cslsi-09-Muller_Hoch_Bharadhwaj

December 12, 2018

Group Members: Simon Muller, Alexander Hoch, Vinay Bharadhwaj Group Member Mail ID's: s6simue2@uni-bonn.de s6alhoch@uni-bonn.de vinaysbharadhwaj@gmail.com

0.0.1 Exercise 1

In [2]:
```
from graphviz import Digraph


        class BinaryTree():
            def __init__(self, root):
                self.root = root
                self.leftChild = None
                self.rightChild = None

            def insert(self, n):
                if self.root:
                    if n < self.root:
                        if self.leftChild is None:
                            self.leftChild = BinaryTree(n)
                        else:
                            self.leftChild.insert(n)
                    elif n > self.root:
                        if self.rightChild is None:
                            self.rightChild = BinaryTree(n)
                        else:
                            self.rightChild.insert(n)
                else:
                    self.root = n

            def delete(self, n):
                if self.root:
                    if n == self.root:
                        if self.rightChild is None:
                            self.root = self.leftChild.root
                        else:
                            self.root = self.rightChild.goLeft(None)
                            self.rightChild.leftChild = None
```

```
                self.leftChild.rightChild = None
        elif n < self.root:
            if self.leftChild is None:
                print('Value not present')
            else:
                self.leftChild.delete(n)
        elif n > self.root:
            if self.rightChild is None:
                print('Value not present')
            else:
                self.rightChild.delete(n)
    else:
        print('Tree not present')

def goLeft(self, x):
    self.x = x
    if self.leftChild is None:
        self.x = self.root
        return self.x
    else:
        self.x = self.leftChild.goLeft(self.x)
        return(self.x)

def search(self, n):
    if self.root:
        if n < self.root:
            if self.leftChild is None:
                return str(n) + " Not Found"
            return self.leftChild.search(n)
        elif n > self.root:
            if self.rightChild is None:
                return str(n) + " Not Found"
            return self.rightChild.search(n)
        else:
            return(str(self.root) + ' is found')
    else:
        return str(n) + " Not Found"

def traverse(self, order, i=0):
    x = None
    if self.root:
        if self.leftChild:
            if self.rightChild:
                x = [0, self.root, self.leftChild.root, self.rightChild.root]
            else:
                x = [3, self.root, self.leftChild.root, None]
        else:
            if self.rightChild:
```

```
                    x = [2, self.root, None, self.rightChild.root]
                else:
                    x = [23, self.root, None, None]
        if(order == 'pre'):
            print('PreOrder Traversal, Level = {}: {}'.format(i, x[1:]))
            i += 1
            if(x[0] == 0):
                self.leftChild.traverse(order, i)
                self.rightChild.traverse(order, i)
            elif(x[0] == 3):
                self.leftChild.traverse(order, i)
            elif(x[0] == 2):
                self.rightChild.traverse(order, i)
            elif(x[0] == 23):
                print('End of level {}', format(i))
        elif(order == 'in'):
            x[1], x[2] = x[2], x[1]
            print('InOrder Traversal, Level = {}: {}'.format(i, x[1:]))
            i += 1
            if(x[0] == 0):
                self.leftChild.traverse(order, i)
                self.rightChild.traverse(order, i)
            elif(x[0] == 3):
                self.leftChild.traverse(order, i)
            elif(x[0] == 2):
                self.rightChild.traverse(order, i)
            elif(x[0] == 23):
                print('End of level {}', format(i))
        elif(order == 'post'):
            x[1], x[2], x[3] = x[3], x[2], x[1]
            print('PostOrder Traversal, Level = {}: {}'.format(i, x[1:]))
            i += 1
            if(x[0] == 0):
                self.leftChild.traverse(order, i)
                self.rightChild.traverse(order, i)
            elif(x[0] == 3):
                self.leftChild.traverse(order, i)
            elif(x[0] == 2):
                self.rightChild.traverse(order, i)
            elif(x[0] == 23):
                print('End of level {}', format(i))

    def dotgen(self):
        self.dot = Digraph()
        if self.root:
            if self.leftChild:
                if self.rightChild:
                    self.dot.node(str(self.root), str(self.root))
```

```
                self.dot.node(str(self.leftChild.root), str(self.leftChild.root))
                self.dot.node(str(self.rightChild.root), str(self.rightChild.root))
                self.dot.edge(str(self.root), str(self.leftChild.root))
