DATA STRUCTURE(2분반)
TERM PROJECT Report

2010130191
사회학과 강호영

```
m = dict()

def sum():
    res = 0
    for i in m:
        res += m[i]
    return res

def count(word):
    if m.get(word):
        m[word] += 1
    else:
        m[word] = 1
```

<그림 1> Dictionary

기본적인 Data Structure로 Hash function을 지원하는 **Dictionary 자료형**을 사용하였다. Dictionary 자료형은 다른 자료형에 비해 **연산(Querying, Inserting, Deleting)을 빠르게 할 수 있는 장점**이 있다.
Hashing을 사용할 경우, 찾고자 하는 값이 Uniform하다면 Constant Time 에 검색이 가능하다. 뿐만 아니라 Hash Table에 Key 값을 저장하고 Key 값에 해당하는 Value값 들을 Linked List로 연결할 때, Table의 크기를 적절히 조절하면 Constant Time에 검색이 가능하다.

<그림1>에서는 dictionary 자료형 m을 초기화하고, count 함수를 별도로 구현하여 dictionary에 Value 값을 저장하고 그때마다 그 숫자를 counting 하도록 하였다.

<그림2>의 read_file 함수는 Twitter 사용자의 정보와 친구관계 정보, Tweet 단어 정보가 담긴 Txt 형식의 파일을 열어, txt 파일의 각 Line별로 Counting을 하여 전체 사용자의 수(Total User), 전체 친구관계의 수(Total Friendship Records), 전체 Tweet 단어의 수(Total Tweets)을 계산한다.

User.txt 파일과 Word.txt 파일은 4개의 Line을 주기로 반복하기 때문에 모든 Line의 수를 더해 4로 나눈다면 반복되는 정보를 하나로 묶어 각 User별 정보와 Tweet단어를 계산할 수 있다.

Friend.txt. 파일은 3개의 Line을 주기로 반복하기 때문에 모든 Line의 수를 더해 3으로 나누면 반복되는 정보를 하나로 묶어 계산할 수 있다.

```
def read_file():
    print("Reading Data Files...")
    ucount = 0
    u = open('user.txt')
    for line in u:
        line = line[0:-1]
        ucount += 1
    print('Total User: ', ucount / 4)
    fcount = 0
    f = open('friend.txt')
    for line in f:
        line = line[0:-1]
        fcount += 1
    print('Total Friendship Records: ', fcount / 3)
    wcount = 0
    w = open('word.txt')
    for line in w:
        line = line[0:-1]
        wcount += 1
    print('Total Tweets: ', wcount / 4)
```

<그림 2> Read File

```python
def statistic():
    print("Displaying Statistics...")
    f = open('friend.txt')
    for line in f:
        line = line[:-1]
        count(line)
    del m['']
    maximumf = max(m, key=m.get)
    minimumf = min(m, key=m.get)
    print("Average Number of Friends:", sum() / len(m))
    print("Maximum Number of Friends:", maximumf, "=>", m[maximumf])
    print("Minimum Number of Friends:", minimumf, "=>", m[minimumf])
    m.clear()
    for i, l in enumerate(open('word.txt')):
        if i % 4 == 0:
            count(l)
    maximumt = max(m, key=m.get)
    minimumt = min(m, key=m.get)
    print("Average Tweet per User:", sum() / len(m))
    print("Maximum Tweet per User:", maximumt.rstrip(), "=>", m[maximumt])
    print("Minimum Tweet per User:", minimumt.rstrip(), "=>", m[minimumt])
    m.clear()
```

왼쪽 <그림 3>
Statistic

아래쪽 <그림 3-1>
Statistic Result

```
Select Menu : 1
Displaying Statistics...
Average Number of Friends: 19.998588965711868
Maximum Number of Friends: 167059014 => 33
Minimum Number of Friends: 362788891 => 11
Average Tweet per User: 15.82023423169183
Maximum Tweet per User: 460950929 => 207
Minimum Tweet per User: 195291401 => 1
None
```

