**자료구조 06**

****

**중앙대학교**

**자료구조 05 분반**

**20165079 김영빈**

**목차**

과제

[**1.** **지하철 빠른 길 찾기** 3](#_Toc73903007)

[**2.** **사전 탐색 트리 만들기** 10](#_Toc73903008)

# **지하철 빠른 길 찾기**

* 1. **클래스와 메서드 정의**

**import** collections

**import** random

*#역의 정보를 담아 둘 Station Class*

class Station:

**def** \_\_init\_\_(*self*, **name**):

*self*.name**=**name

*self*.adjacent\_stations **=** {}

**def** \_\_str\_\_(*self*):

**return** *self*.name

*#서울 지하철 노선 저장 및 경로 탐색을 위한 Class*

class Map:

**def** \_\_init\_\_(*self*, **stations**, **connections**):

*self*.stations **=** stations

*self*.connect\_stations(connections)

*self*.path **=** None

*#BFS를 이용하여 연결되어 저장되어 있는 모든 역 이름을 출력한다.*

**def** get\_available\_station(*self*):

**if** **not** len(*self*.stations):

**return** None

**else**:

*#random한 역 하나를 추출하여 그 역에서부터 탐색을 진행한다.*

            start **=** random.sample(list(*self*.stations), 1)[0]

            station\_list **=** [start]

            queue **=** collections.deque([start])

**while** queue:

                vertex **=** queue.popleft()

**for** next\_station **in** *self*.stations[vertex].adjacent\_stations:

**if** next\_station **not** **in** station\_list:

                        station\_list.append(next\_station)

                        queue.append(next\_station)

*#추출된 역은 정렬하여 화면에 표시한다.*

        station\_list.sort()

**for** station **in** station\_list:

            print(station, **end=**"  ")

*#역과 역 사이를 연결하고 Weight를 지정한다.*

**def** connect\_stations(*self*, **connections**):

**for** connection **in** connections:

**if** len(connection) **!=** 3:

**continue**

            station1 **=** connection[0]

            station2 **=** connection[1]

            weight **=** connection[2]

*self*.stations[station1].adjacent\_stations[station2] **=** weight

*self*.stations[station2].adjacent\_stations[station1] **=** weight

*#다익스트라 알고리즘을 사용하여 최소 경로를 구한다.*

**def** get\_path(*self*, **start**, **end**):

*#distance는 경로 확정이 되지 않은 역들이 모여있는 Dictionary*

*#final\_distance는 경로 확정이 된 역들이 모여있는 Dictionary*

*#첫 initialize*

        distance **=** {start:[0, start]}

        final\_distance **=** {}

*#시작역에서부터 주변에 있는 역들과 그 weight를 먼저 distance에 넣음*

        cur **=** *self*.stations[start]

**for** station **in** cur.adjacent\_stations:

            distance[station] **=** [cur.adjacent\_stations[station], cur.name]

*#distance에 들어있는 역들 중 가장 weight 즉 거리가 짧은 역을 distance에서 제거하고 final\_distance에 넣음*

*#이 경우에는 가장 첫 역이 들어가게 됨*

        nearest, weight **=** sorted(distance.items(), **key=**lambda **item**: item[1][0])[0]

**del** distance[nearest]

        final\_distance[nearest] **=** weight

*#distance에 아무 역도 남아있지 않을 때 까지 실행한다.*

**while** len(distance) **!=** 0:

*#distance에 들어있는 역들 중 가장 weight 즉 거리가 짧은 역을 distance에서 제거하고 final\_distance에 넣음*

            nearest, weight **=** sorted(distance.items(), **key=**lambda **item**: item[1][0])[0]

**del** distance[nearest]

            final\_distance[nearest] **=** weight

*#distance에서 제거한 역의 인접 역과 weight를 저장하기 위해 cur에 해당 역을 할당하고*

*#시작점부터 그 역까지의 거리를 start\_to\_current\_weight에 할당한다.*

            cur **=** *self*.stations[nearest]

            start\_to\_current\_weight **=** weight[0]

