자료구조

과제 7. 트리

과목명 : 자료구조
담당교수 : 김승태
소속학과 : 응용통계학과
학번 : 20193011
이름 : 유승희

1. 사전 탐색 트리 만들기

1.1 사전 만들기

데이터 불러오기

제공된 사전 파일 randdict.TXT 파일을 읽기모드로 한 줄씩 불러와 랜덤 사전 dictionary 리스트에 저장해주었다. 단어들을 불러올 때 편의를 위해 줄바꿈은 제거하고 ':'를 기준으로 나누어 저장했다.

- 코드

```
dictionary = []
with open('randdict.TXT', 'r') as file:
    for text in file:
        dictionary.append(text.strip('\n').split(' : '))
```

이진 탐색 트리 정의하기

이진 탐색 트리 형태로 사전을 만들기 위해 트리 구조체를 다음과 같이 정의하였다.

<이진 탐색 트리>
노드 클래스 : 각 노드는 데이터(영어단어와 뜻)를 포함한 data와 왼쪽 자식 노드를 가리키는 self.left , 오른쪽 자식 노드를 가리키는 self.right 속성을 생성했다. (data[0]은 영단어, data[1]은 뜻을 의미한다.)

이진 탐색 트리 클래스 : 노드들을 감싸는 전체 이진 탐색 트리 클래스로 만든다. 이 클래스를 실행하면 트리의 첫 노드를 가리키는 루트 변수를 포함한 아직 노드가 없는 연결리스트를 생성한다. 따라서 헤드는 None을 가리킨다. 이진 탐색 트리 클래스에 다음과 같은 기능(함수)들로 추가했다.

1. 이진 탐색 트리의 삽입
- insert(self, item) : 추가할 단어 item (item[0] 영단어, item[1] 뜻으로 포함된 리스트) 입력한다. 트리가 비워져 있는 상태면 (self.root 가 None이면) 입력된 item을 루트 노드로 설정해준다. 그렇지 않으면 self.__insert_node(self.root, item)함수를 이용해 노드를 추가한다.
- __insert_node(self, cur, item) : 현재 위치의 노드 cur, 추가할 단어 item을 입력받는다. 비교를 위해 현재 노드의 영단어와 뜻을 data 변수에 저장해주고, 소문자로 비교해주기 위해 data[0].lower()를 transform_data_key에 저장하고 item[0].lower()를 transform_item_key에 저장한다.
  transform_data_key(현재 노드 키)보다 transform_item_key (입력할 데이터 키)가 작거나 같은 경우 : 현재 노드의 왼쪽 자식 노드가 None이 아니면 왼쪽 자식

노드로 내려가 재귀적으로 알맞은 위치를 찾아 삽입한다. 왼쪽 자식 노드가 None이면 왼쪽 자식에 item 노드를 생성해준다.

transform_data_key(현재 노드 키)보다 transform_item_key (입력할 데이터 키)가 큰 : 현재 노드의 오른쪽 자식 노드가 None이 아니면 오른쪽 자식 노드로 내려가 재귀적으로 알맞은 위치를 찾아 삽입한다. 오른쪽 자식 노드가 None이면 오른쪽 자식에 item 노드를 생성해준다.

2. 이진 탐색 트리의 탐색

- search(self, item) : 이진 탐색 트리의 탐색 기능이다. 탐색을 위한 키 (item)을 입력했을 때 노드가 없는 트리면 즉 트리의 루트 노드 데이터가 None이면 None을 그렇지 않으면 self.__search_node(self.root, item, level)함수를 이용해 노드를 탐색해 반환한다. 탐색한 노드의 레벨을 계산하기 위한 level변수를 1로 설정해준다.
- __search_node(self, cur, item, level) : 현재 노드를 의미하는 cur, 탐색을 위한 단어 item, 노드의 레벨 level을 입력한다. 위의 search 함수에서 self.root를 입력하였으므로 현재 노드(cur)는 루트 노드로부터 시작하여 탐색을 시작한다. 현재 노드가 None이면 None을 반환한다.