<그림 3>의 Statistic 은 실제적으로 두 가지 기능을 실행한다. 먼저 **friend.txt**에 있는 각 Line을 읽으면서 Line의 마지막에 있는 공백문자(₩n)를 제외한 value값을 count 함수를 호출하여 미리 만들어 놓은 Dictionary 자료형 m에 저장한다. m에는 Friend의 수를 구하기 위해 필요한 User 정보만 들어있어야 하므로 앞서 Line을 추가할 때 포함된 빈 Line을 제거해준다. 다음으로 m안에 있는 User중에 가장 많이(가장 적게) 등장하는 User와 그 숫자를 구한다. 평균의 경우 각 User가 등장하는 횟수를 모두 더하는 sum 함수를 호출하여 그 합을 구하고, 전체 m의 길이로 나눈다.
다음으로 **word.txt**를 Index와 Line을 구별하도록 enumerate 함수를 통해 호출한다. 해당 파일에서 가장 많이 등장하는 User를 찾아야 하므로 4개씩 반복되는 각 Line중에서 User 정보가 있는 Line인 첫 번째 라인의 Index값을 모듈러 연산을 통해 구하여 해당되는 value값을 Dictionary 자료형 m에 추가한다. 나머지 방식은 앞서 Friend 수를 구하는 방법과 같다.
<그림 3-1>은 Statistic 을 실행한 결과이다. 각 조건에 해당되는 유저의 ID Number와 Friend의 수, Tweet의 수를 각각 보여준다.

[Statistic 구현의 한계점]

Friend의 수를 구하기 위해서는 Friend 관계를 먼저 정의해야 한다. 앞서 작성한 코드에 의하면 Friend 관계를 양방향적인 관계로 고려하였기 때문에 각 User가 Friend.txt 파일에서 등장하는 모든 횟수를 계산하였다. 하지만 Friend 관계를 단방향적인 관계(Following 연결)로 고려한다면,
실제적인 Friend의 수는 위 결과값보다 작아질 것이다.(한 User가 다른 User를 일방향적으로 Following 하는 경우에는 Friend 관계로 보기 힘들기 때문이다.) 이 문제는 5번 기능에서 특정 단어를 언급한 User의 모든 Friend 관계를 구할 때, Following 연결로 수정하여 보완하였다.

○ **Friend 관계(양방향적) => Undirected graph**
○ **Following 연결(일방향적) => Directed graph**

<그림 4> 에서는 가장 많이 Tweet된 Word와 가장 많이 Tweet을 한 User를 찾는 함수를 각각 구현하였다.

mostword 함수는 word.txt 파일을 Index 와 Line으로 enumerate한 후 Word가 위치한 모듈러 Line 들의 value값을 m에 저장한다. m에서 내림차순으로 정렬한 후 Top5를 추출한다.

mostuser 함수도 mostword 함수와 마찬가지로 word.txt 파일에서 가장 많이 등장하는 User Top5를 추출한다.

```
def mostword():
    print("Showing Top 5 Most Tweeted Words...")
    for i, l in enumerate(open('word.txt')):
        if i % 4 == 2:
            count(l)
    top = sorted(m, key=m.get, reverse=True)[:5]
    for i in range(5):
        print(top[i].rstrip(), ":", m[top[i]])
    m.clear()

def mostuser():
    print("Showing Top 5 Most Tweeted Users...")  # Tweet 많이 한 사람
    for i, l in enumerate(open('word.txt')):
        if i % 4 == 0:
            count(l)
    top = sorted(m, key=m.get, reverse=True)[:5]
    for i in range(5):
        print(top[i].rstrip(), ":", m[top[i]])
    m.clear()
```

```
Select Menu : 2
Showing Top 5 Most Tweeted Words...
@ : 6889
. : 3430
_ : 2973
... : 1678
: : 1572
None
```

```
Select Menu : 3
Showing Top 5 Most Tweeted Users...
460950929 : 207
449511651 : 143
130003821 : 140
403292271 : 130
527632800 : 123
None
```

위쪽 <그림 4>Top 5 Finding

왼쪽 <그림 4-1>
Top 5 Finding Results

```
def userfind():
    print("Finding Users who tweeted a word...")
    text = str(input("Word: "))
    lst = []
    buffer = 0
    for i, l in enumerate(open('word.txt')):
        l = l[:-1]
        if i % 4 == 0:
            buffer = l
        if i % 4 == 2 and text == l:
            if buffer not in lst:
                lst.append(buffer)
    return lst

def userfriend():
    print("Finding All People who are friends of the above users...")
    users = userfind()
    d = {}
    buffer = 0
    for i, l in enumerate(open('friend.txt')):
        l = l[:-1]
        if i % 3 == 0:
            if l in users:
                if d.get(l) is None:
                    d[l] = []
                buffer = l
        if i % 3 == 1 and buffer in d:
            d[buffer].append(l)
            buffer = 0
    k = d.keys()
    for e in k:
        print("Above User:", e, "=> Friends:", d.get(e))
```

