데이터구조설계

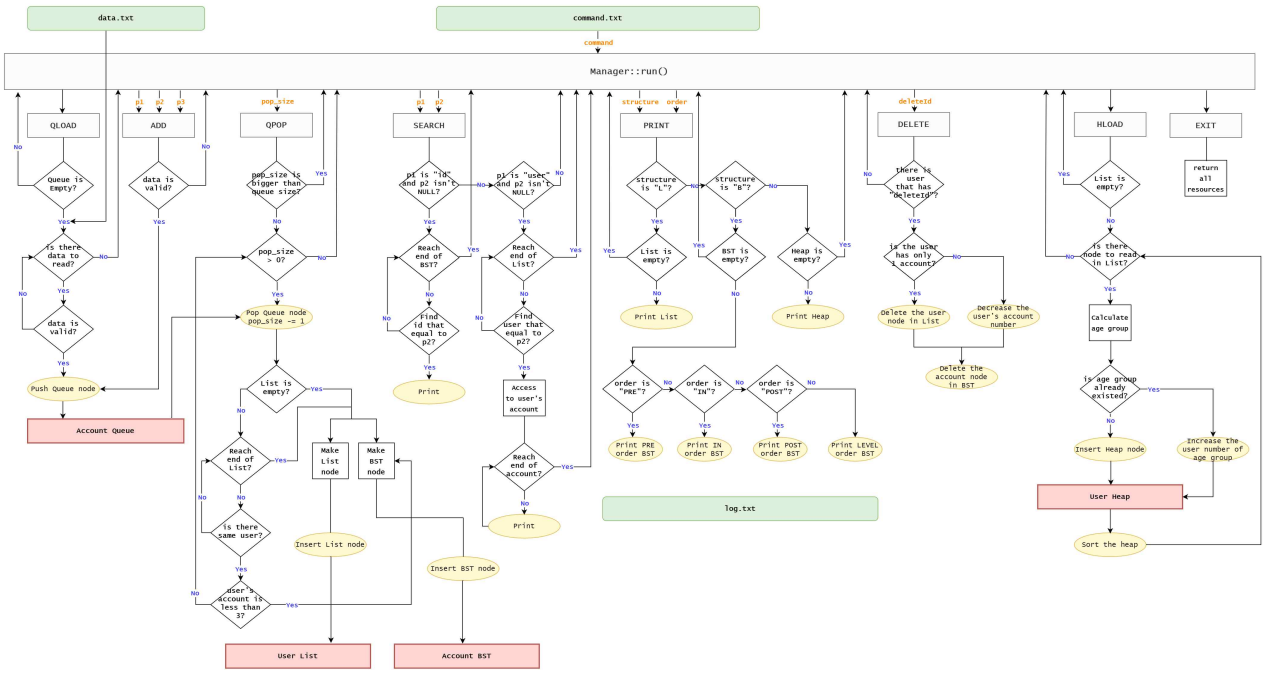
Project 1

담당교수	최상호 교수님
학 과	컴퓨터정보공학부
학 번	2015722035
성 명	최한솔
날 짜	2021. 10. 11 (월)

1. Introduction

본 프로젝트에서는 Queue, Binary Search Tree, Linked List, Heap에 해당하는 자료구조를 활용하여 계정 관리 프로그램을 구현하는 것을 목표로 한다.
해당 프로그램은 <data.txt>파일에 기록되어 있는 사용자의 이름, 나이, 계정 ID를 통하여 각각의 자료구조들을 설계한다. AccountQueue는 해당 파일의 사용자 정보를 읽어와 만들어진 node들로 구성되며, UserList와 AccountBST는 이러한 AccountQueue에 기반하여 만들어진다.
UserList는 사용자의 이름과 나이, 보유한 계정 수 정보를 가진 node들로 구성되며, AccountBST는 계정 ID와 해당 계정을 소유한 사용자 이름 정보를 가진 node들로 구성된다. 여기서 node들은 계정 ID의 사전적 순서에 의해 정렬된다.
또한, AccountBST 내의 node들은 특정 사용자가 가지고 있는 계정의 node들끼리 별도로 연결하여 Linked List를 구성하게 된다. UserHeap은 UserList의 node들을 통해, 특정 나잇대에 해당하는 사용자 수에 대한 정보를 가진 node들을 통해 구성되며, 사용자 수가 많은 나잇대를 기준으로 정렬된다.
해당 프로그램은 <command.txt>파일의 모든 명령어를 순차적으로 읽어온 후, QLOAD, ADD, QPOP, SEARCH, PRINT, DELETE, HLOAD, EXIT에 해당하는 기능을 수행하고 해당 결과를 <log.txt>파일에 출력하게 된다.

