DFS algorithm

```
from collections import defaultdict
class DFS:
  def __init__(self,n):
    self.nodes = n
    self.colors = ['white'] * n
    self.start_time = [0] * n
    self.finish_time = [0] * n
    self.tree = defaultdict(list)

  def add_edges(self,parent,child):
    self.tree[parent].append(child)


  def dfs_visit(self,u):
    self.time += 1
    self.start_time[u] = self.time
    self.colors[u] = 'gray'
    for child in self.tree[u]:
      if self.colors[child] == 'white':
        self.dfs_visit(child)
    self.time += 1
    self.finish_time[u] = self.time
    self.colors[u] = 'black'


def dfs(self):
  for node in range(self.nodes):
    self.time = 0
    if self.colors[node] == 'white':
      self.dfs_visit(node)
  print(self.start_time)
  print(self.finish_time)
  print(self.tree)

def search_pred(self,root,child):


d = DFS(6)
d.add_edges(0, 1)
d.add_edges(0, 2)
d.add_edges(1, 2)
d.add_edges(2, 3)
d.add_edges(3, 1)
d.add_edges(4, 3)
d.add_edges(4, 5)
d.add_edges(5, 5)
d.dfs()
```

```
[1, 2, 3, 4, 1, 2]
[8, 7, 6, 5, 4, 3]
defaultdict(<class 'list'>, {0: [1, 2], 1: [2], 2: [3], 3: [1], 4: [3, 5], 5: [5]})
```

Printing predecessor of an element using DFS.

```
class node:
  def __init__(self,info):
    self.info = info
    self.left = None
    self.right = None

def insert(ptr,key):
  if ptr is None:
    ptr = node(key)
  elif key <= ptr.info:
    ptr.left = insert(ptr.left,key)
  elif key > ptr.info:
    ptr.right = insert(ptr.right,key)
  return ptr

def searchPredecessor(root, key):
    if not root:
      return False
    if root.info == key:
      return True
    if (searchPredecessor(root.left,key) or searchPredecessor(root.right,key)):
      print(root.info)
      return 1
    return 0

if __name__=='__main__':
    root=None
    root=insert(root,10)
    root=insert(root,5)
    root=insert(root,15)
    root=insert(root,30)
    searchPredecessor(root,30)
```

```
15
10
```