

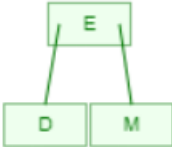
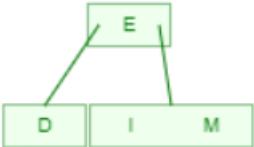




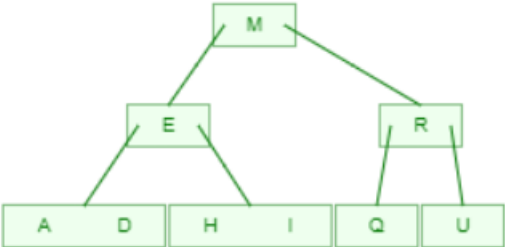
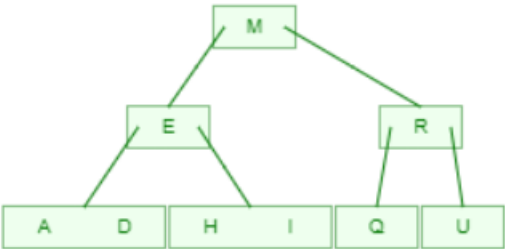
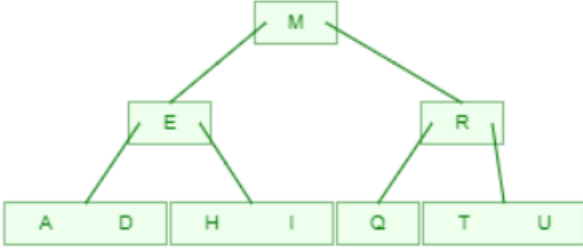
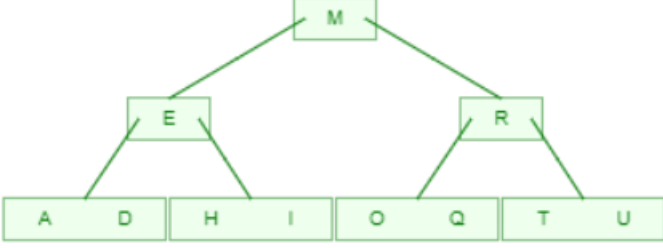
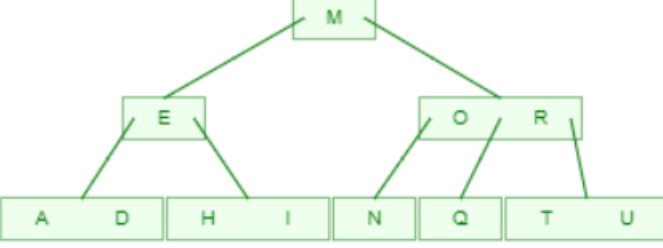


Problem Set 8



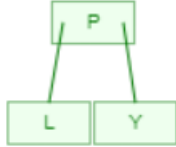
Daniel Wang (S01435533)

1. Insertion steps are visualized as follows:

Insert M	
Insert E	
Insert D	
Insert I	
Insert U	
Insert H	
Insert A	
Insert R	
Insert Q	