2. Flow Chart



QLOAD

기존의 Queue가 존재하지 않으면 <data.txt>파일에 기록된 사용자의 정보를 하나씩 읽어오며, 해당 정보가 사용자의 이름, 나이, 계정 ID에 대한 정보를 모두 가지고 있으면 새로운 node를 생성하여 Queue에 삽입한다. <data.txt>파일에서 더이상 읽어올 데이터가 없거나, Queue가 이미 존재하는 경우 해당 기능은 수행되지 않는다.

ADD

사용자로부터 직접 사용자의 이름, 나이, 계정 ID에 대한 데이터를 입력받으며 모두 정상적으로 입력받은 경우, 새로운 node를 생성하여 Queue에 삽입한다. 정상적인 입력을 받지 못한 경우 이 기능은 수행되지 않는다.

QPOP

Queue에서 추출하는 node의 수에 대한 데이터 <pop_size>를 함수의 인자로 전달받는다.
해당 인자가 Queue의 크기보다 작거나 같을 때, 해당 인자만큼 Queue에서 node를 하나씩 추출해온다.
만약 List가 비어있을 경우, Queue node의 정보들을 통해 List node와 BST node를 생성하여 각각 List와 BST에 삽입하게 된다.
만약 List가 비어있지 않을 경우, List의 끝에 도달할 때까지 해당 정보를 지닌 사용자가 이미 존재하는지를 탐색하여 존재할 경우엔 사용자의 List node에 접근하여 계정 수를 확인한다.
만약 계정 수가 3개보다 적을 경우, 계정에 대한 새로운 BST node를 생성하여 BST에 삽입하며, 계정 수가 3개보다 적지 않을 경우, 새로운 계정의 삽입을 수행하지 않는다.
List에서 해당 정보를 지닌 사용자가 탐색 되지 않은 경우엔 새로운 List node와 BST node를 생성하여 각각 List와 BST에 삽입한다.

SEARCH

전달받은 <p1>이 "id"일 경우, <p2>에 해당하는 계정 ID를 지닌 node가 BST에 존재하는지를 탐색한다.
만약 존재할 경우, 계정 ID와 이를 지닌 사용자의 이름에 대한 정보를 <log.txt>파일에 출력한다.
만약 전달받은 p1이 "user"일 경우, p2에 해당하는 사용자가 List에 존재하는지를 탐색한다. 존재한다면 해당 사용자가 지닌 BST node들에 대한 연결 List에 접근하여 모든 계정들의 정보를 출력한 후 수행을 종료한다. 만약, p1과 p2에 대하여 정상적인 입력을 받지 못한 경우엔 해당 기능은 수행되지 않는다.

PRINT

출력하고자 하는 자료구조를 의미하는 <structure>와 출력 순서를 의미하는 <order>를 인자로 전달받는다. 입력받은 structure에 해당하는 List, BST 혹은 Heap에 접근하여 해당 자료구조가 비어있지 않을 때 이를 출력하게 된다. 특히 BST의 경우에는 입력받은 order에 해당하는 출력 순서에 따라서 출력한 후 수행을 종료한다.

DELETE

삭제하고자 하는 계정 ID를 의미하는 <deleteId>를 인자로 전달받아 해당 계정 ID가 존재하는지 여부를 확인하기 위해 BST를 탐색한다. 탐색 결과 해당 계정 ID가 존재하지 않으면 수행을 종료한다. 만약 해당 계정이 존재한다면, 해당 계정 ID를 지닌 사용자가 몇 개의 계정을 가지고 있는지를 확인한다.
1개의 계정을 가지고 있을 경우에는 List에서 해당 사용자의 List node를 삭제하고 해당 계정 ID의 BST node 또한 삭제한다. 만약 2개 이상의 계정을 가지고 있을 경우에는 사용자의 계정 수를 감소시킨 후 해당 계정 ID의 BST node를 삭제한 후 수행을 종료한다.

HLOAD

List가 비어있을 경우 해당 기능은 수행되지 않으며, List가 존재할 경우에 List의 각 node에 접근하여 해당 node의 사용자 나잇대를 계산한다.

