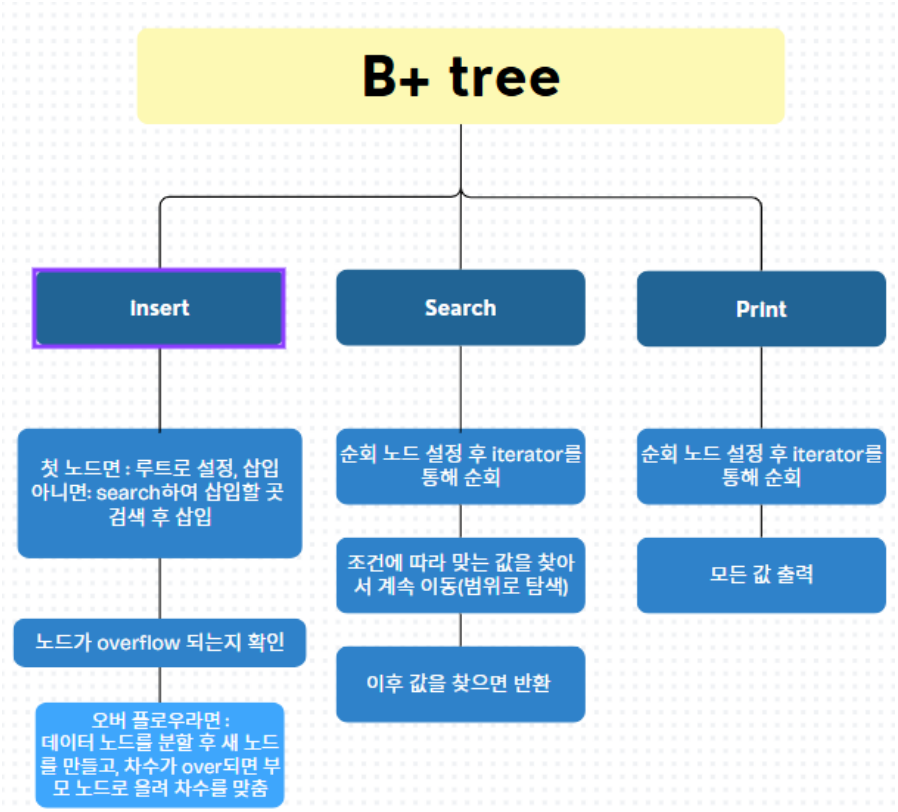
- Introduction:

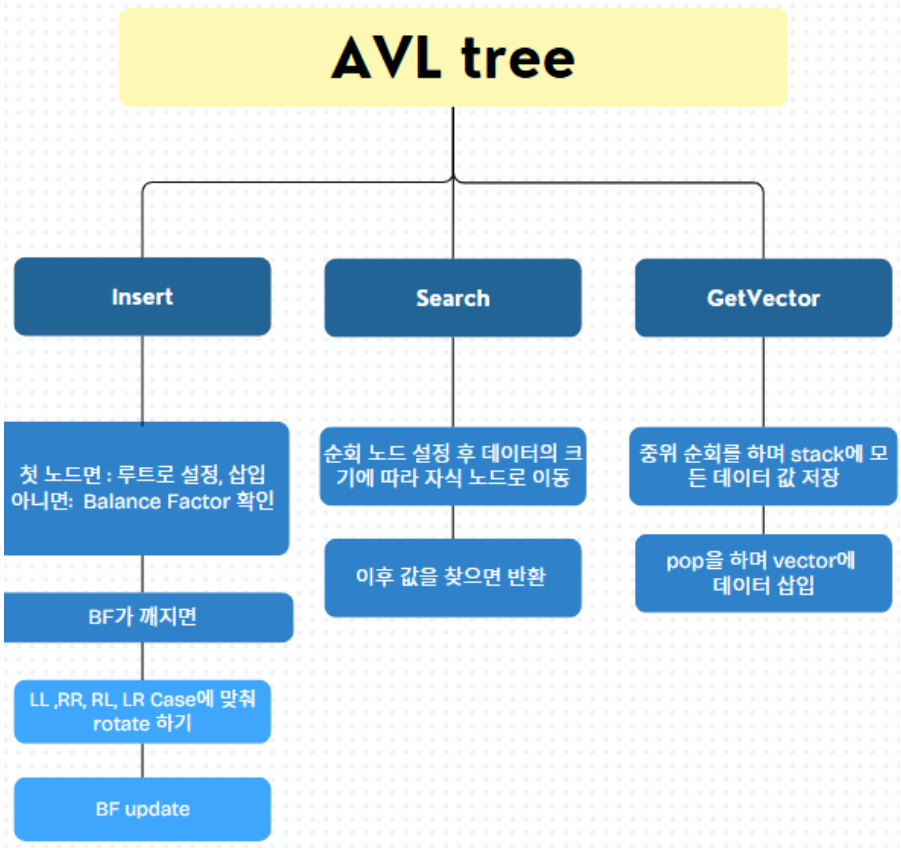
이번 프로젝트는 항공권 정보를 저장하고 Bp tree, Avl tree, vector를 이용하여 데이터를 저장하고 검색할 수 있는 프로그램을 만드는 것이다. 이 프로그램에서는 그동안 우리가 많이 써보지 않은 vector, map, iterator를 이용하여 프로그램을 설계해야 하기 때문에 이에 대한 지식을 쌓는 것이 중요한 프로젝트였다. 그리고 이번 데이터 구조에서 처음 배운 avl, bp tree를 처음 구현하는 만큼 난이도가 있는 프로젝트라고 할 수 있다.

