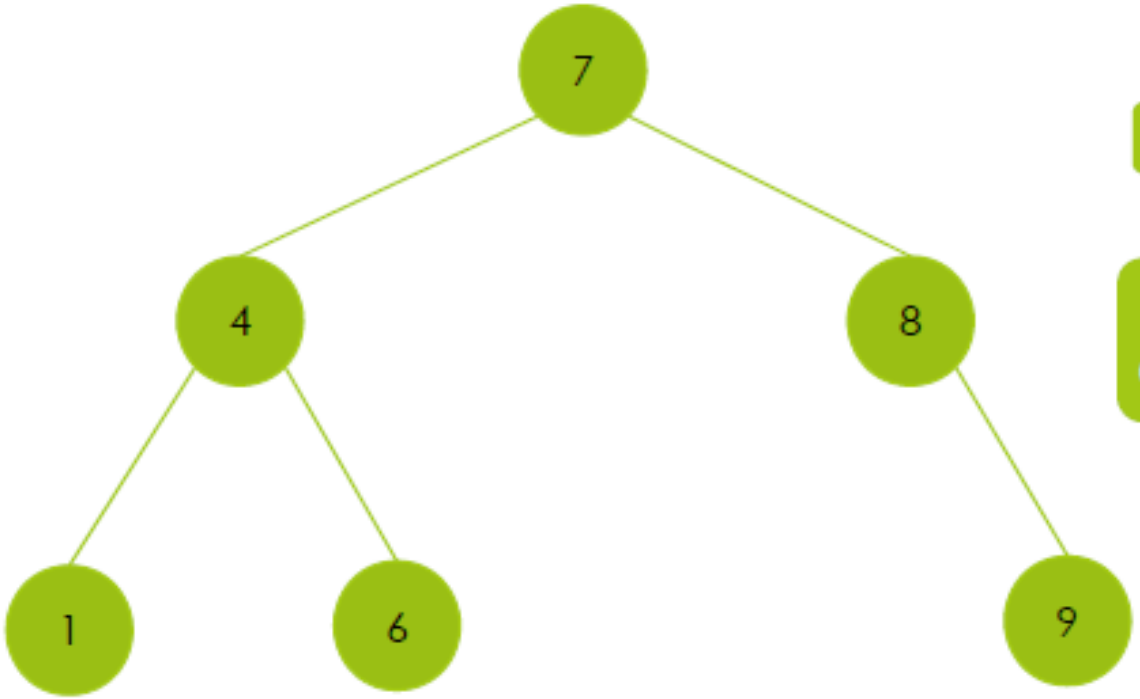
Team Project 5 (Deadline: May 25 Noon) :

- (100점) Binary Search Tree(BST)의 OOP방식으로 code를 완성하는 mission
- 본PPT에는 BST를 OOP로 설계한 Frame이 잘 표현되어 있다. 팀별로 본 PPT를 충분하게 토론하고 이해한다
- 각 page에 있는 Box를 완성하고, 마지막 page에 있는 main()을 통해서 작동하는것을 보이는것이 mission
- 팀별로 python code와 보고서PPT를 만들어서 제출한다
  - PPT는 intuitive, informative, visua하게 만들어지는것이 중요함
  - PPT에 BST의 다양한 변화모습을 잘 보이면 평가에 유리함

Binary Search Tree

BST ordering invariant: At any node with key *k*, all keys of elements in the left subtree are strictly less than *k* and all keys of elements in the rightsubtree are strictly greater than *k* (assume that there are no duplicates in the tree)



Binary tree

Satisfies the ordering invariant

BST TreeNode Class [1/3]

```
class TreeNode:

    def __init__(self, key, val, left=None,
                 right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):          # return True or False
        [redacted]

    def hasRightChild(self):         # return True or False
        [redacted]

    def isLeftChild(self):           # return True or False
        [redacted]

    def isRightChild(self):          # return True or False
        [redacted]
```

```
def isRoot(self):          # return True or False
    [redacted]

def isLeaf(self):          # return True or False
    [redacted]

def hasAnyChildren(self): # return True or False
    [redacted]

def hasBothChildren(self): # return True or False
    [redacted]

def replaceNodeData(self, key, value, lc, rc):
    [redacted]
```

BST TreeNode Class [2/3]

```
def findSuccessor(self):
    # self node의 next key value를 가진 node를 찾아서 self node가 delete 되면
    #  그자리에 넣기 위한 작업
    self가 leaf node이면 return no-successor

    self가 right child를 가지고 있으면 return right child side's minimum value

    self가 parent가 있고 left child를 가지고 있으면 return parent
```

```
def findMin(self):
        if self.hasLeftChild():
            return self.leftChild.findMin()
        else:
            return self
```

BST Tree Node [3/3]

```
def sliceOut(self):
    """ move node's child to its own position """
    if self.parent and self.hasRightChild():
        if self.isLeftChild():
            self.parent.leftChild = self.rightChild
        else:
            self.parent.rightChild = self.rightChild
    # !!! the successor node never has a left child.
```

```
def inorder_traverse(self):
    # ! in-order traverse prints out an sorted list.
    if self.hasLeftChild():
        self.leftChild.inorder_traverse()
    print(self.payload)
    if self.hasRightChild():
        self.rightChild.inorder_traverse()
```