만약 Heap에 같은 나잇대의 node가 이미 존재할 경우, 해당 node의 사용자 수를 증가시키며 Heap에 같은 나잇대의 node가 존재하지 않을 경우에는 새로운 Heap node를 생성하여 삽입한 후 Heap을 정렬하게 된다. List 안의 모든 node에 접근한 후 해당 수행이 종료된다.

EXIT

할당받은 모든 자원을 반납하고 프로세스를 종료한다.

3. Algorithm

QLOAD : data.txt 파일의 데이터에 기반하여 node를 생성, Queue에 저장한다

[1] Queue가 존재하지 않을 경우에 한하여 수행을 계속한다.

[2] <data.txt>파일에서 데이터를 한 줄씩 읽어온다.

- 읽어올 데이터가 없을 경우, 수행을 멈춘다.

[3] 읽어온 데이터를 사용자의 이름, 나이, 계정 ID로 나눈다.

- 문자열을 tokenize 한 후에 userInformation[i]에 각각 저장한다.
- 각각을 읽은 후에 포인터의 주소 공간을 초기화한다.

[4] 나눈 데이터를 할당받은 주소 공간에 저장한다.

- 이름, 나이, 계정 ID 중 하나라도 없을 경우, [1]의 과정으로 돌아간다.

[5] 해당 데이터를 통해 새로운 Queue node를 생성한다.

[6] Queue에 삽입한다.

ds_queue->PUSH() 실행에서의 알고리즘

1. Queue가 비어있을 경우, front에 해당 node를 저장한다.
2. Queue가 비어있지 않은 경우, 마지막 node의 다음 위치에 새로운 node를 저장한다.
3. Queue의 크기를 증가시킨다.

[7] <data.txt>파일에 대해 [1]-[5]의 과정을 모두 마친 이후, Queue node들을 출력한다.

ADD : 직접 입력받은 데이터에 기반하여 node를 생성하고 Queue에 저장한다

[1] 전달받은 인자를 새로운 주소 공간에 저장한다. 인자가 완전하지 않을 경우에는 수행을 멈춘다.

[2] 해당 데이터를 통해 새로운 Queue node를 생성한다.

[3] Queue에 삽입한다.

[4] 생성한 node의 정보를 출력한다.

QPOP : Queue node의 정보들을 통해 List와 BST를 생성한다

[1] Queue의 크기보다 많은 수의 pop 연산을 요구할 경우, 수행을 멈춘다.

[2] 요구한 pop 연산 횟수만큼 Queue의 head에 위치한 node를 읽어온다.

[3] 읽어온 node를 삽입하기 위해 우선 List를 탐색한다.

ds_list->Insert() 실행에서의 알고리즘

1. List를 탐색한다.

<case 1> List가 빈 경우

i . 새로운 List node를 만들고, 이를 List의 root 위치에 삽입한다.

ii. BST node를 생성한 후, 해당 List node의 Linked list (account)에 추가한다.
iii. 해당 BST node를 return 한다.

<case 2> List가 비지 않았고, 동일한 사용자의 node가 존재하는 경우
i. 해당 사용자의 계정 수를 확인한다.
ii. 계정 수가 3개 이상일 경우, 빈 BST node를 return 한다. -> Error 301 출력
iii. 계정 수가 3개 미만인 경우, 해당 사용자 List node를 기억한다.

<case 3> List가 비지 않았고, 동일한 사용자가 존재하지 않을 경우
i. 삽입하기 위한 List의 마지막 node를 기억한다.

2. 기억한 List node의 위치에 새롭게 만든 node를 삽입한다.
3. BST node를 생성한 후, 해당 List node의 Linked list에 위치시킨다.
4. 해당 BST node를 return 한다.

[4] Return 받은 BST node를 BST에 삽입한다. -> 빈 BST node일 경우, 수행되지 않는다.

ds_bst->Insert() 실행에서의 알고리즘

1. BST를 탐색한다.
<case 1> BST가 빈 경우 -> BST의 root 위치에 삽입한다.
<case 2> BST가 비지 않은 경우 -> ID 값의 사전 순서를 비교하여 BST 탐색
i. 삽입하고자 하는 ID 값의 사전 순서가 작을 경우, left child로 이동
ii. 삽입하고자 하는 ID 값의 사전 순서가 클 경우, right child로 이동
2. 최종적으로 찾은 위치에 BST node를 삽입한다.