Insert U	 <p>(U already exists, so ignore it)</p>
Insert T	
Insert O	
Insert N	

2. Insertion steps are visualized as follows:

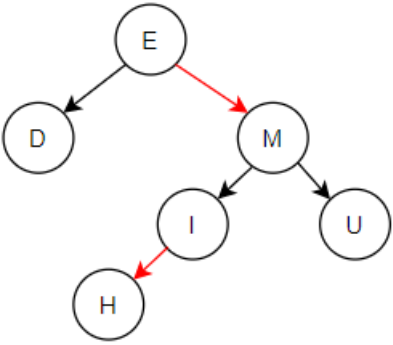
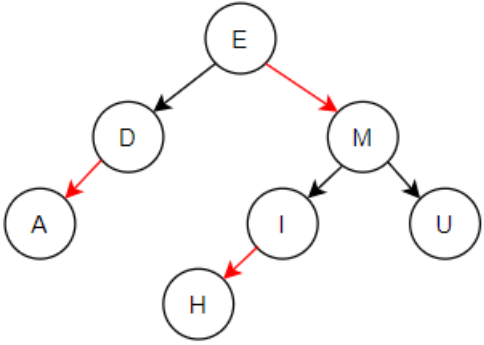
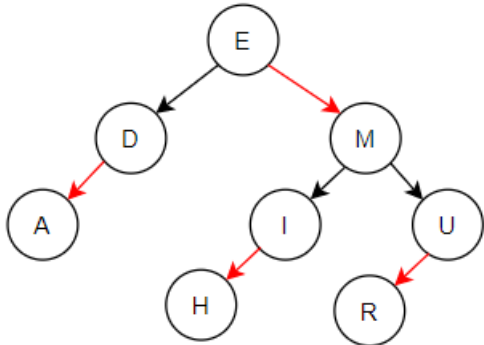
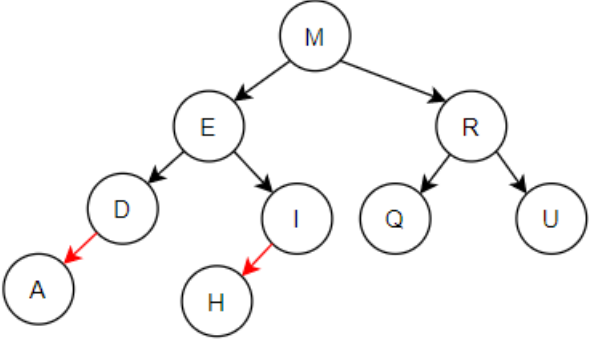
Insert Y	
Insert L	
Insert P	

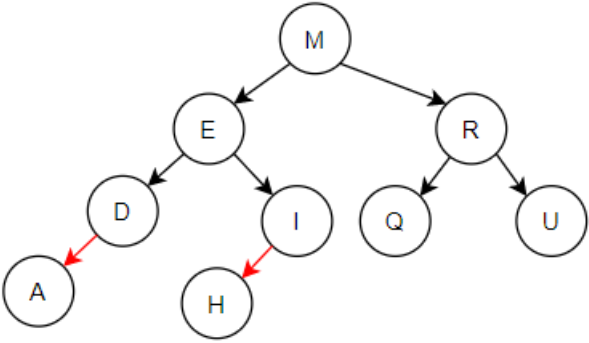
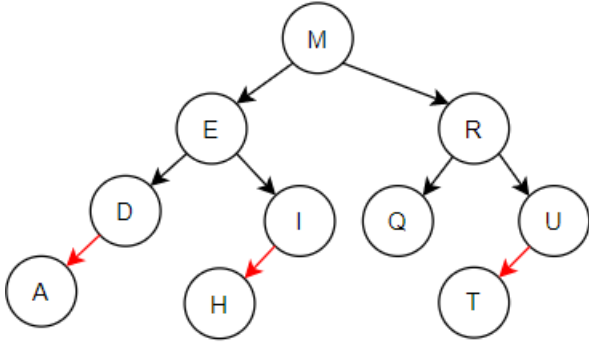
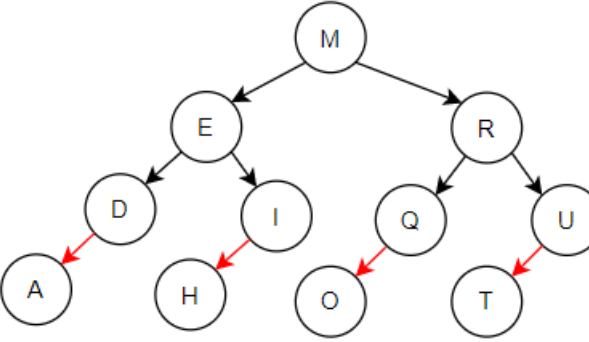
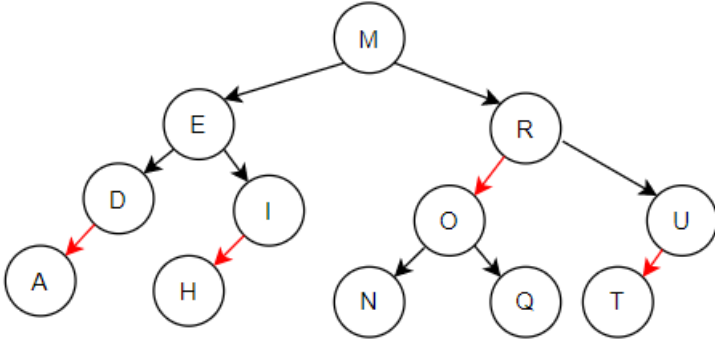
Insert M	<pre>graph TD; P[P] --- L1[L, M]; P --- Y1[Y];</pre>
Insert Z	<pre>graph TD; P[P] --- L1[L, M]; P --- Y1[Y, Z];</pre>
Insert H	<pre>graph TD; P[P] --- LP[L, P]; P --- Y1[Y]; LP --- H[H]; LP --- M[M]; Y1 --- Y2[Y, Z];</pre>
Insert C	<pre>graph TD; P[P] --- LP[L, P]; P --- Y1[Y]; LP --- C[C]; LP --- H[H]; LP --- M[M]; Y1 --- Y2[Y, Z];</pre>
Insert R	<pre>graph TD; P[P] --- L1[L]; P --- Y1[Y]; L1 --- C[C]; L1 --- H[H]; L1 --- M[M]; Y1 --- R[R]; Y1 --- Z[Z];</pre>
Insert A	<pre>graph TD; P[P] --- CL[C, L]; P --- Y1[Y]; CL --- A[A]; CL --- H[H]; CL --- M[M]; Y1 --- R[R]; Y1 --- Z[Z];</pre>
Insert E	<pre>graph TD; P[P] --- CL[C, L]; P --- Y1[Y]; CL --- A[A]; CL --- E[E]; CL --- H[H]; CL --- M[M]; Y1 --- R[R]; Y1 --- Z[Z];</pre>