**for** station **in** cur.adjacent\_stations:

                path **=** {**\*\***distance, **\*\***final\_distance}

*#distance와 final\_distance 중 어느 부분에 존재하던 역이라면 더 짧은 거리의 정보로 업데이트한다.*

**if** station **in** path:

                    path1 **=** path[station]

                    path2 **=** [cur.adjacent\_stations[station] **+** start\_to\_current\_weight, cur.name]

**if** path1[0] **>** path2[0]:

                        distance[station] **=** path2

*#아직 distance와 final\_distance어디에도 없던 역이면 새로 추가한다.*

**else**:

                    distance[station] **=** [cur.adjacent\_stations[station] **+** start\_to\_current\_weight, cur.name]

*self*.path **=** final\_distance

*self*.printing\_path(start, end)

*#구한 결과 path를 출력하고 None으로 다시 초기화한다.*

**def** printing\_path(*self*, **start**, **end**):

        path **=** [end]

        cur **=** *self*.path[end]

        distance **=** cur[0]

**while** cur[1] **!=** start:

            path.append(cur[1])

            cur **=** *self*.path[cur[1]]

        path.append(start)

**for** i **in** range(len(path) **-** 1, **-**1, **-**1):

            print(path[i], **end=**" - ")

        print(" (%dkm)\n\n" **%** distance)

*self*.path **=** None

과제 1번을 수행하기 위해서 선언한 클래스는 다음과 같다.

|  |  |
| --- | --- |
| **클래스** | **내용** |
| Station | Tree의 levelorder 메서드에서 사용할 자료 구조 |
| Map | 식의 피연산자 혹은 연산자를 저장할 노드 |

Station는 각 역의 정보를 담기 위해 선언하였다. Station의 name은 역의 이름을 저장하도록 했고, adjacent\_stations에는 인접한 vertex 즉 역의 이름과 거리를 딕셔너리 형태로 저장하도록 했다. 딕셔너리 형태이기 때문에 별도의 Sorting은 하지 않았다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| name | 데이터들이 저장되는 장소 |
| adjacent\_stations | 큐의 items 변수에 데이터의 존재 여부를 bool 값으로 반환하는 메서드 |

Map 클래스는 stations와 path라는 인스턴스 변수를 가진다. stations는 해당 인스턴스 안에 기록되는 역들의 정보를 담고 있는 딕셔너리로 init을 하면서 전달받는다. Path는 get\_path 메서드를 이용하여 발견한 최단 경로를 저장하는 임시 공간으로 사용하는 변수이다.

Map 클래스는 init 되면서 stations 변수를 업데이트함과 동시에 connect\_stations 메서드를 통해서 역과 역 사이의 연결 여부와 거리 정보를 저장한다. 연결 정보는 각 Station 인스턴스에서 adjacent\_stations 변수를 수정하면서 저장했다. 저장 형태는 {“인접 역”:”거리”} 이다.

get\_available\_station은 Map 인스턴스에 연결되어 있는 모든 역을 표시해주는 메서드이다. 이 메서드는 BFS 알고리즘을 이용하여 너비 우선 탐색을 통해 시작점으로부터 연결된 모든 역을 출력하도록 정의했다. BFS 알고리즘을 사용할 때 collections를 import 하여 데크를 선언하여 사용했다. BFS를 통해 모든 역을 돌아다니면서 연결되어 있는 모든 역들의 이름을 station\_list에 저장한 뒤에 station\_list를 정렬하여 사용자에게 출력하였다.

get\_path 메서드는 사용자로부터 입력 받는 출발지와 목적지 정보를 통해 최단 경로를 찾아주는 메서드이다. 이 메서드는 다익스트라 알고리즘을 이용하여 구현했다. 우선 distance와 final\_distance 총 2가지 딕셔너리를 선언했다. distance 딕셔너리에는 아직 경로 확정이 되지 않은 역들을 모아두었고, final\_distance에는 경로가 확정된 역들을 모아두었다. 사용자가 출발지로 설정한 역에 대해 인접해 있는 역들을 구해 final\_distance와 distance 딕셔너리에 적절하게 넣어준다. 이후에 distance 딕셔너리 안에 남아있는 역이 없을 때까지 역과 역 사이의 거리를 업데이트하여 최단 경로를 찾게 된다.