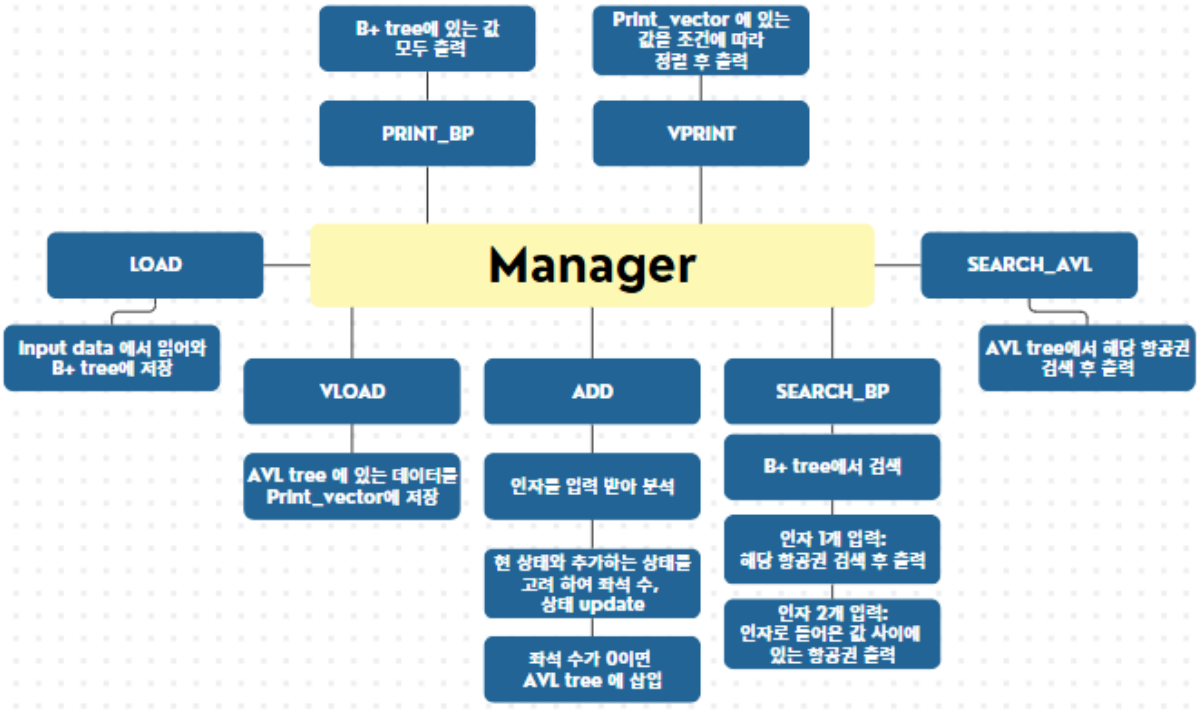
Bp tree에서는 leaf노드만 데이터를 가지고 있는 노드이고, 차수가 3인 tree를 구현하기에 overflow가 나면 분할하는 로직을 구현하는 것이 중요한 부분이다. Avl tree는 Banace Factor를 유지하며 이진트리의 모양을 갖춰야 하기에, BF를 계산하고, 균형이 깨지면 여러 case를 통해 다시 균형을 맞추는 로직을 구현하는 것이 가장 중요할 것이다. 그리고 manager함수에서는 텍스트에서 데이터를 읽어와 여러 명령어를 처리하는데,