그렇지 않으면 비교를 위해 현재 노드의 data의 영단어를 data_key로 저장해준다.(data_key = cur.data[0])

data_key 와 찾고자 하는 단어 item과 일치하면 현재 해당하는 노드와 레벨을 반환한다.

data_key 보다 찾고자 하는 단어 item이 문자열 비교시 작거나 같으면 왼쪽 자식 노드로 내려가므로 레벨을 1 증가 시키고 현재 노드의 왼쪽 자식 노드에서 다시 탐색을 재귀적으로 실시한다.

반대로 data_key 보다 찾고자 하는 단어 item이 크면 동일하게 레벨을 1 증가시키면서 현재 노드의 오른쪽 서브 트리에서 탐색을 실시한다.

그 외 사전 트리를 만들기 위해 사용할 함수들을 다음과 같이 정의했다.

- inorder(n) : 이진 탐색 트리를 만든 후 오름차순으로 출력해주는 함수이다. 중위 순회 방법을 이용했다.
- calc_height(n) : 트리의 높이를 계산해주는 함수로 서브트리의 반환값 중 가장 큰 값과 노드 자신의 높이 1을 더한 값을 반환해준다.
- count_node(n) : 트리의 노드의 개수를 세주는 함수로 왼쪽 서브트리의 노드 수, 오른쪽 서브트리의 노드 수, 자기 자신을 더한 값을 반환한다.

-코드

```
class Node :
    def __init__(self, data, left=None, right = None):
        self.data = data
        self.left = left
        self.right = right
class BinaryTree:
    def __init__(self):
        self.root = None

    ## 이진 탐색 트리의 탐색

    def search(self, item):
        if self.root.data is None:
            return None
        else:
            level = 1
            return self.__search_node(self.root, item , level)

    def __search_node(self, cur, item, level):
        if cur is None:
            return None

        data = cur.data
        data_key = data[0]

        if data_key == item:
            return cur, level
        else:
            if data_key >= item:
                level += 1
                return self.__search_node(cur.left , item, level)
            else:
                level += 1
                return self.__search_node(cur.right, item, level)

        return None
## 이진 탐색 트리의 삽입

def insert(self, item):
    if self.root is None:
        self.root = Node(item)
    else:
        self.__insert_node(self.root , item)

def __insert_node(self, cur, item):
    data = cur.data
    transform_data_key = data[0].lower()
    transform_item_key = item[0].lower()

    if transform_data_key >= transform_item_key :
        if cur.left is not None:
            self.__insert_node(cur.left, item)
        else:
            cur.left = Node(item)
    else:
        if cur.right is not None:
            self.__insert_node(cur.right, item)
        else:
            cur.right = Node(item)
# 오름차순 출력
def inorder(n):
    if n!= None:
        inorder(n.left)
        print(n.data)
        inorder(n.right)

#트리 높이 세기
def calc_height(n):
    if n is None: return 0
    return 1+max(calc_height(n.left),calc_height(n.right))

#트리 노드 개수 세기
def count_node(n):
    if n is None: return 0
    return 1+count_node(n.left) + count_node(n.right)
```

사전 탐색 트리 만들기

dict_tree 라는 이진 탐색 트리를 생성한다. dictionary의 요소([‘영단어’, ‘뜻’]을 반복문을 돌며 dict_tree에 삽입한다. 사전 탐색 트리가 완료 되면, 사전 파일을 모두 읽었다는 메시지와 함께 만들어진 트리의 높이와 노드 수를 출력한다.

-코드

```
## 사전 탐색 트리 만들기
dict_tree = BinaryTree()

for word in dictionary:
    dict_tree.insert(word)

print(f"사전 파일을 모두 읽었습니다. {len(dictionary)} 개의 단어가 있습니다.
\nA 트리의 전체 높이는 {calc_height(dict_tree.root)}입니다. A 트리의 노드 수는 {count_node(dict_tree.root)}개 입니다. ")
```

결과:

```
사전 파일을 모두 읽었습니다. 48406 개의 단어가 있습니다.
A 트리의 전체 높이는 37입니다. A 트리의 노드 수는 48406개 입니다.
```

단어 검색하기 (개선 후 방법)

무한 반복문을 돌며 검색할 단어를 입력받고 입력 받은 단어가 사전 트리 내에 존재하면 어느 노드인지 탐색해 node와 그 노드의 level 값을 돌려받아 node의 데이터 부분에 뜻을 의미하는 node.data[1] 과 level을 출력한다. 만약 검색한 단어가 존재하지 않으면 ‘사전에 단어가 없습니다.’를 출력하며 다음 검색으로 넘어간다.

