In [15]:

```
class Node:
    def __init__ (self,state,parent,actions,totalcost):
        self.state=state
        self.parent=parent
        self.actions=actions
        self.totalcost=totalcost

graph={'A':Node('A',None,['B','E','C'],None),
       'B':Node('B',None,['D','E','A'],None),
       'C':Node('C',None,['A','F','G'],None),
       'D':Node('D',None,['B','E'],None),
       'E':Node('E',None,['A','B','D'],None),
       'F':Node('F',None,['C'],None),
       'G':Node('G',None,['C'],None),
      }

def DFS():
    initialState='A'
    goalState='D'


    graph={'A':Node('A',None,['B','E','C'],None),
       'B':Node('B',None,['D','E','A'],None),
       'C':Node('C',None,['A','F','G'],None),
       'D':Node('D',None,['B','E'],None),
       'E':Node('E',None,['A','B','D'],None),
       'F':Node('F',None,['C'],None),
       'G':Node('G',None,['C'],None),
      }

    frontier=[initialState]
    explored=[]

    while len(frontier)!=0:
        currentnode=frontier.pop(len(frontier)-1)
        print(currentnode)
        explored.append(currentnode)
        currentchildren=[]
        for child in graph(currentnode).actions:
            if child not in frontier and child not in explored:
                graph[child].parent=cuurentnode
                if graph[child].state==goalstate:
                    print(explored)
                    return actionsequence(graph,initialstate,goalstate)
                currentchildren=currentchildren+1
                frontier.append(child)
        if currentchildren==0:
            del explored[len(explored)-1]

solution=DFS()
print(solution)

def actionsequence(graph,initialstate,goalstate):
    solution=[goalstate]
    currentparent=graph[goalstate].parent
    while currentparent!=None:
        solution.append(currentparent)
        currentparent=graph(currentparent).parent
    solution.reverse()
```

```
    return solution

solution=DFS()
print(solution)
```

```
A

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_1136/3056991769.py in <module>
     48             del explored[len(explored)-1]
     49
---> 50 solution=DFS()
     51 print(solution)
     52

~\AppData\Local\Temp/ipykernel_1136/3056991769.py in DFS()
     37         explored.append(currentnode)
     38         currentchildren=[]
---> 39         for child in graph(currentnode).actions:
     40             if child not in frontier and child not in explored:
     41                 graph[child].parent=cuurentnode

TypeError: 'dict' object is not callable
```

In [ ]:

```
#solution:

Final path:['D', 'B', 'A', 'C']
Explored:D, E, A
```

In [58]:

```
# lab task 2 Practice:

class Trie:
    def __init__(self):
        self.character = {}
        self.isLeaf = False


def insert(root, s):
    curr = root

    for ch in s:
        curr = curr.character.setdefault(ch, Trie())
    curr.isLeaf = True



row = [-1, -1, -1, 0, 1, 0, 1, 1]
col = [-1, 1, 0, -1, -1, 1, 0, 1]



def isSafe(x, y, processed, board, ch):
    return (0 <= x < len(processed)) and (0 <= y < len(processed[0])) and \
            not processed[x][y] and (board[x][y] == ch)


def searchBoggle(root, board, i, j, processed, path, result):

    if root.isLeaf:

        result.add(path)


    processed[i][j] = True


    for key, value in root.character.items():
        for k in range(len(row)):
            if isSafe(i + row[k], j + col[k], processed, board, key):
                searchBoggle(value, board, i + row[k], j + col[k],
                             processed, path + key, result)


    processed[i][j] = False



def searchInBoggle(board, words):

    result = set()
    if not board or not len(board):
        return
    root = Trie()
    for word in words:
        insert(root, word)
    (M, N) = (len(board), len(board[0]))


```

```
    processed = [[False for x in range(N)] for y in range(M)]
    for i in range(M):
        for j in range(N):
            ch = board[i][j]
            if ch in root.character:
                searchBoggle(root.character[ch], board, i, j, processed, ch, result)


    return result


if __name__ == '__main__':
    board = [
        ['M','S','E','F'],
        ['R','A','T','D'],
        ['L','O','N','E'],
        ['K','A','F','B']
    ]

    words = ['START', 'NOTE', 'SAND', 'STONED']
    searchInBoggle(board, words)

    validWords = searchInBoggle(board, words)
    print(validWords)

```

```
{'NOTE', 'STONED', 'SAND'}
```

In [ ]:

```
1
```

In [ ]:

```
1
```