2022-1 자료구조 과제 2

2112979 어연수

(1) 138 쪽, 5 장-15 번 (30 점)

```
def move(self, item, index):

    #존재하던 요소라면 거기부터 이동
    crt = self.find(item.name)
    #없었다면 헤드부터
    if not crt: crt = self.head.rlink
    else: crt.data = 0   #원래 장소 0으로 설정
    move = index

    #index가 음수값
    if index<0:
        item.course = not item.course
        move = abs(move)

    for i in range(move):
        if item.course:
            crt = crt.rlink
            if crt == self.head:
                crt = crt.rlink
        else:
            crt = crt.llink
            if crt == self.head:
                crt = crt.llink

    #index가 음수라서 바꾸었던 방향 원래대로
    if index<0:
        item.course = not item.course

    #도착한 곳에 이미 데이터가 있다면
    if crt.data:
        global judged
        judged = True
    crt.data = item.name
```

각 플레이어가 이동할 때마다

없던 노드라면 (게임 초기세팅) -> 헤드부터 index 만큼 이동

받은 값이 음수일 경우 (1,1) -> 방향을 바꿔서 index 의 절댓값만큼 이동

* 확장성을 고려해서 백도만 가능한게 아닌 음수로 받은 값만큼 반대로 이동할 수 있도록 구현

일반적인 이동 -> item 의 방향에 따라 index 만큼 이동

도착한 곳에 다른 플레이어가 있다면 judged 변수 True 로 변경 -> 게임 종료

게임 초기 구성:

게임판인 16 길이의 Linkedlist 에는 값이 0 인 node 가 가득 차있다.