- 코드

```
while 1:
    word = input('단어를 입력하세요.: ')
    if dict_tree.search(word) is not None:
        node, level = dict_tree.search(word)
        print(f"{node.data[1]} (레벨 {level})")
    else:
        print("사전에 단어가 없습니다.")
```

결과 :

```
단어를 입력하세요.: add
vt.추가하다 (레벨 23)
단어를 입력하세요.: apple
n.사과 (레벨 24)
단어를 입력하세요.: seashell
n.조개 (레벨 28)
단어를 입력하세요.: zymurgy
n.양조학 (레벨 8)
단어를 입력하세요.: dog
n.개 (레벨 19)
단어를 입력하세요.:
```

2. 효과적인 사전 탐색 트리 만들기

2.1 최적의 사전 트리 만들기

사전 파일 정렬하기

높이가 낮은 트리를 만들어 단어를 빨리 찾을 수 있게 하기 위하 사전 파일을 정렬하여 저장해준다. 정렬을 하기 위해 앞서 정의한 중위 순회의 개념을 이용한 sort_inorder(n) 함수를 정의해 준다. 정렬된 사전을 저장하기 위한 빈 리스트 sorted_dictionary를 만들어 주고 재귀함수를 통해 왼쪽 서브트리, 루트, 오른쪽 서브트리를 방문하며 데이터를 append해준다.

오름차순으로 정렬되는 것을 확인하기 위해 inorder 함수 출력 결과의 앞 10개, 중간 10개, 끝 10개를 첨부한다.

앞 10개

```
['abaca', 'n.마닐라삼(필리핀 주산)']
['abaci', 'n.abcus의 복수형']
['aback', 'ad.뒤로']
['abacus', 'n.계산기']
['abaft', 'ad.(배의)고물에']
['abalone', 'abalone']
['abandon', 'vt.버리다']
['abandoned', 'a.버림받은']
['abandoner', 'n.유기자']
['abandonment', 'n.포기']
```

중간 10개

```
['mythology', 'n.신화학']
['mythopeic', 'adj.신화를 만드는']
['mythopoeic', 'adj.신화를 만드는']
['mythos', 'n.신화']
['myxedema', 'n.점액 수종']
['myxoedema', 'n.점액 수종']
['myxomatosis', 'n.(다발성)점액종증']
['nab', 'vt.잡아채다']
['nabob', 'n.태수']
['nacelle', 'n.항공기의 엔진 덮개']
```

끝 10개

```
['zymogen', 'n.치모겐']
['zymogenic', 'n.발효를 일으키는']
['zymologist', 'n.발효학자']
['zymology', 'n.발효학']
['zymometer', 'n.발효도 측정계']
['zymosimeter', 'n.발효도 측정계']
['zymosis', 'n.발효']
['zymotechnics', 'n.양조법']
['zymotic', 'adj.발효성의']
['zymurgy', 'n.양조학']
```

- 코드

```
# 최소 높이 사전 탐색 트리 만들기

dict_tree = BinaryTree()

for word in dictionary:
    dict_tree.insert(word)

## A 트리를 이용해 사전 정렬하기
sorted_dictionary = []
def sort_inorder(n):
    if n!= None:
        sort_inorder(n.left)
        sorted_dictionary.append(n.data)
        sort_inorder(n.right)

sort_inorder(dict_tree.root)
```

정렬된 사전으로 최소 높이 트리 만들기

최적의 트리를 저장할 dict_comp_tree라는 이진 트리를 생성해준다. 앞서 정의한 데이터 삽입 규칙과 조금 달리하여 정렬된 사전의 단어들을 삽입하는 함수를 만들었다.

-insert_comptree(dict_sample) : dict_sample 이라는 사전 리스트를 입력받는다. 이 사전 리스트의 길이가 1이면 만들어준 이진트리에 A 트리의 삽입 함수를 이용해 dict_comp_tree에 데이터를 넣어준다.
길이가 1보다 길면 리스트의 중간 단어를 트리에 삽입후 중간 단어를 기준으로 작은쪽 큰쪽으로 나누어 다시 insert_comptree를 진행하여 dict_comp_tree에 삽입한다.