<그림 5> Finding User/Friend

<그림 5>는 특정 단어를 언급한 User들을 찾는 userfind 함수와 특정 단어를 언급한 User들의 친구들을 찾는 userfriend 함수를 보여준다. **userfind 함수**는 List와 buffer를 이용하여 구현하였다. 우선, 특정 단어를 input 값으로 받고, word.txt 파일을 Index와 Line으로 enumerate 한다. 먼저 User 정보가 담긴 Line에서 User 정보를 buffer에 임시로 저장한 후, Word 정보가 담긴 Line에서 input 값과 일치하는 단어를 발견할 경우 buffer에 저장되어있던 User정보를 List에 저장한다. 여기서 User 정보의 중복을 방지하기 위해 List에 User 정보가 없는 경우에만 추가하도록 하였다.

userfriend 함수는 dictionary 자료형과 List를 이용하였으며, dictionary의 key값에 해당되는 value의 집합을 list로 연결하도록 구현하였다. userfriend 함수는 시작과 동시에 앞서 실행했던 userfind 함수를 호출하여, 그 결과값을 변수 users에 저장한다. userfind 함수가 실행되면서 다시 한번 특정 단어를 input 값으로 받은 후, friend.txt 파일을 열어 Index와 Line으로 enumerate한다. (friend.txt 파일에서 친구관계는 directed graph 라고 가정하고, 첫 번째 Line에 있는 User가 두 번째 Line에 있는 User를 Following 한다고 보았다.) 다음으로 users에 있는 User(특정 단어를 언급한)와 Friend.txt의 첫 번째 Line에 있는 User 정보가 일치하는 경우를 찾아 buffer에 임시로 저장한다. 이렇게 buffer에 저장된 User정보는 dictionary의 key값으로 지정된다. 두 번째 Line에서는 앞 단계의 key값이 dictionary에 저장 되어있는 경우, 두 번째 Line의 User 정보를 그 key값에 해당되는 value로서 list를 만들어 추가한다.

```
Select Menu : 4
Finding Users who tweeted a word...
Word: 0
['218457497', '245760441', '433927133', '367170899', '433322045',
```

<그림 5-1> userfind() Result

<그림 5-2>
userfriend() Result

```
Select Menu : 5
Finding All People who are friends of the above users...
Finding Users who tweeted a word...
Word: 0
Above User: 391196436 => Friends: ['242993374', '365878085', '402534362',
Above User: 473772464 => Friends: ['155834979', '307870701', '364117226',
Above User: 398375611 => Friends: ['198114074', '243651317', '246645572',
Above User: 519999591 => Friends: ['154339140', '215633356', '247248983',
Above User: 402720231 => Friends: ['126472424', '129447797', '185649447',
Above User: 255960324 => Friends: ['156830478', '160972250', '165357213',
Above User: 350203152 => Friends: ['218996272', '243869535', '244009384',
Above User: 112804505 => Friends: ['176421052', '227180659', '250953728',
```

[Finding User / Friend 구현의 기대 및 한계점]

생각보다 빠른 시간 내에 연산을 실행하여 만족스러웠다. 하지만 Find All People who are friend of the above users 의 5번 기능을 사용할 때 4번 기능을 실행한 후 결과값으로 반환된 User List를 5번 기능의 input으로 바로 사용하지 못하고, 다시 한번 Word를 입력해야 한다는 번거로운 문제점을 해결하지 못하였다. 이보다 더 빠르게 구현할 수 있는 다른 방법이 있는지 찾아보고, 4번의 output을 5번의 input 으로 바로 사용할 수 있도록 수정이 필요하다.

```python
def delword():
    print("Deleting Words which is mentioned")
    text = str(input("Word: "))
    lst = []
    for i, l in enumerate(open('word.txt')):
        l = l[:-1]
        if i % 4 == 2 and text != l:
            if l not in lst:
                lst.append(l)
    print("New Word List:", lst)

def deluser():
    print("Deleting Users who mentioned a word")
    text = str(input("Word: "))
    people = []
    for i, l in enumerate(open('word.txt')):
        l = l[:-1]
        if i % 4 == 0:
            if l not in people:
                people.append(l)
    lst = []
    buffer = 0
    for ii, ll in enumerate(open('word.txt')):
        ll = ll[:-1]
        if ii % 4 == 0:
            buffer = ll
        if ii % 4 == 2 and text == ll:
            if buffer not in lst:
                lst.append(buffer)
    for k in lst:
        if k in people:
            people.remove(k)
    print("New User List:", people)
```