먼저 LOAD 명령어 실행 시 이번 프로젝트에서 데이터 정보로 쓰이는 항공권 정보 데이터를 텍스트에서 읽어와 Bp tree 에 저장한다. 이때 초기 조건은 pdf를 참고하여 input.txt를 작성하면 된다. 상태 정보는 총 4개가 있는데, 이는 ADD 명령어를 통해 업데이트가 가능하고, 이에 따른 조건도 많아 이를 빠짐없이 구현하는 것이 ADD에서의 포인트이다. ADD명령어에서 좌석 수가 0인 항공편이 나온다면 이는 Avl tree에 삽입을 한다. 정렬은 항공편명을 기준으로 정렬을 하고, 이를 Vector에 삽입하여 데이터에 접근할 수 있는 로직을 구현하면 된다. 이외의 Manager함수의 명령어들은 각 Bp, Avl의 기능을 잘 이용하여 구현하면 되는 프로젝트이다.

- Flow Chart:


B+ tree
Insert
Search
Print
첫 노드면 : 루트로 설정, 삽입
아니면: search하여 삽입할 곳 검색 후 삽입
노드가 overflow 되는지 확인
오버 플로우라면 :
데이터 노드를 분할 후 새 노드를 만들고, 차수가 over되면 부모 노드로 올려 차수를 맞춤
순회 노드 설정 후 iterator를 통해 순회
조건에 따라 맞는 값을 찾아서 계속 이동(범위로 탐색)
이후 값을 찾으면 반환
순회 노드 설정 후 iterator를 통해 순회
모든 값 출력


AVL tree
Insert
Search
GetVector
첫 노드면 : 루트로 설정, 삽입
아니면: Balance Factor 확인
BF가 깨지면
LL ,RR, RL, LR Case에 맞춰 rotate 하기
BF update
순회 노드 설정 후 데이터의 크기에 따라 자식 노드로 이동
이후 값을 찾으면 반환
중위 순회를 하며 stack에 모든 데이터 값 저장
pop을 하며 vector에 데이터 삽입


B+ tree에 있는 값 모두 출력
Print_vector 에 있는 값을 조건에 따라 정렬 후 출력
PRINT_BP
VPRINT
Manager
LOAD
SEARCH_AVL
Input data 에서 읽어와 B+ tree에 저장
AVL tree에서 해당 항공권 검색 후 출력
VLOAD
ADD
SEARCH_BP
AVL tree 에 있는 데이터를 Print_vector에 저장
인자를 입력 받아 분석
현 상태와 추가하는 상태를 고려 하여 좌석 수, 상태 update
좌석 수가 0이면 AVL tree 에 삽입
B+ tree에서 검색
인자 1개 입력: 해당 항공권 검색 후 출력
인자 2개 입력: 인자로 들어온 값 사이에 있는 항공권 출력

- 알고리즘:

1. B+ tree

B+ tree 는 모든 key, data 가 리프노드에 모여있고, 리프 노드 끼리는 연결리스트로 되어있어 리프노드 끼리의 접근이 가능하다는 특징이 있다. 그리고 이번에 구현하는 B+ tree는 3차 tree 이므로 노드는 최대 3개부터 1개의 자식을 가질 수 있고, 노드에는 2개부터 1개의 키가 포함 될 수있다. 이 특징을 이용하여 분할 할 때 이용을 하면 된다.