구현한 알고리즘은 다음과 같은 방식으로 동작된다. distance안에 있는 역 정보들 중 출발지에서 가장 짧은 거리에 있는 역을 cur에 할당한 뒤, distance에서 제거한다. 그러고 cur의 adjacent\_stations들을 distance에 넣는다. 이때, 이미 distance 혹은 final\_distance에 존재하는 역이 입력된다면 기존의 등록된 경로와 새로 추가되는 경로의 거리를 비교하여 더 짧은 거리의 정보로 업데이트 한다. 마지막으로 cur을 final\_distance 딕셔너리에 저장하면서 하나의 trial을 마친다. 이 동작을 distance안에 역 정보가 다 사라질 때, 즉 모든 경로에 대한 탐색이 끝날 때까지 진행한 뒤 결과를 path 인스턴스 변수에 할당한다.

printing\_path 메서드를 통해서 사용자의 목적지부터 출발지까지 경로를 거꾸로 추적하여 최종 경로와 총 거리를 포맷에 맞게 출력했다. 마지막으로 path 인스턴스 변수는 None으로 초기화하여 다음에 있을 입력에 대비하도록 했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| stations | Map안에서 사용할 모든 역의 정보를 담기 위한 딕셔너리 |
| path | 최종으로 최단 거리를 출력하기 전에 임시로 저장하는 공간 |
| get\_available\_station() | 사용자에게 선택 가능한 역을 보여주는 메서드 |
| connect\_stations() | Map을 init 할 때 역과 역 사이의 연결 유무와 거리 정보를 등록하는 메서드 |
| get\_path() | 사용자의 입력으로 받은 출발역과 도착역 사이의 최단 경로를 찾기 위한 메서드 |
| printing\_path() | 경로를 표시해주는 메서드 |

* 1. **실행**

*#모든 역을 Dictionary에 선언했다.*

all\_stations **=** {

    "중앙": Station("중앙"), "금정" : Station("금정"), "인덕원" : Station("인덕원"), "과천": Station("과천"), "사당" : Station("사당"),

    "이수": Station("이수"), "동작" : Station("동작"), "동대문" : Station("동대문"), "당고개" : Station("당고개"), "성균관대": Station("성균관대"),

    "문래": Station("문래"), "안양" : Station("안양"), "가산디지털단지" : Station("가산디지털단지"), "신도림": Station("신도림"),

    "노량진": Station("노량진"), "가양": Station("가양"), "서초": Station("서초"), "서울대입구": Station("서울대입구"),

    "광명사거리": Station("광명사거리"), "대림": Station("대림"), "상도": Station("상도"), "고속터미널": Station("고속터미널"),

    "흑석": Station("흑석"), "당산": Station("당산"),

}

*#역과 역 사이의 Edge와 Weight를 선언했다.*

all\_directions **=** [

    ['가양', '당산', 6], ['당산', '노량진', 4], ['노량진', '흑석', 2], ['흑석', '동작', 1], ['동작', '고속터미널', 3],

    ['고속터미널', '이수', 2], ['이수', '상도', 3], ['상도', '대림', 5], ['대림', '가산디지털단지', 2], ['가산디지털단지', '광명사거리', 2],

    ['당산', '문래', 2], ['문래', '신도림', 1], ['신도림', '대림', 1], ['대림', '서울대입구', 5], ['서울대입구', '사당', 2],

    ['사당', '서초', 2], ['성균관대', '금정', 4], ['금정', '안양', 2], ['안양', '가산디지털단지', 5], ['가산디지털단지', '신도림', 2],

    ['신도림', '노량진', 4], ['노량진', '동대문', 8], ['중앙', '금정', 7], ['금정', '인덕원', 4], ['인덕원', '과천', 2],

    ['과천', '사당', 5], ['사당', '이수', 1], ['이수','동작', 1], ['동작', "동대문", 10], ['동대문', "당고개", 9],

]

*#Seoul\_Train 클래스를 all\_stations와 all\_directions를 이용해서 initialize 한다.*

Seoul\_Train **=** Map(all\_stations, all\_directions)

**while** True:

    print("갈 수 있는 역의 종류입니다.")

