자료구조

과제 5. 정렬

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

1. 대량의 데이터 생성

1.1 학생 데이터 생성하기

모든 학생들의 데이터를 저장하기 위한 빈 리스트(data), 학생 한명의 정보를 담을 빈 리스트(student)를 선언하고 각 데이터(학번, 이름, 전화번호)들을 다음과 같이 생성해주었다.

<학번 데이터 >
> 19 - 22년도에 입학한 학생의 8자리 숫자로 랜덤하게 생성하는데 각 입학년도의 학생수는 5000명이다. 동일한 학번을 가진 학생은 없어야 한다.

학번 8자리를 저장할 학번 리스트(st_id)를 선언해주고 학번 앞 4자리를 담당할 2019부터 2022 요소를 가진 입학년도 리스트(st_yr)를 만들어주었다. 입학년도가 다를 경우 학번 뒷 네자리는 동일해도 결국은 다른 학번이기 때문에 입학년도별로 5000개씩 중복없이 랜덤 추출했다. 랜덤하게 추출할 범위는 0~9999로 입학년도 앞 네자리와 랜덤 네자리를 붙여서 20,000개의 학번 리스트를 만들었다.
학번 리스트엔 (5000개씩) 입학년도 순으로 배치되어 랜덤하게 섞어주어 최종 길이가 20,000인 학번 리스트를 생성해주었다.

<이름 데이터>
>10자의 영문자로 랜덤하게 생성하고 중복되어도 상관없다.

공백으로 이름 문자열(st_name)을 선언해주고 10자의 아스키를 랜덤하게 추출하여 이름을 생성해주었다.

<전화번호 데이터>
'010'으로 시작하고 뒷 8자리는 0~99999999까지 랜덤으로 정수 8자리를 추출하여 뒤이어 붙이는 전화번호 문자열(st_phone)을 만들었다.

학번 데이터(st_id), 이름 데이터(st_name), 전화번호 데이터(st_phone)을 반복문을 이용해 한명씩 student 리스트에 담아주고 한 명의 정보가 다 채워지면 data 리스트에 저장해 총 20,000명의 학생 데이터를 생성하는 함수를 작성했다.

- 코드

```python
##  (1) 데이터 생성하기
def student_data():
    data = []

    st_id = [] #학번 8자리 저장할 데이터
    st_yr = [2019,2020, 2021, 2022]  #입학년도
    for i in st_yr:
        id_b = random.sample(range(10000), 5000)
        for j in range(5000):
            id_b[j]= str(i) +'%04d'%id_b[j]
        st_id += id_b
    random.shuffle(st_id)

    for i in range(20000):
        student = []
        student.append(st_id[i])

        st_name = ''
        for j in range(10):
            st_name += str(random.choice(string.ascii_uppercase))
        student.append(st_name)

        st_phone = '010'+'%08d'%random.randint(0,99999999)
        student.append(st_phone)

        data.append(student)

    return data
```

1.2 학번 중복 확인하기

데이터에 동일한 학번을 가진 학생이 없다는 것을 보증하기 위해 학번 중복 확인 함수를 구성했다. 데이터에 학번 중복이 있는지 없는지 알려주기 위한 변수(rep)을 False로 선언해주고 반복문을 돌면서 중복이 생기면 중복 변수를 True로 바꾸고 반복문을 빠져나와 중복이 발견되었음 알려준다. 반복문을 다 돌았음에도 중복이 발견되지 않으면 즉, 반복문이 끝났음에도 rep이 False라면 동일한 학번 데이터가 없다는 의미로 중복이 발견되지 않았음을 알려주는 함수이다.

학번 데이터 비교는 동일한 입학년도 내에서 해주었다. 각 입학년도에 따라 뒷 네자리를 랜덤으로 5000개씩 추출해줬기 때문에 중복 비교 또한 동일한 입학년도 내에서만 실시했다.

- 코드

```python
# 중복 발견하는 함수
def id_rep_check(st_data):
    rep = False
    for i in range(len(st_data)-1):
        for j in range(i+1, len(st_data)):
            if (st_data[i][0][0:4] == '2019') and (st_data[j][0][0:4] == '2019') :
                if st_data[i][0][4:10] == st_data[j][0][4:10]:
                    rep = True
                    break

            elif (st_data[i][0][0:4] == '2020') and (st_data[j][0][0:4] == '2020') :
                if st_data[i][0][4:10] == st_data[j][0][4:10]:
                    rep = True
                    break
            elif (st_data[i][0][0:4] == '2021') and (st_data[j][0][0:4] == '2021') :
                if st_data[i][0][4:10] == st_data[j][0][4:10]:
                    rep = True
                    break
            elif (st_data[i][0][0:4] == '2022') and (st_data[j][0][0:4] == '2022') :
                if st_data[i][0][4:10] == st_data[j][0][4:10]:
                    rep = True
                    break
        if rep == True:
            print("중복 발견")
            break
        else:
            print("중복 없음")
            break
```