SEARCH : BST와 List에서 해당하는 정보를 찾아 출력한다

[1] 전달받은 두 인자값을 확인한다.

<case 1> 첫 번째 인자값이 "id"이고, 두 번째 인자값이 NULL이 아닌 경우

ds_bst->SearchId() 함수 실행에서의 알고리즘

1. BST를 탐색한다.
i. 찾고자 하는 ID 값의 사전 순서가 작을 경우, left child로 이동
ii. 찾고자 하는 ID 값의 사전 순서가 클 경우, right child로 이동
2. 찾은 경우, 탐색을 종료하고 해당 정보를 출력한 후, true를 return 한다.
3. 찾지 못한 경우, false를 return 한다. -> Error 400 출력

<case 2> 첫 번째 인자값이 "user"이고, 두 번째 인자값이 NULL이 아닌 경우

ds_list->Search() 함수 실행에서의 알고리즘

1. List를 탐색한다.
i. List의 root node를 할당한 후, 두 번째 인자값과 문자열을 비교한다.
ii. GetNext()를 통해 다음 node로 이동하면서 비교를 계속하여 나간다.
2-1. 찾은 경우, 탐색을 종료하고 해당 list node의 linked list (account)에 접근한다.
linked list의 head node부터 마지막 node까지 모든 account 정보를 출력한다.
2-2. 찾지 못한 경우, false를 return 한다. -> Error 400 출력

PRINT : List, BST 혹은 Heap의 구조를 출력한다

[1] 전달받은 첫 번째 인자값을 확인한 후, 해당 자료구조의 존재 유무를 확인한다.

<case 1> List 출력 -> **ds_list->Print_L()**

<case 2> BST 출력 -> 두 번째 인자값을 확인한 후 해당 순서로 출력한다.

-> **ds_bst->Print_PRE()**, **Print_IN()**, **Print_POST()**, **Print_LEVEL()**

- Level order 출력의 경우, queue 구조를 활용하여 구현하게 된다.

<case 3> Heap 출력 -> **ds_heap->Print()**

DELETE : 입력받은 데이터를 List와 BST에서 삭제한다

[1] 입력받은 ID를 지닌 사용자가 존재하는지 List를 탐색한다.

ds_list->FindIdUser() 실행에서의 알고리즘

1. List의 root node부터 순차적으로 탐색한다.
2. List node의 linked list (account)의 head node부터 순차적으로 탐색한다.

<case 1> 해당하는 계정의 사용자를 찾은 경우, 이름을 return 한다.

<case 2> 찾지 못한 경우, NULL을 return 한다.

[2] 사용자의 이름을 return 받은 경우, 해당 사용자를 List에서 삭제한다.

ds_list->Delete_Account() 실행에서의 알고리즘

1. List에서 해당 사용자 이름의 node를 탐색한다.
2. 탐색한 해당 사용자의 계정 수를 확인한다.

<case 1> 계정이 1개인 경우, List에서 해당 node의 링크를 제거한다.

<case 2> 계정이 2개 이상인 경우, 계정 수를 감소시킨다.

-> **deleteListNode->Delete_Account()** 실행 : 해당 계정 node의 링크를 제거

[3] BST에서 해당 Id의 node를 삭제한다.

ds_bst->Delete() 실행에서의 알고리즘

1. BST에서 삭제할 Id의 BST node를 탐색한다.
2. BST node를 삭제한다.

<case 1> child node가 없는 경우 -> 링크를 제거한다.

<case 2> child node를 1개 가지고 있는 경우

-> 삭제할 node의 parent node와의 관계를 고려하여 child node의 링크를 연결해준다.

<case 3> child node를 2개 가지고 있는 경우

-> 삭제할 node의 right sub tree에서 가장 작은 값의 node를 대신 위치시키고 제거한다.

HLOAD : List를 기반으로 새로운 Heap을 생성한다

[1] List가 비어있지 않을 경우, Heap을 초기화한다.

[2] List node를 하나씩 읽어 나잇대를 계산한다. - 나잇대 = (사용자의 나이 / 10) * 10

[3] Heap에 삽입한다. - Heap은 vector 구조로 구현된다.

ds_heap->Insert() 실행에서의 알고리즘

1. 해당 나잇대의 Heap node가 존재하는지 탐색한다.

<case 1> 해당 나잇대의 node가 존재하는 경우, 해당 나잇대의 사용자 수를 증가시킨다.