Players 배열에 name 과 course(이동방향) 값을 가진 Player 객체 p1 과 p2 가 등록됨

리스트 0 번과 8 번에 p1 과 p2 의 name 을 넣어주고 게임이 시작됨

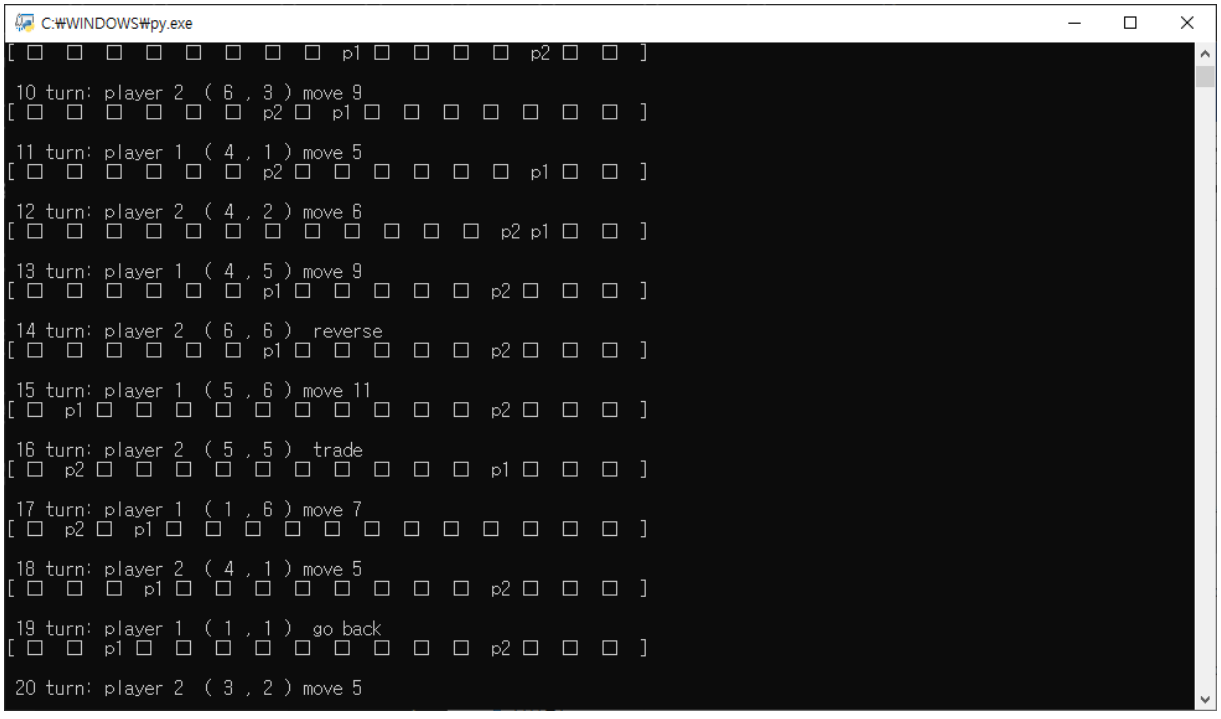
매 턴 주사위 1, 2 의 값을 1~6 의 랜덤값으로 받아서

(6,6) (5,5) (1,1) 라면 특수한 이벤트를 수행하고 아니라면 일반적인 이동을 수행하고 상태를 출력함

move 함수에서 judged 변수가 True 로 변경되었다면 승자를 출력하고 게임 종료됨

- 실행 화면


C:₩WINDOWS₩py.exe
Game Start!
[p1 □ □ □ □ □ □ □ p2 □ □ □ □ □ □ □ □]
1 turn: player 1 (3 , 3) move 6
[□ □ □ □ □ □ p1 □ p2 □ □ □ □ □ □ □]
2 turn: player 2 (6 , 4) move 10
[□ □ p2 □ □ □ p1 □ □ □ □ □ □ □ □]
3 turn: player 1 (3 , 6) move 9
[□ □ p2 □ □ □ □ □ □ □ □ □ □ □ p1]
4 turn: player 2 (3 , 4) move 7
[□ □ □ □ □ □ □ □ □ p2 □ □ □ □ □ p1]
5 turn: player 1 (1 , 3) move 4
[□ □ □ p1 □ □ □ □ □ p2 □ □ □ □ □ □]
6 turn: player 2 (5 , 3) move 8
[□ p2 □ p1 □ □ □ □ □ □ □ □ □ □ □ □]
7 turn: player 1 (1 , 1) go back
[□ p2 p1 □ □ □ □ □ □ □ □ □ □ □ □ □]
8 turn: player 2 (2 , 3) move 5
[□ □ p1 □ □ □ p2 □ □ □ □ □ □ □ □ □]
9 turn: player 1 (1 , 1) go back
[□ p1 □ □ □ □ p2 □ □ □ □ □ □ □ □ □]
10 turn: player 2 (5 , 1) move 6
[□ p1 □ □ □ □ □ □ □ □ □ p2 □ □ □]
11 turn: player 1 (6 , 5) move 11
[□ □ □ □ □ □ □ □ □ □ □ □ p1 □ □ □]
p1 is winner!


C:₩WINDOWS₩py.exe
[□ □ □ □ □ □ □ □ p1 □ □ □ □ p2 □ □]
10 turn: player 2 (6 , 3) move 9
[□ □ □ □ □ □ p2 □ p1 □ □ □ □ □ □ □]
11 turn: player 1 (4 , 1) move 5
[□ □ □ □ □ □ p2 □ □ □ □ □ □ p1 □ □]
12 turn: player 2 (4 , 2) move 6
[□ □ □ □ □ □ □ □ □ □ □ □ p2 p1 □ □]
13 turn: player 1 (4 , 5) move 9
[□ □ □ □ □ □ p1 □ □ □ □ □ p2 □ □ □]
14 turn: player 2 (6 , 6) reverse
[□ □ □ □ □ □ p1 □ □ □ □ □ p2 □ □ □]
15 turn: player 1 (5 , 6) move 11
[□ p1 □ □ □ □ □ □ □ □ □ □ p2 □ □ □]
16 turn: player 2 (5 , 5) trade
[□ p2 □ □ □ □ □ □ □ □ □ □ p1 □ □ □]
17 turn: player 1 (1 , 6) move 7
[□ p2 □ p1 □ □ □ □ □ □ □ □ □ □ □ □]
18 turn: player 2 (4 , 1) move 5
[□ □ □ p1 □ □ □ □ □ □ □ □ p2 □ □ □]
19 turn: player 1 (1 , 1) go back
[□ □ p1 □ □ □ □ □ □ □ □ □ p2 □ □ □]
20 turn: player 2 (3 , 2) move 5

(3) 232 쪽, 최소 비용 신장 트리 (Prim 방법) (25 점)

```python
class Graph:
    def __init__(self):
        self.graph = {}      # 그래프의 기본 모양 (중복있음)
        self.v_list = []     # 현재까지 확장된 트리의 vertex 리스트
        self.e_list = []     # 현재 트리에서 선택할 수 있는 edge 리스트
        self.edge = []       # 실제로 추가된 edge
        self.total = 0       # edge의 총 cost
```

graph 에 각 노드마다 연결될 수 있는 간선이 전부 표시됨

```python
# 프림 알고리즘 구현
def prim(self, start):
    # 시작 지점부터 v_list와 e_list에 추가함
    self.v_list.append(start)
    print('\nStart vertex =', self.v_list)
    self.e_list.extend(self.graph[start])
    self.e_list = self.sort_edge(self.e_list)

    while self.e_list:  # 더 선택할 수 있는 간선이 없을때까지 반복
        cost = self.e_list[0][0]
        v1 = self.e_list[0][1]
        v2 = self.e_list[0][2]
        print('\n(',v1,',',v2,')','cost = ',cost)

        # 사용한 간선 리스트에서 삭제
        del self.e_list[0]

        # 간선이 v_list에 이미 있는 정점를 가르킨다면 사이클을 형성하므로 연결안함
        if v1 in self.v_list and v2 in self.v_list:
            print("rejected for cycle")
            continue

        # 간선을 연결하고 합계 비용 계산
        self.edge.append((v1, v2, cost))
        self.total += cost

        # 확장된 정점에서 선택할 수 있는 간선이 v_list에 이미 있다면 중복이므로 빼고 추가
        extend_list = []
        for s1, s2, cost in self.graph[v2]:
            if s2 not in self.v_list:
                extend_list.append((s1, s2, cost))
        extend_list = self.sort_edge(extend_list)

        # 현재 정점과 간선 리스트 확장
        self.e_list.extend(extend_list)
        self.e_list.sort()
        self.v_list.append(v2)

        print('Spanning Tree =',self.v_list)
```