Insert S	<pre> graph TD P[P] --> C[C] P --> Y[Y] C --> A[A] C --> E[E] C --> H[H] Y --> M[M] Y --> Z[Z] style S fill:#90EE90 </pre>
----------	---

3. Insertion steps are visualized as follows:

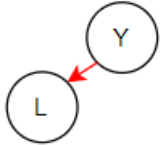
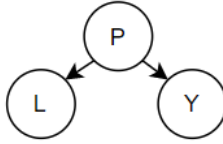
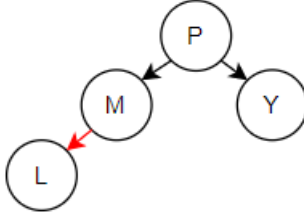
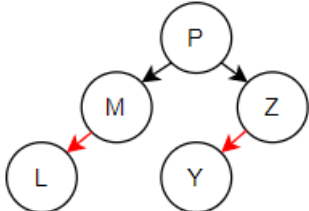
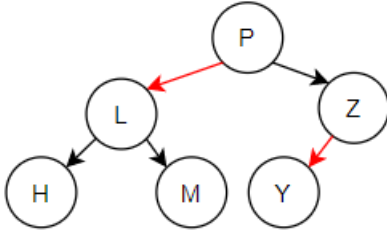
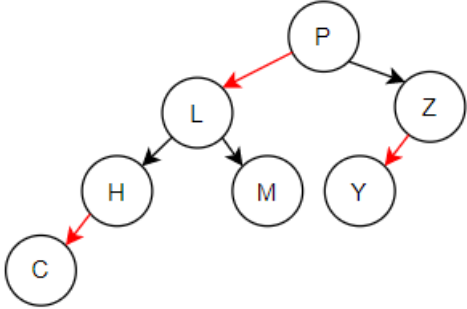
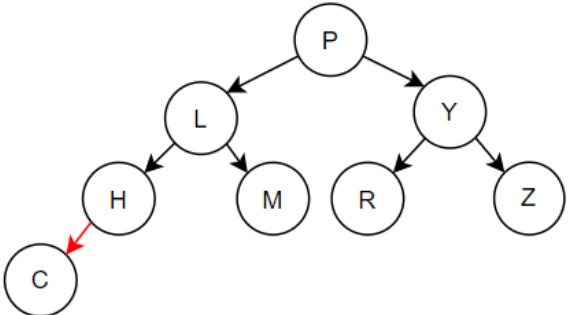
Insert M	<pre> graph TD M((M)) </pre>
Insert E	<pre> graph TD M((M)) -- red --> E((E)) </pre>
Insert D	<pre> graph TD E((E)) --> D((D)) E --> M((M)) </pre>
Insert I	<pre> graph TD E((E)) --> D((D)) E --> M((M)) M -- red --> I((I)) </pre>
Insert U	<pre> graph TD E((E)) --> D((D)) E -- red --> M((M)) M --> I((I)) M --> U((U)) </pre>