먼저, 아무 것도 없는 상태(root == null) 이면 새 노드를 만들고 flight data를 입력 값으로 받아 dataMap에 삽입을 해주고 리턴을 한다. 만약 새 노드가 아니라면 현재 구성되어 있는 B+ tree에서 순회를 통해 삽입 할 값의 위치를 찾아주어야 한다. B+ tree도 이진트리의 일종이기 때문에 작은 값은 왼쪽 자식, 클수록 오른쪽 자식으로 구성이 되어있다. 이번 프로젝트에서는 차수가 3인 트리이기 때문에 데이터 값이 최대 3개가 들어가 구성이 될 수 있고, 키 값은 최대 2이다. 이를 이용하여 원하는 위치를 찾았다면, excessdata Node 와 excessIndexNode 함수를 이용하여 최대 수를 넘지 않는지 확인해준다.

최대 수를 넘지 않았다면 그대로 그 자리에 삽입을 해주면 되는데, 만약 최대 수를 넘었다면 분할 해주는 과정을 거쳐야한다. 이때 분할 과정은 먼저 새로운 노드를 생성 후 기존 노드의 가운데의 값을 찾아 이를 기준으로 데이터를 이동해준다. 그 후에 data노드의 연결을 Update 시켜준다. index노드가 넘쳤다면, 새로운 인덱스 노드를 생성 후 중간 키를 부모로 이동시키고, 자식 노드들의 부모 포인터를 업데이트 시켜주면 된다.

B+ tree는 leaf 노드에만 data가 들어가 있고, 그 부모는 자식을 가르키는 포인터와 키 값을 가지고 있는 노드로 구성이 되기 때문에 이를 유의하여 data노드와 index노드가 각각 최대 가질 수 있는 수를 넘었는지 확인을 거쳐 분할 및 삽입을 해야한다. 이때 중간 키를 찾거나, 삽입할 위치를 찾을 때에는 key값을 기준으로 찾아야하는데, Map에 저장이 되어있기에 iterator로 접근을 해야한다. 그리고 manager에서 처리해야하는 명령어 중 B+ tree에서 가지고 있는 값을 리턴해주는 함수를 구현해야 하기에 fligth data를 직접 리턴해주는 함수를 만들어준다.

2. AVL tree

AVL tree는 balance factor를 이용해 tree의 균형을 유지하는 자료구조로, 이진트리와 같은 특성을 가지고 있지만 BF를 이용하여 완전 이진 트리 형태로 유지한다는 점이 다르다고 볼 수 있다.

먼저 차수를 구하는 함수와 BF를 구하는 함수를 먼저 만들어 삽입할때에 이를 이용할 수 있게 하는데, 차수는 왼쪽자식의 끝과 오른쪽 자식의 끝 중 더 많은 노드를 가지고 있는 자식이 그 노드의 차수가 된다. 그리고 BF는 왼쪽 자식의 차수에서 오른쪽 자식의 차수를 빼주면 BF가 된다.

그 후에 Rotation함수를 구현해 주어야하는데, 균형이 깨질 경우 경우에 맞추어 회전을 진행해야 균형을 계속 유지하며 AVL의 구조를 유지할 수 있기에, Rotation함수도 삽입보다 먼저 구현을 해준다. Rotation의 경우 4가지 케이스가 있는데, RR,LL,RL,LR이 그 것이다. RR,LL rotate함수는 왼쪽,오른쪽으로 치우쳐져 있을 때 사용을 하는데, 돌리고자 하는 노드를 받아 그 자식의 연결을 바꿔 주어 구현을 하면 된다. RL,LR의 경우는 RR+LL로 이루어져있어 위에 구현한 RR,LL rotate함수를 사용해 주면 된다.