<case 2> 해당 나잇대의 node가 존재하지 않는 경우, 새로운 node를 생성하여 Heap에 삽입한다.

[4] Heap을 재정렬한다.

ds_heap->Sort() 실행에서의 알고리즘

1. child node와 parent node의 index 값을 저장한다.
2. Heap의 구조적 특성을 고려하여, child와 parent node의 값을 비교하여 위치를 변경한다.

- Parent node의 index = Child node의 index / 2

4. Result Screen

코드에 대한 테스트 과정에서 사용한 파일의 내용은 다음과 같다.

```
kevin 42 dangun
bob 17 zlxen2
stuart 33 aaabbc
tom 22 batman
bob 17 5102842
tom 22 user123
elsa 24 frozen
david 28 david28
bob 17 cusine7
stuart 33 stuart3
bob 17 bobbob17
```

▲ **data.txt 파일**

```
LOAD
QLOAD
QLOAD
ADD james 29 james19
QPOP 4
QPOP 5
QPOP 3
QPOP 1
PRINT L
PRINT B PRE
SEARCH id batman
SEARCH bob
SEARCH user bob
PRINT H
HLOAD
PRINT H
DELETE stuart
DELETE dangun
PRINT L
HLOAD
PRINT H
EXIT
```

command.txt 파일 ▶

1. LOAD

존재하지 않는 command에 대해서 오류를 출력한다.

```
======= ERROR =======
800
======================
```

2. QLOAD

data.txt 파일의 데이터를 통해 Account Queue를 구축한다.

```
======== QLOAD ========
kevin/42/dangun
bob/17/zlxen2
stuart/33/aaabbc
tom/22/batman
bob/17/5102842
tom/22/user123
elsa/24/frozen
david/28/david28
bob/17/cusine7
stuart/33/stuart3
bob/17/bobbob17
======================
```

3. QLOAD

이미 Account Queue가 존재하기 때문에 해당 수행 결과 오류를 출력한다.

```
======= ERROR =======
100
=====================
```

4. ADD james 29 james19

command 상에 사용자가 직접 입력한 사용자의 이름, 나이, 계정 ID 정보를 통해 Queue node를 생성하여 Account Queue에 삽입한 후 이에 대한 정보를 출력한다.

```
======== ADD ========
james/29/james19
=====================
```

5. QPOP 4

6. QPOP 5

Account Queue에서 4개의 Queue node를 추출하여 User List와 Account BST를 구축한다.
현재 Account Queue에 들어있는 node의 개수는 총 12개이기 때문에 오류 없이 동작한다.
이후 5개의 Queue node를 추가로 추출하지만 역시 Account Queue에 들어있는 node의 개수는 총 8개이기 때문에 오류 없이 동작한다.

```
======= QPOP =======
Success
=====================
```

```
kevin 42 dangun
bob 17 zlxen2
stuart 33 aaabbc
tom 22 batman
bob 17 5102842
tom 22 user123
elsa 24 frozen
david 28 david28
bob 17 cusine7
stuart 33 stuart3
bob 17 bobbob17
```

```
======= QPOP =======
Success
=====================

======= QPOP =======
Success
=====================

======= ERROR =======
301
=====================

======= QPOP =======
Success
=====================

======= ERROR =======
300
=====================
```

7. QPOP 3

8. QPOP 1

Account Queue에서 3개의 Queue node를 추출하여 User List와 Account BST를 구축한다.
현재 Account Queue에 들어있는 node의 개수는 총 3개이기 때문에 오류 없이 동작해야 한다.
하지만 11번째에 위치한 데이터의 경우 이를 추가할 경우, bob의 4번째 계정에 해당하기 때문에 User List와 Account BST에 추가하지 않고 301 오류코드를 출력한다.
이후, ADD를 통해 추가된 마지막 1개의 Queue node를 추출하여 List와 BST를 구성하고 QPOP을 성공적으로 수행했음을 알린다. 다음 QPOP 1 수행 결과, Account Queue에 더는 node가 존재하지 않기 때문에 300 오류코드를 출력하는 것을 확인할 수 있다.