                self.dot.edge(str(self.root), str(self.rightChild.root))
                temp = self.leftChild.dotgen()
                for idx, i in enumerate(temp):
                    if(idx == 0 or idx == 1 or i == '}'):
                        continue
                    if('label' in i):
                        self.dot.node(i.split(" [label=", 1)[0][1:], \
                                      i.split("label=", 1)[1][:-1])
                    elif('->' in i):
                        self.dot.edge(i.split(" ->", 1)[0][1:], \
                                      i.split(" ->", 1)[1][1:])

                temp = self.rightChild.dotgen()
                for idx, i in enumerate(temp):
                    if(idx == 0 or idx == 1 or i == '}'):
                        continue
                    if('label' in i):
                        self.dot.node(i.split(" [label=", 1)[0][1:], \
                                      i.split("label=", 1)[1][:-1])
                    elif('->' in i):
                        self.dot.edge(i.split(" ->", 1)[0][1:],\
                                      i.split(" ->", 1)[1][1:])
                return(self.dot)
            else:
                self.dot.node(str(self.root), str(self.root))
                self.dot.node(str(self.leftChild.root), str(self.leftChild.root))
                self.dot.edge(str(self.root), str(self.leftChild.root))
                temp = self.leftChild.dotgen()
                for idx, i in enumerate(temp):
                    if(idx == 0 or idx == 1 or i == '}'):
                        continue
                    if('label' in i):
                        self.dot.node(i.split(" [label=", 1)[0][1:], \
                                      i.split("label=", 1)[1][:-1])
                    elif('->' in i):
                        self.dot.edge(i.split(" ->", 1)[0][1:], \
                                      i.split(" ->", 1)[1][1:])
                return(self.dot)
        else:
            if self.rightChild:
                self.dot.node(str(self.root), str(self.root))
                self.dot.node(str(self.rightChild.root), str(self.rightChild.root))
                self.dot.edge(str(self.root), str(self.rightChild.root))
                temp = self.rightChild.dotgen()
                for idx, i in enumerate(temp):
```

```
                            if(idx == 0 or idx == 1 or i == '}'):
                                continue
                            if('label' in i):
                                self.dot.node(i.split(" [label=", 1)[0][1:], \
                                              i.split("label=", 1)[1][:-1])
                            elif('->' in i):
                                self.dot.edge(i.split(" ->", 1)[0][1:], \
                                              i.split(" ->", 1)[1][1:])
                        return(self.dot)
                    else:
                        self.dot.node(str(self.root), str(self.root))
                        return(self.dot)

            def visualize(self, name):
                self.x = self.dotgen()
                self.x.render(name, cleanup=True)


        X = BinaryTree(10)
        for i in [5, 7, 1, 15, 3, 6, 9, 8, 11]:
            X.insert(i)
        print(X.search(14))
        print(X.search(7))
        X.visualize('Before_Delete_vis')
        X.delete(7)
        print(X.search(7))
        X.visualize('After_Delete_vis')
        X.traverse('pre')
        print('\n\n')
        X.traverse('in')
        print('\n\n')
        X.traverse('post')
```

```
14 Not Found
7 is found
7 Not Found
PreOrder Traversal, Level = 0: [10, 5, 15]
PreOrder Traversal, Level = 1: [5, 1, 8]
PreOrder Traversal, Level = 2: [1, None, 3]
PreOrder Traversal, Level = 3: [3, None, None]
End of level {} 4
PreOrder Traversal, Level = 2: [8, 6, 9]
PreOrder Traversal, Level = 3: [6, None, None]
End of level {} 4
PreOrder Traversal, Level = 3: [9, None, None]
End of level {} 4
PreOrder Traversal, Level = 1: [15, 11, None]
PreOrder Traversal, Level = 2: [11, None, None]
```

```
End of level {} 3
```

```
InOrder Traversal, Level = 0: [5, 10, 15]
InOrder Traversal, Level = 1: [1, 5, 8]
InOrder Traversal, Level = 2: [None, 1, 3]
InOrder Traversal, Level = 3: [None, 3, None]
End of level {} 4
InOrder Traversal, Level = 2: [6, 8, 9]
InOrder Traversal, Level = 3: [None, 6, None]
End of level {} 4
InOrder Traversal, Level = 3: [None, 9, None]
End of level {} 4
InOrder Traversal, Level = 1: [11, 15, None]
InOrder Traversal, Level = 2: [None, 11, None]
End of level {} 3
```

```
PostOrder Traversal, Level = 0: [15, 5, 10]
PostOrder Traversal, Level = 1: [8, 1, 5]
PostOrder Traversal, Level = 2: [3, None, 1]
PostOrder Traversal, Level = 3: [None, None, 3]
End of level {} 4
PostOrder Traversal, Level = 2: [9, 6, 8]
PostOrder Traversal, Level = 3: [None, None, 6]
End of level {} 4
PostOrder Traversal, Level = 3: [None, None, 9]
End of level {} 4
PostOrder Traversal, Level = 1: [None, 11, 15]
PostOrder Traversal, Level = 2: [None, None, 11]
End of level {} 3
```