이를 다 구현했다면, 이제 삽입을 하면 되는데 삽입과 동시에 BF를 체크하고, 만약 균형이 깨졌다면 rotate를 해주어야 삽입이 마무리 되기에 삽입하는 함수를 나누어 구현하였다. Insert함수는 InsertNode함수를 호출하고, 리턴된 값을 root에 연결하여 구현하였고, InsertNode함수는 먼저 아무것도 만들지 않은 상태라면 새 노드를 만들고 리턴해준다. 만약 이미 AVL이 생성이 되어 있다면 이진트리에서 삽입하는 방식 그대로 data값이 현재 값보다 작으면 왼쪽 자식으로, 크면 오른쪽 자식으로 이동하며 위치를 찾아 삽입을 해주면 된다. 그 후에는 BF를 체크하여 각 케이스에 맞는 함수를 호출하여 균형을 맞춰주면 이를 해결할 수 있다. Insert함수를 이렇게 나누어 구현하면, Manager함수에서 삽입을 이용할 때에는 삽입할 Data만 함수의 인자로 넣어주면 내부 로직에 의해 삽입이 되기에, 코드도 보다 간단해지고 관리가 용이해진다는 장점이 있어 이렇게 구현하였다.

그 다음으로 search는 이진트리와 다른 점이 없다. 찾고자 하는 값을 기준으로 현재

값이 더 작으면 오른쪽 자식으로, 크다면 왼쪽 자식으로 이동하며 vector에 저장을 해야 하기에, 모든 모드의 값을 순회하며 vector에 저장해주는 방식을 택했다. 중위 순회 방식을 택하여 오름차순으로 vector의 값을 정렬시켜 활용하기 편하게 구현을 하였다.

3. Manager

Manager 에서는 avl, b+,vector를 이용하여 데이터를 관리할 수 있게 하였는데, 먼저 command와 log 파일을 열고, command에서 한줄 씩 읽어와 vector에 tap을 기준으로 나누어 저장을 하게 하였다. 명령어와 인자가 같이 들어올 경우, 인자는 tap으로 구분하기 때문이다. 후에 space를 기준으로 다시한번 나누어 vector에 저장해 명령어를 인식할 수 있게 하였다.

LOAD 명령어가 불릴 경우 input data.txt를 열고 안에 있는 데이터를 읽어와 B+ tree에 삽입을 한다.

VLOAD 명령어는 AVL에서 만든 vector를 불러오는 함수를 호출하여 Print_vector에 그 값을 복사하게 만들면 된다.

ADD 함수는 먼저 B+ tree에 추가하고자 하는 값이 있는지를 먼저 확인을 하고, 없다면 ADD하고자 하는 값을 그대로 B+ tree에 삽입을 해준다. 만약 값이 이미 존재한다면 조건을 따라 예외처리와 좌석 수 감소, 상태 업데이트를 해주면 된다.

SEARCH_BP 명령어는 인자를 몇 개 받냐에 따라 다르게 작동하는데, 이는 함수의 같은 이름을 쓰지만 인자를 통해 호출할 함수를 구분하는 오버로딩을 사용해 구현을 하면 된다. 그리고 string은 내부적으로 대소 비교를 할 수 있게 되어있기 때문에 따로 string을 비교하는 로직은 구현하지 않아도 사용할 수 있다.

SEARCH_AVL은 avl class 내부의 검색 함수를 통해 그 값을 반환 받아 그대로 출력하게 구현하면 된다.

VPRINT는 조건 두 개에 따라 다르게 출력이 되게 만들어야하는데, 이때 compare 함

수를 구현하여 이를 해결하면 된다. Vector 는 내부적으로 정렬을 할 수 있는 함수가 존재하는데, 이를 조건문을 추가하여 쓰면 내가 원하는 조건대로 정렬하여 사용할 수 있다. 이를 이용하여 VPRINT 내부에서 compare 함수를 호출해 정렬된 값을 차례대로 출력하면 된다.