9. PRINT L

Account List의 모든 node의 정보를 사용자의 이름/나이/계정 수의 순서로 출력한다.
12개의 queue node 중에서 중복된 bob의 계정에 대한 것을 제외한 11개가 List에 들어있는 것을 확인할 수 있다.

```
======= PRINT ========
LIST
kevin/42/1
bob/17/3
stuart/33/2
tom/22/2
elsa/24/1
david/28/1
james/29/1
======================
```

10. PRINT B PRE

AccountBST에 존재하는 node들을 PRE order로 출력한다.

```
======= PRINT =======
BST PRE
dangun/kevin
aaabbc/stuart
5102842/bob
batman/tom
david28/david
cusine7/bob
zlxen2/bob
user123/tom
frozen/elsa
stuart3/stuart
james19/james
=====================
```

11. SEARCH id batman

12. SEARCH bob

13. SEARCH user bob

11번 command에 대해선 "batman"에 해당하는 계정 ID를 지닌 사용자의 이름을 검색하여 계정 ID /사용자 이름의 순서로 출력한다. 12번 command에서는 id 혹은 user의 값을 지닌 인자를 정상적으로 입력받지 못하였기 때문에 오류를 출력한다. 13번 command에서는 "bob"이라는 이름을 가진 사용자의 모든 계정을 출력하는 것을 확인할 수 있다.

```
======= SEARCH =======
ID
batman/tom
======================
```

```
======= ERROR =======
400
=====================
```

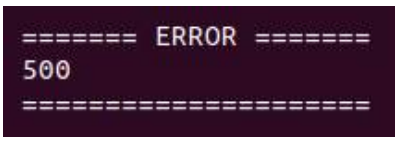
```
======= SEARCH =======
User
bob/17
zlxen2
5102842
cusine7
======================
```

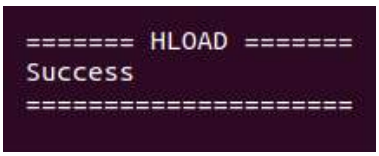
14. PRINT H

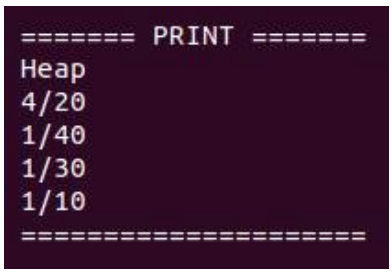
15. HLOAD

16. PRINT H

14번 command에 대해선 아직 User Heap이 생성되지 않았기 때문에 오류를 출력한다.
이후 15번 command를 통해 User List의 node 정보를 기반으로 새로운 Heap을 생성하고
16번 command를 통해 이를 출력한다.





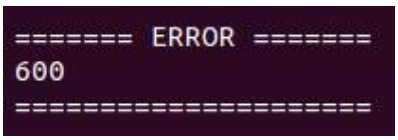


17. DELETE stuart

18. DELETE dangun

19. PRINT L

DELETE를 수행하는 데 있어서 삭제하고자 하는 계정 ID에 대한 인자를 필요로 한다.
17번 command에서 입력한 "stuart"에 해당하는 계정 ID는 탐색 결과 존재하지 않기 때문에 오류를 출력한다. 18번 command를 통해선 정상적으로 해당 계정 ID를 가진 "kevin"이라는 사용자가 삭제된 것을 List 출력을 통해 확인할 수 있다.

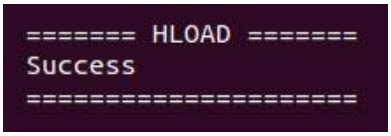


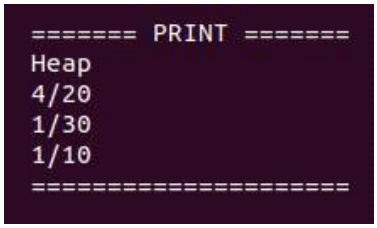
20. HLOAD

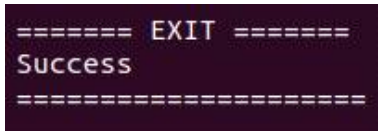
21. PRINT H

22. EXIT

DELETE 수행을 통해 변화된 User List에 대해 다시 node의 정보들을 읽어와 User Heap을 구축하고 이를 출력한다. 이후 EXIT command를 통해 할당받았던 자원들을 반납한 다음 종료한다.