Insert H	 <pre>graph TD; E((E)) --> D((D)); E --> M((M)); M --> I((I)); M --> U((U)); I --> H((H)); style E fill:#fff,stroke:#000; style D fill:#fff,stroke:#000; style M fill:#fff,stroke:#000; style I fill:#fff,stroke:#000; style U fill:#fff,stroke:#000; style H fill:#fff,stroke:#000; linkStyle 1,2,3 stroke:#f00,stroke-width:2px; linkStyle 0 stroke:#000,stroke-width:2px;</pre>
Insert A	 <pre>graph TD; E((E)) --> D((D)); E --> M((M)); M --> I((I)); M --> U((U)); I --> H((H)); D --> A((A)); style E fill:#fff,stroke:#000; style D fill:#fff,stroke:#000; style M fill:#fff,stroke:#000; style I fill:#fff,stroke:#000; style U fill:#fff,stroke:#000; style H fill:#fff,stroke:#000; style A fill:#fff,stroke:#000; linkStyle 1,2,3,5 stroke:#f00,stroke-width:2px; linkStyle 0,4 stroke:#000,stroke-width:2px;</pre>
Insert R	 <pre>graph TD; E((E)) --> D((D)); E --> M((M)); M --> I((I)); M --> U((U)); I --> H((H)); D --> A((A)); U --> R((R)); style E fill:#fff,stroke:#000; style D fill:#fff,stroke:#000; style M fill:#fff,stroke:#000; style I fill:#fff,stroke:#000; style U fill:#fff,stroke:#000; style H fill:#fff,stroke:#000; style A fill:#fff,stroke:#000; style R fill:#fff,stroke:#000; linkStyle 1,2,3,5 stroke:#f00,stroke-width:2px; linkStyle 0,4,6 stroke:#000,stroke-width:2px;</pre>
Insert Q	 <pre>graph TD; M((M)) --> E((E)); M --> R((R)); E --> D((D)); E --> I((I)); D --> A((A)); I --> H((H)); R --> Q((Q)); R --> U((U)); style M fill:#fff,stroke:#000; style E fill:#fff,stroke:#000; style D fill:#fff,stroke:#000; style I fill:#fff,stroke:#000; style R fill:#fff,stroke:#000; style A fill:#fff,stroke:#000; style H fill:#fff,stroke:#000; style Q fill:#fff,stroke:#000; style U fill:#fff,stroke:#000; linkStyle 1,2,3,4,6 stroke:#f00,stroke-width:2px; linkStyle 0,5 stroke:#000,stroke-width:2px;</pre>

Insert U	 <p>(U already exists, so ignore it)</p>
Insert T	
Insert O	
Insert N	

4. Insertion steps are visualized as follows:

Insert Y	
----------	--

Insert L	
Insert P	
Insert M	
Insert Z	
Insert H	
Insert C	
Insert R	

Insert A	
Insert E	
Insert S	

5. We first extend the idea from deleting from an ordinary binary search tree. To delete a node in a BST, we should find its successor or predecessor (if successor does not exist), replacing them, and delete the successor or predecessor. Let's write the code for deleting node in ordinary BST:

```

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def predecessor(self, node):
        if not node:

```



```

        return None

    node = node.left
    while node and node.right:
        node = node.right

    return node

def successor(self, node):
    if not node:
        return None

    node = node.right
    while node and node.left:
        node = node.left

    return node

def deleteNode(self, root, key: int):
    if not root:
        return None

    if root.val < key:
        root.right = self.deleteNode(root.right, key)
    elif root.val > key:
        root.left = self.deleteNode(root.left, key)
    else:
        if not root.left and not root.right:
            root = None
        elif root.right:
            root.val = self.successor(root).val
            root.right = self.deleteNode(root.right, root.val)
        else:
            root.val = self.predecessor(root).val
            root.left = self.deleteNode(root.left, root.val)

    return root

```

Now consider RB tree. The logic is the same except that we need to maintain the red-black edge balance. To achieve this, we need to consider the case for its successor or predecessor.