Subtask d) Deletion order is not important because it does not change the relationships (greater or smaller) between the nodes.

In Case of leaf node deletion you do not change the rest of the tree so the order is unimportant.

In case of a node with only one child you just replace the node by its child which also leads to the same tree if you switch the order (case 1).

In case of a node with two childs the firstorder successor does have all the same size relations as the original node so the second delete will affect the same node as if you did it the other way around.

Subtask e) inorder for BST.

0.1 Exercise 2

```
In [3]: def Q21(a, l, r, i):
            i = i - 1
            if l >= r:
                return a[i]
            q = partition(a, l, r)
            if i < q:
                return Q21(a, l, q - 1, i + 1)
            else:
                return Q21(a, q + 1, r, i + 1)


        def partition(a, l, r):
            p = a[r]
            k = l - 1
            for j in range(l, r):
                if a[j] <= p:
                    k += 1
                    a[k], a[j] = a[j], a[k]
            a[k + 1], a[r] = a[r], a[k + 1]
            return k + 1


        a = [5, 3, 1, 7, 2, 3, 1]
        print(Q21(a, 0, len(a) - 1, 3))
        a = [5, 3, 1, 7, 2, 3, 1]
        print(Q21(a, 0, len(a) - 1, 4))
        a = [5, 3, 1, 7, 2, 3, 1]
        print(Q21(a, 0, len(a) - 1, 7))
        a = [5, 3, 1, 7, 2, 3, 1]
        print(Q21(a, 0, len(a) - 1, 1))
```

```
2
3
7
1
```

```
In [4]: def Q22(a, i):
            if len(a) <= 1:
                return a
            done_list = []
            not_list = [[] for x in range(len(a))]
            for j in a:
                if i >= len(j):
                    done_list.append(j)
                else:
                    not_list[ord(j[i]) - ord('a')].append(j)
```

```
        # ord gets the unicode value, then to get the index to append at, subtracting with the
            x = [Q22(b, i + 1) for b in not_list]
            return done_list + [z for y in x for z in y]


        a = ["b", "cda", "db", "da", "ab", "a"]
        print(Q22(a, 0))
```

```
['a', 'ab', 'b', 'cda', 'da', 'db']
```