*#갈 수 있는 역을 BFS로 표시한다.*

    Seoul\_Train.get\_available\_station()

*#사용자에게 역을 입력받는다.*

    print("\n출발역과 도착역을 입력한다.")

    start **=** input("출발역 : ")

    end **=** input("도착역 : ")

*#사용자의 입력에 따라 경로를 구하여 표시하거나 다시 입력을 받도록 한다.*

**if** start **in** Seoul\_Train.stations **and** end **in** Seoul\_Train.stations:

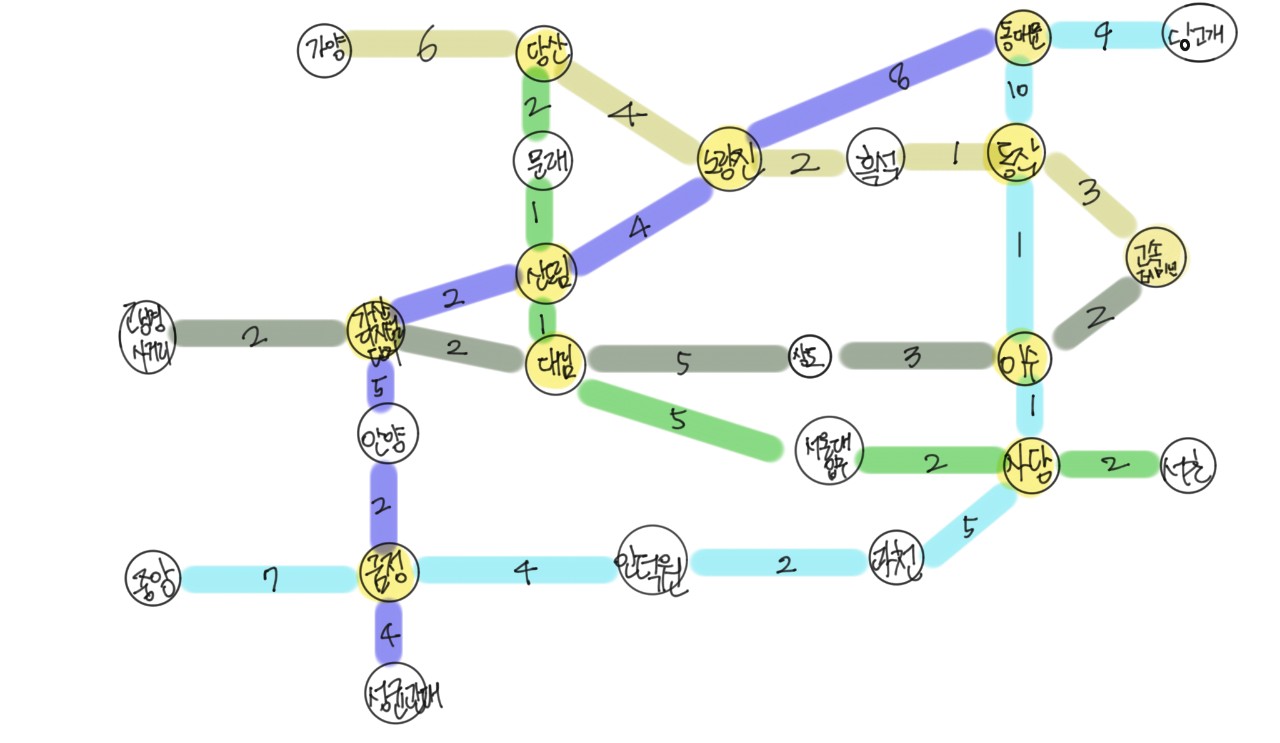
        print("경로와 거리를 표시한다.")