- (1) If deleting successor without having a right child, we can easily delete it since it will not affect the RB balance
- (2) If deleting successor with a right child,

6. Since RB tree belong to a binary search tree, we can find the minimum by recursively finding the leftmost node. The Python code is shown below. Since RB tree is closed to a balance tree, the height of RB tree is $O(\lg n)$ where n is the number of nodes. Thus, finding minimum has a complexity of $O(\lg n)$.

```
class Solution:
```

```
def find_minimum_node(self, root):
    node = root
    while node.left:
        node = node.left

    return node.val
```

7. We can maintain a list which stores the result of inorder traversal of tree. The inorder list is a sorted array, and the length of the list is equal to the size of the tree. To find the median, we can easily index at the middle of the list. The code is shown below. When the size of the tree is n , computing inorder require $O(n)$ time complexity, and indexing the value is in constant time. Thus, the overall complexity is $O(n)$. If we want to insert or delete nodes in the tree, we need to re-compute the inorder list of the new tree.

```
class Solution:
    def inorder(self, node):
        if not node:
            return []

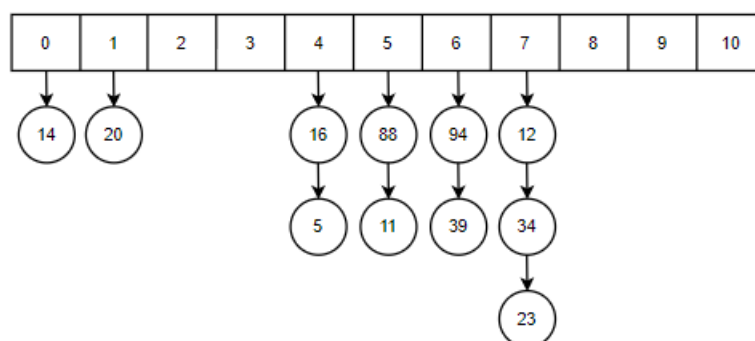
        return self.inorder(node.left) + [node.val] + self.inorder(node.right)

    def get_LLRB_median(self, root):
        inorder = self.inorder(root)
        n = len(inorder)

        if n == 0:
            return None
        if n % 2 == 1:
            return inorder[n//2]
        else:
            return (inorder[n//2-1] + inorder[n//2]) / 2
```

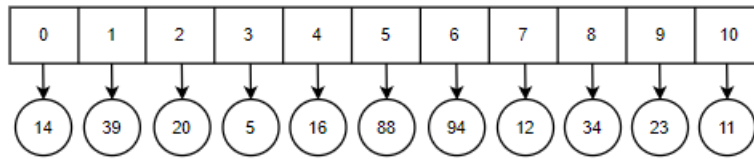
8. Below is the subproblem statements.

(1) The final hash table is shown below:



- (2) In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already

occupied, then we check for the next location. By doing so, the final hash table will be:



9. The Python code is shown below:

```
class Solution:
    def has_target_sum(self, set1, set2, target):
        for num in set1:
            if target-num in set2:
                return "yes"
        return "no"

H = {4, 5, 7, -1, -9, 22}
J = {0, -5, 10, 44, 100, -12}

s = Solution()
print(s.has_target_sum(H, J, 91))
print(s.has_target_sum(H, J, 92))
```

The console output will be ‘yes’ and ‘no’ respectively. Verifying whether an element is inside a set requires only $O(1)$ expected time, which makes this algorithm to have $O(n)$ expected time.

10. The worst case of the algorithm becomes $O(n^2)$. It happens when all elements in set2 collides into the same index, which makes the finding to be degraded into a linear search.

11. The Python code is shown below. I will maintain a temporary set “set2” which stores numbers we have scanned before. If there is a pair sum to target K given current iteration i , then there must exist a previously scanned element at iteration j such that $j < i$ and their sum is K . Thus, the element scanned at iteration j must be inside the temporary set “set2”.

```
class Solution:
    def has_target_sum_from_same_set(self, set1, target):
        set2 = set()
        for num in set1:
            if target-num in set2:
                return "yes"
            set2.add(num)

        return "no"
```