<그림 6>
Deleting Word/User

<그림 6>에서는 언급된 Tweet들 중에 특정 Word를 지우는 delword 함수와 특정 Word를 언급한 User 를 삭제하는 deluser 함수를 볼 수 있다. **delword 함수**는 word.txt 파일을 Index와 Line으로 enumerate한 후, input으로 받은 특정한 단어(삭제할 단어)가 Word 정보가 담긴 Line의 값들과 일치하지 않는 경우에만 미리 만들어놓은 List에 중복을 배제하여 추가하였다. **deluser 함수**는 조금 더 복잡한데, 그 이유는 특정한 **Word를 언급한 User들을 먼저 찾아야** 하기 때문이다. deluser 함수에서는 먼저 Word.txt 파일을 열어 모든 User를 List에 임시로 저장해둔다. 다음으로는 특정 Word를 언급한 모든 User들을 찾는 방식과 동일하다. 특정 Word를 언급한 User들의 정보를 찾아 새로운 List에 저장한 후, 앞서 만든 List와 새로운 List를 비교하여 삭제할 단어를 언급한 User들을 삭제해준다.

[Finding User / Friend 구현의 기대 및 한계점]

Deleting 구현의 특징은 word.txt 파일에 있는 모든 정보를 저장한 후 해당되는 정보를 찾아 지우는 방식이 아니라 **처음부터 삭제할 정보를 제외한 정보만 저장하는 방식**으로 구현했다는 점이다. 이런 방식으로 구현을 할 경우, **연산 속도뿐만 아니라 저장으로 인해 낭비되는 메모리를 최소화할 것이라 기대**하였다. 그러나 deluser를 구현하는데 있어서는, 두 개의 List 자료형을 이용하여 서로 비교하는 방식을 통해 삭제하였기 때문에 불필요한 메모리 낭비가 발생하였다. 입력 데이터가 커질 경우에는 모든 User정보를 List에 저장해두는 방식 이외의 다른 방식이 필요할 것이다.

Term Project 소감 및 향후 발전 방향

자료구조형을 처음부터 직접 설계한 것이 아니라, Python에서 제공하는 기존의 함수들을 이용하여 구현을 했기 때문에, 다른 학생들에 비해 조금 더 쉬운 방법을 사용했다고 생각합니다. 하지만 그럼에도 불구하고 각 함수를 이용하여 원하는 기능을 구현하는 과정은 쉽지 않았습니다. 특히 반복문, 조건문 등을 이용하여 원하고자 하는 값을 구하기 위해서는 논리적인 사고가 필요했습니다. 어떤 조건이 필요하고 어디서부터 어디까지 반복을 해야 원하는 값을 구할 수 있는지 제대로 알아야만 했고, 이것은 프로그래밍 언어뿐만 아니라 그 근본적인 원리가 되는 자료구조의 특성에 대한 이해가 기본적으로 필요하다는 걸 깨달았습니다.

한 학기동안 자료구조 수업을 통해 많은 내용을 배웠지만 제대로 구현하지 못해 아쉬운 부분이 있습니다. 어찌 보면 기존의 함수를 이용하는 조금 더 쉬운 길을 택했던 것이 수업 내용을 충분히 이해하지 못했기 때문인 것 같습니다. 결과적 측면에서는 모든 기능을 다 구현하지 못한 것이, 또 각 기능을 구현함에 있어 기존의 함수를 이용했다는 점이 아쉬운 부분이지만 과정적 측면에서는 정말 많은걸 배우고 느꼈습니다. 특히 이전까지는 생각보다 중요시 여기지 않았던 디버깅(debugging) 과정을 무수히 반복하다 보니 자료구조를 이해하는데 필요한 논리적 흐름을 이해하고, 스스로 코드를 완성해 나가는 능력을 조금이나마 키울 수 있었습니다.

개인적으로는 앞으로도 계속 컴퓨터 관련 공부를 해야하기 때문에, 이번 기회를 발판삼아 다음에는 기본에 충실하여 본인만의 자료구조형을 만들어내고 이를 이용할 수 있도록 노력하겠습니다. 더불어 이번에는 구현하지 못한 graph 알고리즘 부분은 얼핏 감은 잡은 것 같지만, 시간이 제한되어있고 자료구조에 대한 이해가 아직은 부족하여 구체화시키는데 어려움이 있었습니다. 만약 가능하다면, 다른 학생이 작성한 code나 교수님 혹은 조교가 작성한 Sample code를 봐서라도 끝까지 한번 구현해보고 싶습니다. 한편, 기회가 된다면 C언어로도 코드를 짜보고자 합니다. 포인터를 핑계로 C언어보다는 조금 더 쉬운 Python 언어 위주로 학습하였는데, 자료구조를 제대로 이해하고 활용하기 위해서는 C언어부터 제대로 공부를 해야 할 것 같습니다..

교수님께나 조교에게 질문을 드릴 때 마다 매번 친절하게 답변해주셔서 감사했습니다. 한 학기동안 정말 수고 많으셨습니다 ☺ 감사합니다!