        Seoul\_Train.get\_path(start, end)

**else**:

        print("존재하는 역을 입력해주세요\n\n")

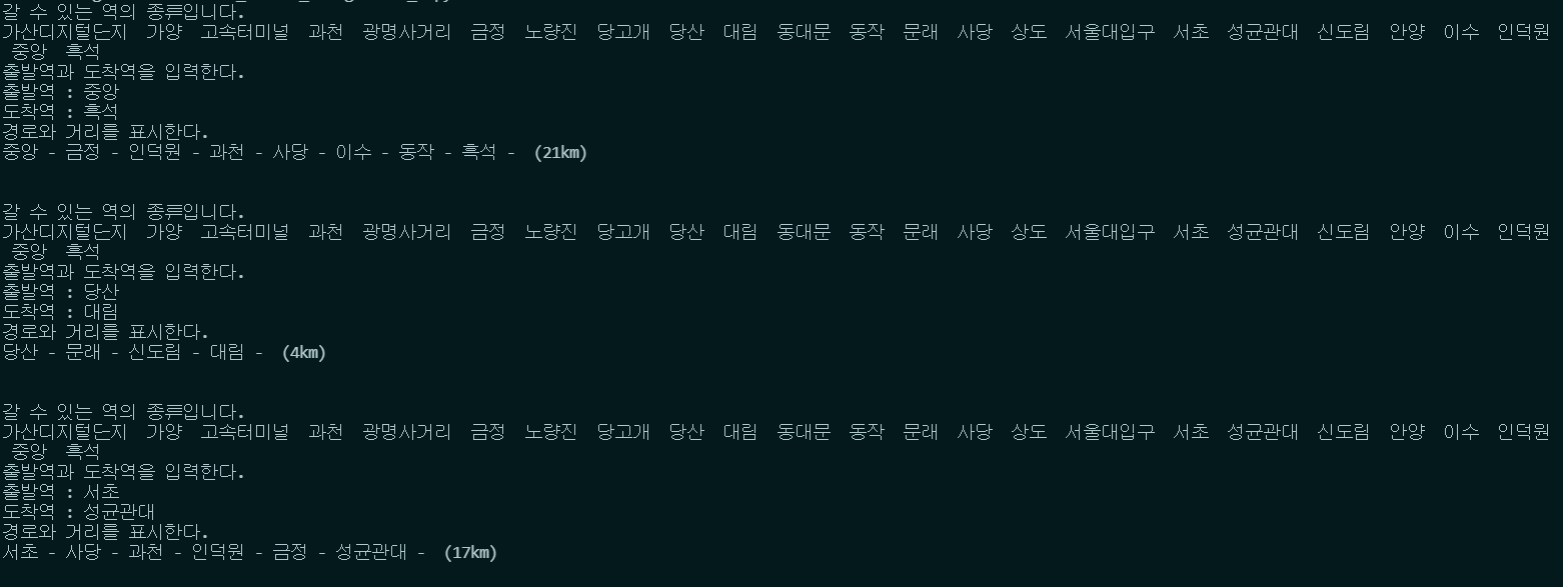
과제 1번을 위해 설정한 지하철 노선도의 구성은 다음과 같다. 24개의 역과 11개의 환승역을 사용했다.