정의한 함수가 잘 작동되는지 확인하기 위해 동일한 학번을 가진 학생 데이터를 임시로 만들어서 확인해주고 학생 데이터를 생성해 중복 데이터가 있는지 확인한다.

```
>> rep_test 데이터 :
['20200249', 'EFQWURSRWA', '01000781634']
['20200249', 'CXRXUOQIWS', '01093779800']
['20216397', 'TIFBRUJTAX', '01099541227']
['20221295', 'YVZQNWACAZ', '01005467894']
['20193110', 'XGMROJSHHC', '01056575793']
>> 'rep_test 데이터' 중복 확인 결과 => 중복 발견
```

```
>> 학생 데이터 (2000명당 1개 출력) :
['20218379', 'ZVOPAAAXHW', '01036305869']
['20194064', 'MGMJWSQQEM', '01042610242']
['20225570', 'XVWJUPFJFK', '01040774545']
['20220444', 'KDMXWYNMGP', '01031996926']
['20201877', 'VJYEURMEHK', '01008796126']
['20196590', 'BXCSBVESFW', '01028502378']
['20227152', 'MZJOHCJBKD', '01010730054']
['20214523', 'IDLMNORPZE', '01093100037']
['20211258', 'XGUBOCWWOH', '01092918815']
['20199578', 'TNACEXARVD', '01008326710']
>> '학생 데이터' 중복 확인 결과 => 중복 없음
```

2. 내장된 정렬 방법으로 정렬하기

2.1 학번을 기준으로 정렬하기

python에 내장되어 있는 정렬 함수를 이용하여 학번을 기준으로 정렬을 해주었다. 원래의 학생 데이터가 변하는 것을 막기 위해 얕은 복사를 실시한 후 정렬을 진행했다. 소요시간을 측정하기 위해 파이썬의 time 라이브러리를 이용하여 정렬 전후의 시간차를 계산해 소요시간을 나타냈다.

- 코드

```
## (2) 내장된 정렬 함수로 정렬하기
data = st_data
print("<Python 내장 함수로 정렬하기>")

#학번 기준으로 정렬
print("> 학번 기준 정렬")
start = time.time()
data.sort(key = lambda x: x[0])
finish = time.time()

pyth_sort_id_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 내장된 함수로 정렬하는데 소요된 시간은 {}초 입니다." .format(pyth_sort_id_time))
```

2.2 이름을 기준으로 정렬하기

위의 방법과 유사하게 기준이 되는 열을 이름열로 변경하여 정렬해주었다.

- 코드

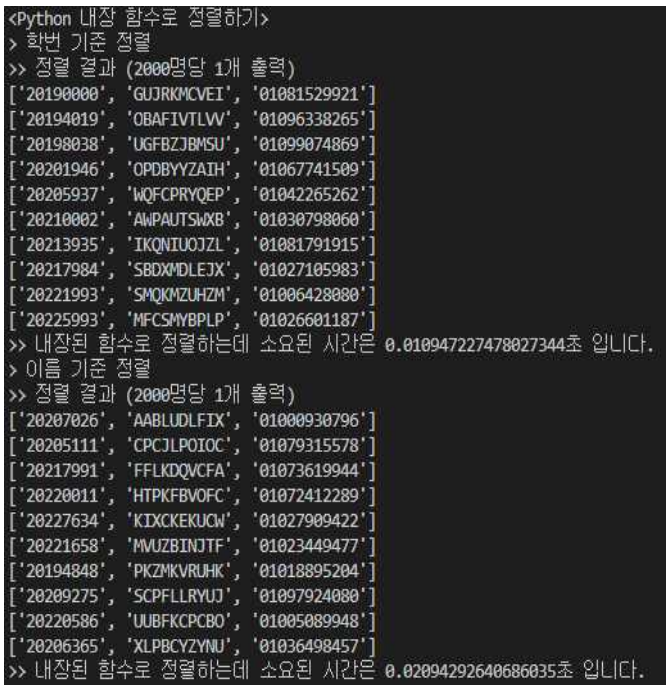
```
#이름 기준으로 정렬
data = st_data
print("> 이름 기준 정렬")
start = time.time()
data.sort(key = lambda x: x[1])
finish = time.time()

pyth_sort_name_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 내장된 함수로 정렬하는데 소요된 시간은 {}초 입니다." .format(pyth_sort_name_time))
```

2.3 내장된 정렬 방법의 결과

정렬이 잘 되었는지 확인하기 위해 2000명당 1개씩 데이터가 보여지게 했다.



