range(LEVEL_SPACE, space):
        print(end=" ")
    print("|" + str(root.data) + "|<")
    print2DTree(root.left, space)


def myDel(root):
    while (1):
        delTarget = int(input('Del Node: '))
        root = delNode(root, delTarget)
        print2DTree(root)


def myAdd():
    while (1):
        myNode = input('Add Node:')
        if myNode.isdigit():
            root.insert(int(myNode))
            myAdd()
        else:
            myDel(root)


while (1):
    myRoot = input('Add Root:')
    if myRoot.isdigit():
        root = Node(int(myRoot))
        print2DTree(root)
        myAdd()
    else:
        print('Only INT!')
```

```
Del Node: 30
      |75|<
            |60|<
|50|<
            |40|<
      |25|<
Del Node: 75
      |60|<
|50|<
            |40|<
      |25|<
Del Node: 40
      |60|<
|50|<
      |25|<
Del Node:
```

```
import collections

class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
        if self.data is None:
            self.data = data
        else:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                    print2DTree(root)
                    print("Height of tree is: " + str(height(root)))
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                    print2DTree(root)
                    print("Height of tree is: " + str(height(root)))
                else:
                    self.right.insert(data)

def child(obj, data):
    if obj:
        if obj.data == data:
            print('Child of node '+' '+str(data)+' is '+str(obj.left.data) +
                  ' and '+str(obj.right.data))
            return
        else:
            return child(obj.left, data) or child(obj.right, data)  # เชควน


def parent(obj, data):
    if obj:
        if (obj.left and obj.left.data == data) or (obj.right and obj.right.data == data):
            return obj.data or parent(obj.left, data) or parent(obj.right, data)
        else:
            return parent(obj.left, data) or parent(obj.right, data)


def sibling(obj, data):
    if obj:
        if (obj.left and obj.left.data == data):
            return obj.right.data
        elif (obj.right and obj.right.data == data):
            return obj.left.data
        else:
            return sibling(obj.left, data) or sibling(obj.right, data)


def PrintLeaf(root):

    s1 = []
    s2 = []

    s1.append(root)
    while len(s1) != 0:
        curr = s1.pop()
        if curr.left:
            s1.append(curr.left)
        if curr.right:
            s1.append(curr.right)
        elif not curr.left and not curr.right:
            s2.append(curr)
    print('All leaf is')
    for i in range(len(s2)):
        print(s2.pop().data)
    return

def height(root):
    ans = 0
    queue = collections.deque()
    if root is None:
        return ans

    queue.append(root)

    while queue:
        currSize = len(queue)
        while currSize > 0:
            currNode = queue.popleft()
            currSize -= 1

            if currNode.left is not None:
                queue.append(currNode.left)
            if currNode.right is not None:
                queue.append(currNode.right)

        ans += 1
    return ans-1
```

```
def lastNodeDel(root, myLastNode):
    myQueue = []
    myQueue.append(root)
    while (len(myQueue)):
        temp = myQueue.pop(0)
        if temp is myLastNode:
            temp = None
            return
        if temp.right:
            if temp.right is myLastNode:
                temp.right = None
                return
            else:
                myQueue.append(temp.right)
        if temp.left:
            if temp.left is myLastNode:
                temp.left = None
                return
            else:
                myQueue.append(temp.left)


def delNode(root, delTarget):
    if root == None:
        return None
    if root.left is None and root.right is None:
        if root.data == delTarget:
            temp = root.right
            root = None
            return temp
    delTarget_node = None
    myQueue = []
    myQueue.append(root)
    temp = None
    while (len(myQueue)):
        temp = myQueue.pop(0)
        if temp.data == delTarget:
            delTarget_node = temp
        if temp.left:
            myQueue.append(temp.left)
        if temp.right:
            myQueue.append(temp.right)
    if delTarget_node:
        lastNodeVal = temp.data
        lastNodeDel(root, temp)
        delTarget_node.data = lastNodeVal
    return root


def print2DTree(root, space=0, LEVEL_SPACE=5):
    if (root == None):
        return
    space += LEVEL_SPACE
    print2DTree(root.right, space)
    # print() # neighbor space
    for i in range(LEVEL_SPACE, space):
        print(end=" ")
    print("|" + str(root.data) + "|<")
    print2DTree(root.left, space)


def myDel(root):
    while (1):
        delTarget = int(input('Del Node: '))
        root = delNode(root, delTarget)
        print2DTree(root)


def myAdd():
    while (1):
        myNode = input('Add Node:')
        if myNode.isdigit():
            root.insert(int(myNode))
            myAdd()
        elif myNode is 'P':
            pNode = int(input('Parent Node of:'))
            print("The parent of node"+' '+str(pNode) +
                  ' '+'is '+str(parent(root, pNode)))
        elif myNode is 'C':
            pNode = int(input('Child Node of:'))
            child(root, pNode)
        elif myNode is 'L':
            PrintLeaf(root)
        elif myNode is 'S':
            pNode = int(input('Sibling Node of:'))
            print('The sibling node '+str(pNode) +
                  ' is '+str(sibling(root, pNode)))
        else:
            myDel(root)


while (1):
    myRoot = input('Add Root:')
    if myRoot.isdigit():
        root = Node(int(myRoot))
        print2DTree(root)
        myAdd()
    else:
        print('Only INT!')
while (1):
    myAdd()
```

```
Add Root:10
|10|<
Add Node:20
     |20|<
|10|<
Height of tree is: 1
Add Node:30
          |30|<
     |20|<
|10|<
Height of tree is: 2
Add Node:
```

```
Add Root:10
|10|<
Add Node:20
     |20|<
|10|<
Height of tree is: 1
Add Node:30
          |30|<
     |20|<
|10|<
Height of tree is: 2
Add Node:P
Parent Node:30
The parent of node is: 20
Add Node:
```

```
Add Root:10
|10|<
Add Node:20
     |20|<
|10|<
Height of tree is: 1
Add Node:30
          |30|<
     |20|<
|10|<
Height of tree is: 2
Add Node:15
          |30|<
     |20|<
          |15|<
|10|<
Height of tree is: 2
Add Node:C
Child Node of:20
Child of node  20 is 15 and 30
Add Node:
```

```
Add Root:20
|20|<
Add Node:10
|20|<
     |10|<
Height of tree is: 1
Add Node:30
     |30|<
|20|<
     |10|<
Height of tree is: 1
Add Node:L
All leaf is
10
30
Add Node:
```

```
Add Node:5
     |20|<
|10|<
     |5|<
Height of tree is: 1
Add Node:S
Sibling Node of:5
The sibling node 5 is 20
Add Node:
```