BinarySearchTree Class [1/7]

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0


    def length(self):
        [redacted]

    def __len__(self):
        [redacted]

    def __iter__(self):
        [redacted]
```

BinarySearchTree Class [2/7]

```
def put(self, key, val):
        if self.root:
        self._put(key, val, self.root)
        else:
        self.root = TreeNode(key, val)
        self.size += 1
```

```
def _put(self, key, val, currentNode):
    key 값이 currentNode.key 보다 작으면
        currentNode에 left child가 있으면 _put() recursion with the left child
        currentNode에 left child가 없으면 key값을 가진 node를 생성하여 currentNode의 left child로 만듬

    key 값이 currentNode.key 보다 크면  (key이므로 값이 같을수는 없다)
        currentNode에 right child가 있으면 _put() recursion with the right child
        currentNode에 right child가 없으면 key값을 가진 node를 생성하여 currentNode에 right child로 만듬
```

BinarySearchTree Class [3/7]

```
def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    return None
```

```
def _get(self, key, currentNode):
    current node가 없으면 return None
    current node의 key가 원하는 key면 return current node
    current node의 key가 원하는 key보다 크면
        current node의 left child를 가지고 _get() recursion
    current node의 key가 원하는 key보다 작으면
        current node의 right child를 가지고 _get() recursion
```

```
def __setitem__(self, k, v):
    self.put(k, v)
```

```
def __getitem__(self, key):
    [redacted]

def __contains__(self, key):
    [redacted]
```

BinarySearchTree Class [4/7]

```
def delete(self, key):
        if self.size > 1:
            nodeToRemove = self._get(key, self.root)
            if nodeToRemove:
                self.remove(nodeToRemove)
                self.size -= 1
            else:    raise KeyError('Error, key is not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None
            self.size = 0
        else:    raise KeyError('Error, key not in tree')
```

BinarySearchTree Class [5/7]

def remove(self, currentNode):

current node가 leaf node이면
current node가 parent node의 left child이면 parent node의 left child part를 none으로 변경
current node가 parent node의 right child이면 parent node의 right child part를 none으로 변경

current node가 leaf node가 아니면 child가 1개 or 2개가 있는 경우
(A) left child와 right child를 다 가지고 있는 경우
replace current node with next largest node (only key and payload)
successor's right child move to its parent's position. This is done with 'node.sliceOut()'
(B) left child를 가지고 있는데
(B-1: LL type) current node가 parent node의 left child이면
parent node의 left child part를 current node의 left child로 변경
(B-2: RL type) current node가 parent node의 right child이면
parent node의 right child part를 current node의 left child로 변경
(C) right child를 가지고 있는데
(C-1: LR type) current node가 parent node의 left child이면
parent node의 right child part를 current node의 right child로 변경
(C-2: RR type) current node가 parent node의 right child이면
parent node의 right child part를 current node의 left child로 변경

Case Analysis of Remove [6/7]

A

```
#          parent
#            |
#        current node << REMOVE
#       /            \
#  left-child     right-child
#-----------------------------------------------------------
# REPLACE CURRENT NODE with NEXT LARGEST NODE (only key and payload)
#
#        current node << REMOVE
#       /            \
#  left-child     right-child << NEXT?
#                /
#           next-left-child
#          /
#   next-next-left-child  <<< NEXT!!! = (SUCCESSOR)
#                      \
#                       Successor's rightChild
#-----------------------------------------------------------
# Successor's right child move to its parent's position.
# This is done with 'node.sliceOut()'
#
#         (SUCCESSOR)
#       /            \
#  left-child     right-child << NEXT?
#                /
#           next-left-child
#          /
#        Successor's right child
```

B-1: LL case

```
# current node is left child of its parent node.
#                  parent
#                 /
#           currentnode << remove
#          /
#    childnode
```

B-2: RL case

```
# current node is right child of its parent node.
#                 parent
#                     \
#                   currentNode <<< REMOVE
#                    /
#                childnode
```

C-1: LR case

```
# current node is left child of its parent node.
#             parent
#            /
#       currentNode <<< REMOVE
#                  \
#                   childnode
```

C-2: RR case

```
# current node is right child of its parent node.
#        parent
#              \
#              currentNode <<< REMOVE
#                          \
#                          childnode
```

BinarySearchTree Class [7/7]

```
def main():
    bst = BinarySearchTree()
    input_data = (17, 5, 25, 2, 11, 29, 38, 9, 16, 7, 8)
    for i in input_data:
        bst.put(i, i)
    bst.root.inorder_traverse()
    #
    print('remove 5')
    bst.delete(5)
    bst.root.inorder_traverse()
    #
    print('put 39')
    bst.put(39, 39)
    bst.root.inorder_traverse()
```