3. 선택 정렬/퀵 정렬/힙 정렬

3.1 선택 정렬 구성하기

- 함수에서 정렬할 데이터(data)와 어떤 기준으로 정렬할지 기준열 값(criterion)을 인수로 지정했습니다. 학생 데이터에서 0일 경우 학번을 기준으로, 1일 경우 이름을 기준으로 정렬하게 됩니다. (이후 모든 정렬에서도 동일하게 적용하므로 처음에만 언급하겠습니다.)
- 데이터의 i 번째 데이터를 최소값 기준(least)으로 설정해놓고 (i+1)번째부터 데이터의 마지막까지 비교하여 least보다 작은 값이 있다면 해당 자릿수를 least로 저장한다.
- 가장 작은 값이 least 번째 값이 되고 least번째 값과 i 번째 값을 바꿔서 제일 작은 순서대로 데이터 앞쪽에 위치하도록 데이터를 정렬한다.(오름차순)

- 코드

```
# 선택 정렬 함수
def selection_sort(data, criterion):
    n = len(data)
    for i in range(n-1):
        least = i
        for j in range(i+1, n):
            if data[j][criterion] < data[least][criterion]:
                least = j
        data[i], data[least] = data[least] , data[i]
    return data
```

3.1.1 학번 기준으로 정렬

- 코드

```
#학번 기준으로 정렬
print("> 학번 기준 정렬")
start = time.time()
selection_sort(data, 0)
finish = time.time()

selection_sort_id_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 선택 정렬하는데 소요된 시간은 {}초 입니다." .format(selection_sort_id_time))
```

3.1.2 이름 기준으로 정렬

- 코드

```
#이름 기준으로 정렬
data = st_data
print("> 이름 기준 정렬")
start = time.time()
selection_sort(data, 1)
finish = time.time()

selection_sort_name_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 선택 정렬하는데 소요된 시간은 {}초 입니다." .format(selection_sort_name_time))
```

3.1.3 선택 정렬 결과

```
<선택 정렬로 정렬하기>
> 학번 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20190000', 'GUJRKMCVEI', '01081529921']
['20194019', 'OBAFIVTLVV', '01096338265']
['20198038', 'UGFBZJBMSU', '01099074869']
['20201946', 'OPDBYYZAIH', '01067741509']
['20205937', 'WQFCPRYQEP', '01042265262']
['20210002', 'AWPAUTSWXB', '01030798060']
['20213935', 'IKQNIUOJZL', '01081791915']
['20217984', 'SBDXMDLEJX', '01027105983']
['20221993', 'SMQKMZUHZM', '01006428080']
['20225993', 'MFCSMYBPLP', '01026601187']
>> 선택 정렬하는데 소요된 시간은 57.006535053253174초 입니다.
> 이름 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20207026', 'AABLUDLFIX', '01000930796']
['20205111', 'CPCJLPOIOC', '01079315578']
['20217991', 'FFLKDQVCFA', '01073619944']
['20220011', 'HTPKFBVOFC', '01072412289']
['20227634', 'KIXCKEKUCW', '01027909422']
['20221658', 'MVUZBINJTF', '01023449477']
['20194848', 'PKZMKVRUHK', '01018895204']
['20209275', 'SCPFLLRYUJ', '01097924080']
['20220586', 'UUBFKCPCBO', '01005089948']
['20206365', 'XLPBCYZYNU', '01036498457']
>> 선택 정렬하는데 소요된 시간은 46.77594184875488초 입니다.
```

3.2 퀵 정렬 구성하기

<quick_sort 함수>

- quick_sort 함수에서 정렬할 데이터(data)와 정렬 대상의 시작할 위치(low), 정렬 대상의 마지막 위치(high), 어떤 기준으로 정렬할지 기준열값(criterion)을 인자로 받는다.
- low가 high 보다 커질 때까지 partition함수를 통해 구한 pivot을 기준으로 작은쪽, 큰쪽 부분의 quick sort를 계속 수행한다. (재귀함수)

<partition 함수>

- 퀵 정렬의 한 단계를 실행하는 함수이고 이 함수를 통해 기준이 되는 pivot값을 구한다.
- pivot을 데이터의 처음 값으로 하기 때문에 pivot+1부터 high 까지 pivot과 비교를 실시한다.
- pivot번째 값(기준)으로 왼쪽에는 작은값, 오른쪽에는 큰값을 배치하기 위해 오른쪽에 있는 기준값보다 작은값과 왼쪽에 있는 큰 값을 바꿔준다. (오름차순)
- pivot을 기준으로 큰 부분과 작은 부분을 나누기 위해 경계인 j번째(작은 부분의 가장 마지막 위치)와 pivot번째 값을 바꾼뒤 pivot의 위치를 반환한다.