- 결과화면

```
========== LOAD ==========
Success
=========================

==========PRINT_BP==========
7C024 | JEJU | CJU | 3 | Delayed
AK101 | AIRKWANGWOON | LHR | 6 | Cancelled
JA111 | JEANAIR | MUC | 4 | Departure
KE901 | KOREANAIR | LAX | 1 | Boarding
OZ107 | ASIANA | SIN | 1 | Boarding
========================

========== ADD ==========
KE901 | KOREANAIR | LAX | 0 | Boarding
========================

========== ADD ==========
OZ107 | ASIANA | SIN | 0 | Boarding
========================

==========SEARCH_BP==========
7C024 | JEJU | CJU | 3 | Delayed
========================

==========SEARCH_BP==========
JA111 | JEANAIR | MUC | 4 | Departure
KE901 | KOREANAIR | LAX | 0 | Boarding
========================

========== SEARCH_AVL ==========
KE901 | KOREANAIR | LAX | 0 | Boarding
========================

========== VOLAD ==========
Success
=========================

========== VPRINT A ==========
ASIANA |OZ107 |SIN |Boarding
KOREANAIR |KE901 |LAX |Boarding
========================

========== VPRINT B ==========
KOREANAIR |KE901 |LAX |Boarding
ASIANA |OZ107 |SIN |Boarding
========================

========== EXIT ==========
Success
=========================
```

Input_data :

KOREANAIR KE901 LAX 1 Boarding

ASIANA OZ107 SIN 1 Boarding

JEJU 7C024 CJU 3 Delayed

JEANAIR JA111 MUC 4 Departure

AIRKWANGWOON AK101 LHR 6 Cancelled

먼저 LOAD 명령어가 실행되면 예상 결과로는 Input data가 B+ tree에 모두 삽입이 되어야한다. 이는 PRINT_BP 명령어를 실행했을때 확인 할 수 있다. LOAD의 출력 값을 보

면 success가 출력되었으므로 B+ tree에 잘 삽입이 되었고, 이를 BP_PRINT 명령어로 출력했을 때 오름차순으로 잘 출력이 된 것을 볼 수 있다. 이를 통해 LOAD 명령어도 잘 동작을 했다는 것을 알 수 있다.

ADD 명령어를 수행했을 때 예상되는 결과로는 ADD명령어의 인자로 넣은 항공편명이 존재하지 않으면 그 데이터를 그래도 B+ tree에 삽입을 해주고, 아니라면 해당 항공편을 B+ tree에서 찾아 Status에 따라 좌석수를 감소 시키거나, 상태를 변경해주면 된다. 현재 ADD를 할때에는 해당 항공편이 존재하고, Boarding 상태에서 Boarding으로 ADD를 해주었었으므로 좌석수는 0이 되어 AVL tree에 삽입이 되고, 상태는 변하지 않게 될 것이다. 출력 log를 보면 항공편이 존재해 BOADRDING -> BOARDING 이므로 좌석 수가 1 감소하였고, 좌석 수가 0이 되었으므로 AVL tree에 삽입이 되었을 것이다. 이후에 BP_PRINT를 하였을 때도 ADD한 값이 잘 적용이 되어 출력 된 것을 볼 수 있다.

SEARCH_BP 를 했을 때 첫번째는 해당 항공편을 잘 찾아 출력이 된 것을 볼 수 있고, 두 번째는 항공편이 C로 시작하고 K로 시작하는 것까지 잘 출력이 되었음을 알 수 있다.

현재 AVL tree에는 KoreanAir와 ASIANA가 들어가 있을 텐데, SEARCH_AVL에서 맞는 항공편을 잘 찾아서 출력했다.

VLOAD 명령어는 success가 된 것으로 보아 Print_vector에 데이터가 잘 들어갔음을 알 수있고

VPRINT도 조건 A,B에 따라 잘 출력이 되었음을 볼 수 있다.

마지막으로 EXIT까지 잘 출력이 되었음을 볼 수 있다.