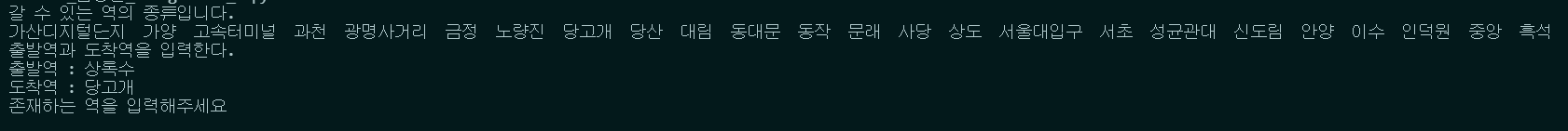


우선 Station 클래스를 이용하여 모든 역을 all\_stations 딕셔너리에 {“이름”:”Station인스턴스”} 형태로 저장했다. 그리고 각 역들의 연결 여부와 사이의 거리 정보를 리스트 형식으로 저장했다. 그리고 Seoul\_Train이라는 Map 클래스의 인스턴스를 만들어 지하철의 노선을 저장했다.

사용자의 입력을 계속 받기 위해 While True: 구문을 사용하여 무한 루프로 동작하게 했다. 우선 사용자에게 갈 수 있는 역의 종류를 출력해준다. 사용자는 출력된 역들을 보고 출발역과 도착역을 입력한다. 만약 사용자의 입력이 기존의 정의했던 역 딕셔너리에 없다면 존재하지 않는 역이라고 알려주고 다시 입력을 받도록 했다. 사용자의 입력이 역 딕셔너리에 존재한다면 경로와 거리를 포맷에 맞추어 출력했다.

* 1. **결과**





# **사전 탐색 트리 만들기**

* 1. **클래스와 메서드 정의**

**import** random

class Node :

**def** \_\_init\_\_(*self*, **item**, **left=**None, **right=**None) :

*self*.item **=** item

*self*.left **=** left

*self*.right **=** right

**def** \_\_str\_\_(*self*):

**return** *self*.item

*#트리*

class TreeA :

**def** \_\_init\_\_(*self*, **word\_list**):

*self*.root **=** *self*.initialize(word\_list)

*#중위 순위를 하면서 트리의 모든 단어를 정렬된 상태로 출력해주는 메서드*

*#디버깅을 위해서 구현*

    @classmethod

**def** print(*cls*, **n**):

**if** n **!=** None :

*cls*.print(n.left)

            print(n.item)

*cls*.print(n.right)

*#입력받은 item을 트리에서 찾아서 돌려주는 탐색 메서드*

**def** search(*self*, **item**):

**if** *self*.root **is** None:

**return** None

**else**:

**return** *self*.\_\_search\_node(*self*.root, item, **level** **=** 0)

*#search메서드에서 단어를 탐색하는데 사용하는 메서드*

**def** \_\_search\_node(*self*, **cur**, **item**, **level**):

**if** cur **is** None:

**return** None, None

**if** cur.item[0] **==** item:

**return** cur, level

**else**:

**if** cur.item[0] **>=** item:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.left, item, level)

**else**:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.right, item, level)

*#입력받은 word\_list를 이진 트리의 형태로 initialize한다.*

**def** initialize(*self*, **word\_list**):

        data\_length **=** len(word\_list)

**if** data\_length **<** 4:

**if** data\_length **<=** 0:

**return** None

**elif** data\_length **<=** 1:

                root **=** Node(word\_list[0])

**return** root

**elif** data\_length **<=** 3:

                root **=** Node(word\_list[1])

                root.left **=** Node(word\_list[0])

**if** data\_length **==** 3:

                    root.right **=** Node(word\_list[2])

**return** root

**else**:

            root\_num **=** len(word\_list) **//** 2

            root **=** Node(word\_list[root\_num])

            root.left **=** *self*.initialize(word\_list[:root\_num])

            root.right **=** *self*.initialize(word\_list[root\_num **+** 1:])

**return** root

*#트리의 높이를 구하는 메서드*

**def** calc\_height(*self*, **n=**None):

**if** n **is** None: **return** 0

**return** 1 **+** max(*self*.calc\_height(n.left), *self*.calc\_height(n.right))

*# 총 몇개의 노드가 있는지를 구하는 메서드*

**def** get\_total\_count(*self*, **n**):

**if** n **is** None: **return** 0

**return** 1 **+** *self*.get\_total\_count(n.left) **+** *self*.get\_total\_count(n.right)

과제 2번을 수행하기 위해서 선언한 클래스는 다음과 같다. **과제 5에서 이미 AVL 트리로 해당 내용을 작성했다.** 이에 과제 6에서는 원래 과제 5에서 해야 했던 정렬 후 트리를 구성하는 방식으로 사전 탐색 트리를 구현했다.