- 코드

```python
# 퀵 정렬 함수
def quick_sort(data, low, high, criterion):
    if low < high:
        pivot = partition(data, low, high , criterion)
        quick_sort(data, low, pivot-1, criterion)
        quick_sort(data, pivot+1, high, criterion)

    return data

def partition(data, pivot, high, criterion):
    i = pivot+1
    j = high

    while True:
        while i < high and data[i][criterion] < data[pivot][criterion]:
            i+=1
        while j > pivot and data[j][criterion] > data[pivot][criterion]:
            j-=1
        if j <= i :
            break

        data[i] , data[j] = data[j], data[i]
        i+=1
        j-=1
    data[pivot] , data[j] = data[j] , data[pivot]

    return j
```

3.2.1 학번 기준으로 정렬

- 코드

```
# 퀵 정렬 결과
data = st_data
print("<퀵 정렬로 정렬하기>")

#학번 기준으로 정렬
print("> 학번 기준 정렬")
start = time.time()
quick_sort(data, 0 , len(data)-1, 0)
finish = time.time()

quick_sort_id_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 퀵 정렬하는데 소요된 시간은 {}초 입니다." .format(quick_sort_id_time))
```

3.2.2 이름 기준으로 정렬

- 코드

```
#이름 기준으로 정렬
data = st_data
print("> 이름 기준 정렬")
start = time.time()
quick_sort(data, 0 , len(data)-1, 1)
finish = time.time()

quick_sort_name_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 퀵 정렬하는데 소요된 시간은 {}초 입니다." .format(quick_sort_name_time))
```

3.2.3 퀵 정렬 결과

```
<퀵 정렬로 정렬하기>
> 학번 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20190000', 'GUJRKMCVEI', '01081529921']
['20194019', 'OBAFIVTLVV', '01096338265']
['20198038', 'UGFBZJBMSU', '01099074869']
['20201946', 'OPDBYYZAIH', '01067741509']
['20205937', 'WQFCPRYQEP', '01042265262']
['20210002', 'AWPAUTSWXB', '01030798060']
['20213935', 'IKQNIUOJZL', '01081791915']
['20217984', 'SBDXMDLEJX', '01027105983']
['20221993', 'SMQKMZUHZM', '01006428080']
['20225993', 'MFCSMYBPLP', '01026601187']
>> 퀵 정렬하는데 소요된 시간은 0.11374258995056152초 입니다.
> 이름 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20207026', 'AABLUDLFIX', '01000930796']
['20205111', 'CPCJLPOIOC', '01079315578']
['20217991', 'FFLKDQVCFA', '01073619944']
['20220011', 'HTPKFBVOFC', '01072412289']
['20227634', 'KIXCKEKUCW', '01027909422']
['20221658', 'MVUZBINJTF', '01023449477']
['20194848', 'PKZMKVRUHK', '01018895204']
['20209275', 'SCPFLLRYUJ', '01097924080']
['20220586', 'UUBFKCPCBO', '01005089948']
['20206365', 'XLPBCYZYNU', '01036498457']
>> 퀵 정렬하는데 소요된 시간은 0.10271716117858887초 입니다.
```

3.3 힙 정렬 구성하기

<heap_sort 함수>

- 정렬할 데이터(data) 와 어떤 기준으로 정렬할지 기준열값(criterion)을 인자로 받는다.
- len(data)-1을 힙의 총 사이즈로 선언한다.
- 트리의 아래부터 adjust 함수를 통해 root에 가장 큰 값이 오도록 정렬한다.
- 트리의 사이즈를 1을 줄여 가장 마지막 값(큰 값)을 제외 하고 다시 힙 정렬을 실시한다.(재귀함수)

<adjust 함수>

- 힙 정렬의 한단계를 실행하는 함수이다.
- 한 root의 child가 없을 때 까지 반복하여 child에서 큰쪽을 찾아서 힙 정렬을 실행한다.
- child의 수보다 root의 수가 크면 바꾸지 않고 반복문을 빠져나온다.
- child의 값이 더 크면 root에 큰 값이 오도록 바꿔준다.