위 함수를 이용해 dict_comp_tree를 만들어 주고 사전 트리를 완성하면 사전 파일을 모두 읽었다는 완료 메시지와 함께 최적의 트리의 높이와 노드의 수를 출력한다.

- 코드

```
## 정렬된 사전으로 최소 높이 트리 만드는 함수
def insert_comptree(dict_sample):
    if len(dict_sample) == 1:
        dict_comp_tree.insert(dict_sample[0])

    elif len(dict_sample) > 1:
        idx = len(dict_sample)//2
        dict_comp_tree.insert(dict_sample[idx])

        insert_comptree(dict_sample[:idx])
        insert_comptree(dict_sample[idx+1:])

dict_comp_tree = BinaryTree()

insert_comptree(sorted_dictionary)

print(f"사전 파일을 모두 읽었습니다. {len(dictionary)} 개의 단어가 있습니다.
\nB 트리의 전체 높이는 {calc_height(dict_comp_tree.root)}입니다. B 트리의 노드 수는 {count_node(dict_comp_tree.root)}개 입니다. ")
```

결과 :

```
사전 파일을 모두 읽었습니다. 48406 개의 단어가 있습니다.
B 트리의 전체 높이는 16입니다. B 트리의 노드 수는 48406개 입니다.
```

단어 검색하기

A 트리의 방법과 동일하게 단어를 검색한 후 사전에 있으면 node와 level을 돌려받아 단어의 뜻을 의미하는 node.data[1]과 해당 노드의 레벨인 level을 출력한다. 검색한 단어가 없으면 사전에 단어가 없음을 출력하고 다시 입력을 받는다.

-코드

```
while 1:
    word = input('단어를 입력하세요.: ')
    if dict_comp_tree.search(word) is not None:
        node, level = dict_comp_tree.search(word)
        print(f"{node.data[1]} (레벨 {level})")
    else:
        print("사전에 단어가 없습니다.")
```

결과 :

```
단어를 입력하세요.: add
vt.추가하다 (레벨 16)
단어를 입력하세요.: apple
n.사과 (레벨 16)
단어를 입력하세요.: seashell
n.조개 (레벨 15)
단어를 입력하세요.: zymurgy
n.양조학 (레벨 15)
단어를 입력하세요.: dog
n.개 (레벨 16)
단어를 입력하세요.:
```

최종 결과

- A 사전 트리의 높이와 노드 수

```
사전 파일을 모두 읽었습니다. 48406 개의 단어가 있습니다.
A 트리의 전체 높이는 37입니다. A 트리의 노드 수는 48406개 입니다.
```

- B 사전 트리의 높이와 노드 수

```
사전 파일을 모두 읽었습니다. 48406 개의 단어가 있습니다.
B 트리의 전체 높이는 16입니다. B 트리의 노드 수는 48406개 입니다.
```

=> A 트리와 B 트리의 노드 수는 모두 48,606개로 동일하지만 A 트리의 높이는 37, B 트리의 높이는 16으로 B 트리의 높이가 더 낮다.

- A 사전 트리에서 검색

```
단어를 입력하세요.: add
vt.추가하다 (레벨 23)
단어를 입력하세요.: apple
n.사과 (레벨 24)
단어를 입력하세요.: seashell
n.조개 (레벨 28)
단어를 입력하세요.: zymurgy
n.양조학 (레벨 8)
단어를 입력하세요.: dog
n.개 (레벨 19)
단어를 입력하세요.:
```

- B 사전 트리에서 검색

```
단어를 입력하세요.: add
vt.추가하다 (레벨 16)
단어를 입력하세요.: apple
n.사과 (레벨 16)
단어를 입력하세요.: seashell
n.조개 (레벨 15)
단어를 입력하세요.: zymurgy
n.양조학 (레벨 15)
단어를 입력하세요.: dog
n.개 (레벨 16)
단어를 입력하세요.:
```

=> 두 사전에 동일한 단어를 검색했을 때 add, apple, seashell, dog의 단어를 보면 B 트리에서 레벨이 더 낮게 나옴을 확인할 수 있다. zymurgy는 A 트리의 레벨이 더 낮지만 평균적으로 B 트리의 레벨이 더 낮아 단어를 더 빨리 찾을 수 있다.