|  |  |
| --- | --- |
| **클래스** | **내용** |
| Node | 사전의 내용을 담기 위한 노드 |
| TreeA | 사전을 정렬된 상태로 구성한 트리 |

Node 클래스는 영어 단어와 뜻을 저장하기 위해서 구현했다. 각 Node는 정보를 저장할 item 변수와 왼쪽 child를 가리키는 left, 오른쪽 child를 가리키는 right 변수를 가지고 있다. 만약 Node 클래스를 이용해서 인스턴스를 생성할 때 파라미터로 아무것도 전달되지 않는다면 left와 right값은 None 값을 가지게 했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| item | 식의 피 연산자 혹은 연산자를 저장하는 변수 |
| left | 해당 Node의 left child를 가리키는 변수 |
| right | 해당 Node의 right child를 가리키는 변수 |

TreeA 클래스는 tree의 최 상단에 위치한 root node를 root라는 변수로 기억하고 있다. 이후에 TreeA 인스턴스를 이용하여 연산을 진행할 때 해당 root를 통해 root node에 접근해 필요한 기능을 수행할 수 있다. 또한 과제를 구현하면서 디버깅을 위해 tree를 중위 순회 하면서 item을 출력하는 print 메서드를 구현했다. tree를 중위 순회 하면서 item을 출력하면 자동으로 정렬된 결과를 얻을 수 있었다.

TreeA는 initialize메서드를 통해서 단어 정보를 tree형태로 구성하게 된다. Initialize메서드는 word\_list를 매개변수로 전달받는데 이때 전달 받는 리스트는 정렬되어 있다. 따라서 리스트의 정 가운데를 루트 노드로 한 뒤, 루트 노드를 기준으로 왼쪽 부분의 정 가운데를 left child, 오른쪽 부분의 정 가운데를 right child로 하여 서브 트리를 만들어 나가면서 사전 탐색 트리를 구성하도록 했다.

search와 \_\_search\_node를 통해 파라미터로 전달 되는 단어를 tree에서 탐색하여 해당 단어가 저장되어 있는 node를 반환 하도록 했다. 이때 해당 node의 level 역시 함께 반환한다.

tree의 높이를 구하는 메서드인 calc\_height 메서드와 총 몇 개의 node가 tree에 연결되어 있는지를 구하는 get\_total\_count 메서드를 구현했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| root | 사전의 root node를 pointing 하는 변수 |
| print() | Tree로 구성된 사전을 중위 순회하며 출력하는 메서드 |
| search() | 입력 받은 item을 트리에서 찾아서 돌려주는 탐색 메서드 |
| \_\_search\_node() | Search 메서드에서 단어를 탐색하는데 사용하는 메서드 |
| initialize | 전달받은 정렬된 사전 내용을 트리의 형태로 만드는 메서드 |
| calc\_height() | 트리의 높이를 구하는 메서드 |
| get\_total\_count() | 총 몇 개의 노드가 있는지 구하는 메서드 |

* 1. **실행**

*#결과를 출력하는 함수*

**def** print\_result(**tree**, **search\_list**):

    print("사전 파일을 모두 읽었습니다. %d개의 단어가 있습니다." **%** tree.get\_total\_count(tree.root))

    print("트리의 전체 높이는 %d 입니다." **%** tree.calc\_height(tree.root))

    print("랜덤하게 선택된 단어 %d개 : " **%** len(search\_list), **end=**"")

**for** word **in** search\_list:

        print(word[0], **end=**" ")

    print()

**for** word **in** search\_list:

        result, level **=** tree.search(word[0])

        print("%s %s (레벨 %d)" **%** (result.item[0], result.item[1], level))