- 코드

```
# 힙 정렬 함수
def adjust(data, root, size, criterion):     #힙 정렬의 한단계 실행
    while 2*root+1 <= size :    #한 root의 child가 없을 때까지 반복
        child = 2*root+1
        if (child < size) and data[child][criterion] < data[child+1][criterion]:  # child에서 큰쪽을 찾아서 힙 정렬 실행
            child+=1
        if data[root][criterion] >= data[child][criterion]:  #child의 수보다 root의 수가 크면 swap 하지 않고 break
            break

        data[root] , data[child] = data[child] , data[root]  # child의 값이 더 크면 root에 큰 값이 오도록 바꿔준다.
        root = child

def heap_sort(data, criterion):      #힙 정렬을 하는 함수
    size = len(data)-1               #힙의 총 사이즈
    for i in reversed(range(0, (size+1)//2-1)): # 트리의 아래부터 adjust 함수를 통해 root에 가장 큰 값임 나오도록 정렬
        adjust(data, i , size, criterion)

    for j in range(size):
        data[0] , data[size] = data[size] , data[0]   #루트의 값(가장 큰 값)를 트리의 끝으로 보낸다.
        size -= 1                                     #트리의 사이즈를 1 줄여 가장 마지막 값(큰 값)을 제외하고 힙 정렬
        adjust(data, 0, size, criterion)
```

3.3.1 학번 기준으로 정렬

- 코드

```
# 힙 정렬 결과
data = st_data
print("<힙 정렬로 정렬하기>")

#학번 기준으로 정렬
print("> 학번 기준 정렬")
start = time.time()
heap_sort(data, 0)
finish = time.time()

heap_sort_id_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 힙 정렬하는데 소요된 시간은 {}초 입니다." .format(heap_sort_id_time))
```

3.3.2 이름 기준으로 정렬

- 코드

```
#이름 기준으로 정렬
data = st_data
print("> 이름 기준 정렬")
start = time.time()
heap_sort(data, 1)
finish = time.time()

heap_sort_name_time = finish-start

# 정렬이 잘 되어있는지 확인
print(">> 정렬 결과 (2000명당 1개 출력)")
for i in range(len(data)):
    if i % 2000 == 0 :
        print(data[i])
    else:
        continue
print(">> 힙 정렬하는데 소요된 시간은 {}초 입니다." .format(heap_sort_name_time))
```

3.3.3 힙 정렬 결과

```
<힙 정렬로 정렬하기>
> 학번 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20190000', 'GUJRKMCVEI', '01081529921']
['20194019', 'OBAFIVTLVV', '01096338265']
['20198038', 'UGFBZJBM5U', '01099074869']
['20201946', 'OPDBYYZAIH', '01067741509']
['20205937', 'WQFCPRYQEP', '01042265262']
['20210002', 'AWPAUTSWXB', '01030798060']
['20213935', 'IKQNIUOJZL', '01081791915']
['20217984', 'SBDXMDLEJX', '01027105983']
['20221993', 'SMQKMZUHZM', '01006428080']
['20225993', 'MFCSMYBPLP', '01026601187']
>> 힙 정렬하는데 소요된 시간은 0.20744299888610 84초 입니다.
> 이름 기준 정렬
>> 정렬 결과 (2000명당 1개 출력)
['20207026', 'AABLUDLFIX', '01000930796']
['20205111', 'CPCJLPOIOC', '01079315578']
['20217991', 'FFLKDQVCFA', '01073619944']
['20220011', 'HTPKFBVOFC', '01072412289']
['20227634', 'KIXCKEKUCW', '01027909422']
['20221658', 'MVUZBINJTF', '01023449477']
['20194848', 'PKZMKVRUHK', '01018895204']
['20209275', 'SCPFLLRYUJ', '01097924080']
['20220586', 'UUBFKCPCBO', '01005089948']
['20206365', 'XLPBCYZYNU', '01036498457']
>> 힙 정렬하는데 소요된 시간은 0.21941280364990234초 입니다.
```

4. 최종 결과

4.1 정렬 방법에 따른 소요시간

각 정렬 방법과 정렬 기준에 따른 정렬 소요 시간은 다음과 같다.

시간 (밀리초 단위)	내장 함수	선택정렬	퀵정렬	힙정렬
학번 기준	0.0109	57.0065	0.1137	0.2074
이름 기준	0.0209	46.7759	0.1027	0.2194

정렬 방법 중 가장 소요시간이 오래걸리는 방법은 선택정렬이다.
내장된 함수를 제외하고 제일 적게 소요된 방법은 퀵 정렬이다.