연결할 간선이 v_list 에 이미 있는 간선을 가르킨다면 사이클을 형성하게 되므로 넘어감

아니라면 v_list 에 정점을 추가함

e_list 에 추가한 정점에서 가능한 간선들을 확장시키고 비용순으로 정렬함

정점을 추가하는데 사용한 간선은 e_list 에서 제거

-> 더 이상 e_list 에 연결할 수 있는 간선이 없을때까지 반복

- 실행 화면

```
network= [(1, 5, 6), (1, 6, 8), (2, 3, 17), (2, 6, 9), (5, 6, 7), (3, 7, 15), (3, 4, 5), (3, 8, 3), (4, 8, 4), (6, 7, 10)]
start node : 3

Start vertex = [3]

( 3 , 8 ) cost =  3
Spanning Tree = [3, 8]

( 8 , 4 ) cost =  4
Spanning Tree = [3, 8, 4]

( 3 , 4 ) cost =  5
rejected for cycle

( 3 , 7 ) cost =  15
Spanning Tree = [3, 8, 4, 7]

( 7 , 6 ) cost =  10
Spanning Tree = [3, 8, 4, 7, 6]

( 6 , 5 ) cost =  7
Spanning Tree = [3, 8, 4, 7, 6, 5]

( 5 , 1 ) cost =  6
Spanning Tree = [3, 8, 4, 7, 6, 5, 1]

( 6 , 1 ) cost =  8
rejected for cycle

( 6 , 2 ) cost =  9
Spanning Tree = [3, 8, 4, 7, 6, 5, 1, 2]

( 3 , 2 ) cost =  17
rejected for cycle

Spanning tree vertices =  [3, 8, 4, 7, 6, 5, 1, 2]
spanning tree edges =  [(3, 8, 3), (8, 4, 4), (3, 7, 15), (7, 6, 10), (6, 5, 7), (5, 1, 6), (6, 2, 9)]
cost total =  54
```

(4) 286 쪽 (이진 탐색 트리) (25)

```
else:                                # 자식이 2개인 노드 삭제
    if self.height(node.right) > self.height(node.left): # 오른쪽 서브트리가 더 깊은 경우
        right_min = self.find_min(node.right)       # 오른쪽의 최소값
        node.data = right_min.data
        node.right = self.delete_node(node.right, right_min.data)
        return node
    else:
        left_max = self.find_max(node.left)         # 왼쪽의 최댓값
        node.data = left_max.data
        node.left = self.delete_node(node.left, left_max.data)
        return node
```

삭제할 노드의 자식 노드가 2 개일 때

-> 오른쪽 서브 트리가 더 깊으면 오른쪽 서브 트리의 최소값으로 대체해줌

아니거나 같으면 그대로 왼쪽

* 마지막에서 자식 노드가 한 개가 아니면 결과에서 높이 차이가 발생하지 않는데, 만약 오른쪽이 훨씬 깊은데도 마지막 자식 노드가 두 개라면 결과적으로 높이 차이가 발생하지 않아서 계속 왼쪽이 선택된다면 효율적이지 않다고 생각해서 그냥 더 깊은 서브 트리 쪽을 선택하도록 함

```
def find_min(self, root):
    if not root: return Node
    node = root
    while node.left:
        node = node.left
    return node

def height(self, root):          # 루트부터 트리의 높이
    if root is None:
        return -1
```

-> 추가된 함수

Height: 선택된 노드로부터 높이를 구함

Find_min: 선택된 노드로부터 자식 노드들 중에서 최소값을 구함

- 실행화면

```
30
20 30
10 20 30
10 20 40 30
10 20 60 40 30
10 20 50 60 40 30
10 25 20 50 60 40 30

10 25 20 50 60 40
10 20 50 60 40
10 50 60 20
10 60 20
60 10
10
```

<자체 평가 보고서>

문항 (배점)	채점 기준 (감점)	자체 점수
(1)술래잡기 (30 점)	- 미구현, 실행 안됨 (-30) - 구현하였으나 실행 안됨 (-25) - 실행은 되지만, 일부 기능 누락시 (-5/기능)	30 / 30
(2)연결리스트 최대힙 (20 점)	- 미구현, 실행 안됨 (-20) - 구현하였으나 실행 안됨 (-15) - 실행은 되지만 기능 구현 누락 (-5/기능)	0 / 20
(3)최소비용신장트리 (25 점)	- 미구현, 실행 안됨 (-25) - 구현하였으나 실행 안됨 (-20) - 실행은 되지만 일부 기능 구현 누락 (-5/기능)	25 / 25
(4)이진탐색트리 삭제 (25 점)	- 미구현, 실행 안됨 (-25) - 구현하였으나 실행 안됨 (-20) - 실행은 되지만 일부 기능 구현 누락 (-5/기능)	25 / 25
합계 점수		80 / 100