*#몇 개의 단어를 랜덤하게 뽑을지 지정하는 변수*

DATA\_SIZE **=** 10

*#파일을 열어서 단어를 읽어온다.*

**with** open('./randdict\_utf8.TXT', 'r', **encoding=**"utf-8") **as** f:

*#word\_list는 단어를 랜덤하게 선택하기 위해 저장해두는 리스트*

    word\_list **=** list()

**for** line **in** f:

        line **=** line.strip().split(":")

        line **=** list(map(lambda **x**: x.strip(), line))

*#형식에 맞지 않으면 tree와 전체 word\_list에 추가하지 않는다.*

**if** line[1] **==** "":

**continue**

**else**:

            word\_list.append(line)

word\_list **=** sorted(word\_list, **key=**lambda **word**: word[0])

treeA **=** TreeA(word\_list)

*#랜덤으로 DATA\_SIZE만큼의 단어를 선택함*

search\_list **=** random.sample(word\_list, DATA\_SIZE)

*#treeA의 결과 출력*

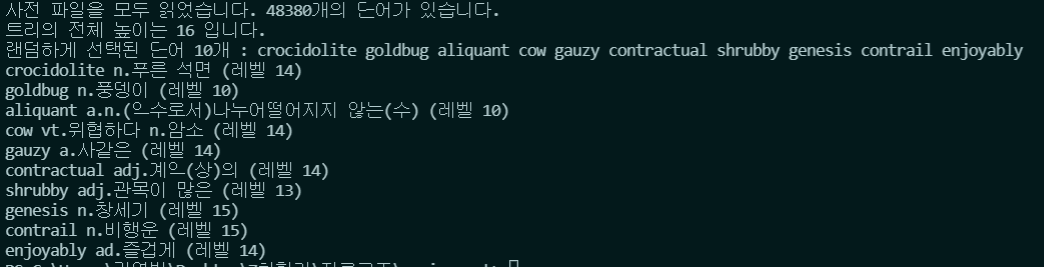
print\_result(treeA, search\_list)

우선 연산의 결과를 지정된 format으로 출력해주는 print\_result 함수를 정의했다. 해당 함수는 tree와 랜덤으로 선택된 단어 리스트를 받는다. 이후에 tree에 저장된 단어 수, tree의 높이, 랜덤으로 선택된 단어의 수와 내용을 출력해준다. 이후에 Tree 클래스의 인스턴스 메서드 search를 이용하여 해당 단어가 저장된 node와 해당 node가 몇 level에 존재하는지를 돌려받은 뒤 단어와 단어의 뜻, 해당 단어가 저장된 node의 위치를 출력하도록 했다.

과제 설명에는 랜덤으로 10개의 단어를 선택하라고 했지만 이후에 보다 유연하게 선택하는 단어의 수를 변경할 수 있도록 DATA\_SIZE에 10을 할당한 뒤에 해당 변수를 이용해서 단어를 랜덤하게 선택하도록 했다.

제공된 TXT파일의 정보를 사용하기 위해서 randdict\_utf8.TXT 파일을 읽기 모드로 open한 뒤에, 각 line에서 단어와 의미를 분리했다. 이때 뜻이 형식에 맞지 않는 단어에 대해서는 tree에 추가하지 않았다. treeA는 TreeA 클래스의 인스턴스를 저장하는 인스턴스 변수이다. 마지막으로 word\_list 사전의 내용을 트리로 구성하기 전에 정렬을 하는 역할과 랜덤하게 단어를 선택하는 역할을 한다. 우선 파이썬 내장 정렬을 이용해서 word\_list에 담겨있는 단어들을 영어 단어의 순서에 맞게 정렬한다. 이후에 random.sample 메서드를 이용하여 DATA\_SIZE만큼의 단어를 임의로 선택한 뒤에 search\_list에 할당한 뒤, 각 treeA를 search\_list와 함께 위에서 정의한 print\_result 함수에 전달하면서 과제에서 요구하는 결과를 출력하도록 했다.

* 1. **결과**

****