5. Consideration

이번 프로젝트를 진행하면서 큐, 연결 리스트, 이진 탐색 트리, 힙 등의 다양한 자료구조에 대해 구현해 보았다. 파일입출력을 통해 데이터들을 읽어오고 이를 통해 사용자들의 정보를 가지고 있는 큐를 구현했으며, 큐에 들어있는 노드들을 읽어와서 사용자의 이름과 나이 그리고 계정의 수를 따로 카운트하여 가지고 있는 연결 리스트를 구현하였고, 각각의 계정 ID와 해당 계정을 소유하고 있는 사용자 정보를 가지고 있는 이진 탐색 트리를 구현하였다. 마지막으로 연결 리스트의 노드들을 읽어와서 나잇대 별로 사용자들의 수를 카운트하고 이를 통해 큰 값을 우선으로 정렬하는 힙을 구현해 볼 수 있었다. 이번 프로젝트를 진행하면서 이러한 기본적인 자료구조들에 대해서 익히고 또한 이들을 직접 구현해보는 기회가 되어 좋았다. 이 과정에서 각각의 자료구조들이 이루는 기본적인 형태와 이를 위해서 구현해야 할 노드들 간의 연결 관계 등을 정확히 숙지할 수 있도록 노력하고자 하였다.
특히 이진 탐색 트리 안의 노드들 사이에서 특정 사용자가 가지고 있는 계정에 대한 노드들을 따로 연결 리스트로 연결하고 이를 유저 리스트 노드의 BST 노드 포인터에 연결해주는 부분이 기억에 남는다. 어떠한 자료구조를 구현할 때 적절한 종류의 트리를 활용하는 것을 통해 삽입, 삭제 등의 연산에서 시간복잡도를 줄여 훨씬 효율적인 구현을 할 수 있다는 것을 몸소 느낄 수 있었다.
이번 프로젝트를 진행하면서 어려움을 겪으면서 생각해보게 된 부분은 크게 두 부분이었다.
첫째로, QLOAD의 과정에서 파일에서 입력받은 데이터들을 통해 노드를 생성하고 이를 통해 큐를 구현하는 데 있어서 처음에 어려움을 겪었다. 큐 노드를 생성하고 이들을 올바르게 연결하여 출력해보았을 때, 모두 같은 데이터를 가지고 있었기 때문이다. 정신없이 생각해보다 출력된 노드들의 데이터 값들을 보았을 때, 사용자의 나이에 대한 정보는 각각의 노드들이 올바르게 가지고 있었기 때문에 큐 노드들을 연결하는 부분에 있어서의 문제는 아니라는 것을 느꼈고, 이후 이것이 파일에서 읽어온 데이터를 그대로 큐 노드의 각 데이터 값에 저장해주었기 때문이라는 것을 알 수 있었다. 굉장히 생각이 짧았던 실수라는 것을 깨닫게 되었고, 파일에서 읽어온 데이터를 메모리 공간을 할당받은 새로운 CHAR 포인터에 저장한 후 노드의 데이터 값에 저장함으로써 해결할 수 있었다. 이에 대해 더욱 확실히 숙지하고 앞으로는 이러한 실수를 하지 않도록 해야겠다고 느꼈다.
두 번째로, 처음에 프로젝트를 시작함에 있어서 각각의 자료구조들에 대한 구현을 우선으로 여기고 진행하였었다. 결론적으로 모든 자료구조에 대한 각 함수들을 구현한 후, 전체적인 동작을 관할하는 Manager 함수에 도달했을 때 나의 접근 방법이 효율적이지 않다는 것을 뼈저리게 느끼게 되었다. 작은 것부터 구현해야 한다는 생각에서 이러한 방식으로 진행하였는데 막상 Manager 함수에서 이 작은 각각의 함수들을 유기적으로 구성하여 하나의 command 동작을 구현하려 하다 보니, 이전에 작성한 작은 부분의 코드들을 수정해야 하는 일들이 종종 발생하였다. 또한, 이러한 접근법으로 인해 각 동작에 대하여 가장 효율적인 방법으로 설계하는 부분에 있어서 좋은 결과를 얻지 못한 것 같다.
비록 프로젝트가 수행해야 할 기능을 구현하면서 큰 공부가 되었지만 많은 아쉬움도 남았다.
보다 나은 결과물을 위해 앞으로는 무엇인가를 설계하고 구현할 때, 전체적인 큰 부분을 이해하는 것을 우선으로 하고, 그 이후에 이를 세분화하여 작은 부분들로 나누어 만들어나가는 방식으로 접근해야겠음을 마음속에 담게 되었고, 다음 프로젝트에서는 이러한 부분을 실천해서 보다 나은 작품을 만들어야겠다고 생각했다.