- 고찰

이번 프로젝트를 진행하며 처음으로 구현해보는 데이터 구조를 다루어야 해서 이를 이해하고 구현하는데 제일 오랜 시간이 걸렸다. 그리고 tree에서 다루는 데이터를 map과 vector로 저장을 하고 이용을 해야 해서 이 둘을 공부하는 데에도 시간을 들였다. 하지만 이를 공부하며 더 많은 지식을 쌓고, 한번 직접 구현을 함으로써 자신감을 얻는 계기가 될 수 있었다. B+ tree를 구현할 때에는 분할하는 과정이 제일 복잡했는데, IndexNode와 DataNode의 역할을 분명하게 구분하여 생각하니 조금 더 편해진 것 같았다. 또한 search 과정에서 Map에 접근을 하기 위해 iterator를 처음 사용해 보았는데, 처음에는 너무 헷갈려서 계속 오류가 나고 디버깅을 수도 없이 했다. 하지만 점차 익숙해져 또 하나를 배운 것 같아 내 실력 향상에 많은 도움이 되는 것 같았다. 그리고 이번에 한 번도

써보지 않은 키워드를 많이 써봤는데, (ex: auto, typeid, advance 등) 이 키워드에 대한 정보도 세세하게 공부하고, 이를 사용해서 구현하니 생각 과정이나 코드 구현 부분에서도 좀 더 간단해진 것 같았다.

B+ tree는 이중 연결리스트를 사용하여 leaf노드간의 접근을 할 수 있게 하는 구조여서 이를 구현하기위해 전에 배운 연결리스트의 내용을 복습하는 시간도 가질 수 있어 좋았던 것 같다. 아직 많이 부족할테지만, 한번 직접 해보니 두려움은 없어진 것 같아 자신감이 조금 생겼다. AVL tree를 구현할 때에는 역시 rotate 함수를 구현하는 것에서 많은 시간을 보냈는데, 처음에 AVL의 개념을 공부하며 어떻게 회전을 하는 함수를 구현해야 할 지 막막했다. 하지만 조금 생각해보니 자식 노드의 연결을 바꿔주는 것이 곧 회전을 하는 모양과 같아 rotate를 하는 것이구나 라는 것을 깨달았다. 그 후에는 이진트리의 구조와 다를 것이 없어 구현하기는 조금 B+ tree에 비해 편했던 것 같았다.

그리고 AVL에서 balance factor를 다루며 직접 그려보고, 구조를 생각하며 구현하니 공부도 더 되고 AVL tree에 대해 더 잘 이해할 수 있는 계기가 되었다. Manager함수를 구현할 때에는 tree에 저장되어 있는 데이터가 map형식으로 구성되어 있어 이에 접근하는 것이 제일 어려웠는데, iterator와 fligthdata를 직접 반환하는 함수를 구현할 생각을 하고 이를 사용하니 간단 해져서 이후 문제들은 잘 풀어 나간 것 같다. Manager에서 제일 까다로웠던 것은 ADD함수였는데, 비교할 조건들도 많고 처리할 error도 많아 차근차근 하나씩 깊게 들여다보며 섬세함과 침착함을 배울 수 있는 부분이었던 것 같다.

항상 프로젝트를 하며 제일 어려운 부분은 헤더 파일에 있는 멤버 변수와 함수의 역할을 알아내고 어떻게 구현할 지 판단하는 점인데, 이번에는 flight data의 헤더만 있고, 소스파일은 존재하지 않아 적잖이 당황을 하였지만, 다 만들고 난 시점에서 생각하니 이런식으로 구현하는 방법도 있구나 하며 새롭게 배운 부분이 되었다. B+ tree에서도 헤더파일과 함수가 워낙 많아서 이를 전부 파악하고 구현하는 것에 애를 먹었지만, 구현을 다 하고 나니 처음에 생각했던 역할과 구현하며 달라지는 함수들도 있었고, 내 예상대로 사용하는 함수들도 있어 아직 많은 공부가 필요하다는 것을 상기할 수 있는 계기였다.

이번 과제를 구현했다고 끝이 아니라, 한 번 더 스스로 구현을 해보고 더 공부를 해 여러 데이터 구조를 익히고 내가 활용할 수 있게 내 것으로 만들자는 목표가 생긴 프로젝트였다.