데이터구조설계 프로젝트 2 보고서

2023202027 정정윤

**Introduction**

2차 프로젝트에서 만드는 것은 항공권 정보 검색 시스템으로, 공항 등지에서 항공권 정보를 저장하고, 관리하며, 필요할 때 빠르게 불러올 수 있도록 돕는 시스템이다. DB 액세스가 언제 이루어질지 모르고, 수많은 데이터들 중에서 빠르게 찾는 것이 중요하기 때문에, 주로 사전 정렬된 형태의 자료를 사용하는 경우가 많다. 따라서, 이번 프로젝트는 B+ Tree와 AVL Tree를 이용해 사전 정렬된, 이진 탐색을 통한 빠른 접근이 가능한 자료구조를 프로그래밍 하는 것이 중요한 프로젝트이다.

1차 프로젝트와 같이 명령어를 입력받고, 이에 따른 처리를 하여 log 파일에 남기는 구조이며, DB에서 자주 보이는 쿼리 처리와 유사한 처리 과정이다. 출력 시에는 원하는 데이터들을 찾아 사용자가 원하는 형태로 표현해줄 필요가 있어 print\_vector에 다양한 정렬 순서를 바탕으로 집어넣은 후 출력하도록 한다.

시스템은 우선 다양한 항공권 정보를 가진 데이터를 입력 파일에서 받아 B+ Tree에 저장한다. 입력되는 정보는 항공사명, 항공편명, 도착지, 좌석 수, 상태 정보가 포함되어 있으며, 이후 몇몇은 정렬 기준으로, 몇몇은 시스템 처리의 기준이 된다.

항공권 마다 줄바꿈이 이루어지며, 항공권 정보 사이에는 스페이스 바를 통한 공백이 하나 있어 이를 split하는 능력을 요구하기도 한다. 각 항공편에는 몇몇 조건들이 있다. 항공사에 따른 좌석 수 제한이 있어 좌석 수가 제공되지 않고, 항공사만 제공되는 다른 명령어가 사용되면, 이 수치들을 바탕으로 좌석 수를 추가해야 한다. 또, 항공편의 상태 정보를 통해 항공편에 대한 요구가 왔을 때, 좌석 수를 감소 시키거나 감소시키지 않는 처리를 진행해야 한다.

구현하는 B+ Tree의 차수는 3으로 노드는 최대 2개의 값을 가지고 있을 수 있다. 일반적으로 B Tree는 [주소, 인덱스, 주소, 인덱스, 주소]로 양끝에 주소를 두고 이를 기준으로 불러오는 방식을 취하는 경우가 있으나, 본 프로젝트에서는 스켈레톤 코드에서 사용한 방식인 가장 왼쪽 자식 주소를 저장한 후, [인덱스, 주소] 벡터를 다수 저장하는 구조를 가지고 있다. 이 저장에는 map을 사용하여 알아서 내부에서 인덱스를 기준으로 정렬되도록 한다.

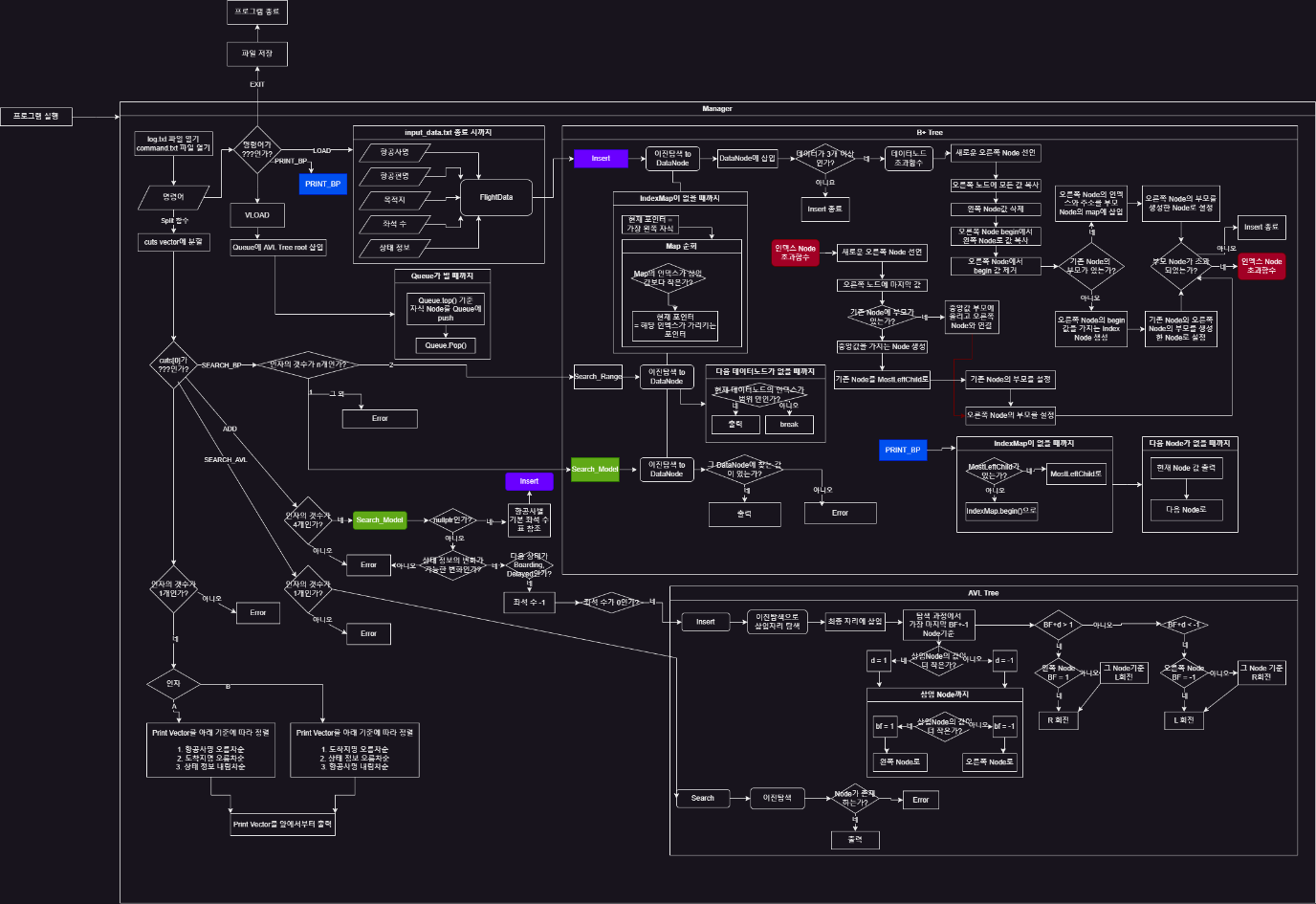
B+ Tree의 정렬은 항공편명을 기준으로 이루어지므로, 즉 인덱스는 항공편명, 데이터는 항공편 정보가 된다.

이후 명령에 따라 B+ Tree에 있는 항공편 데이터들을 관리하며, 항공편의 좌석 수가 0이 될 경우, AVL Tree로 옮기도록 한다. AVL Tree의 데이터는 [인덱스, 항공편 정보]의 구조를 가지고 있으며, 인덱스는 여전히 항공편명이다. AVL Tree에 데이터를 넣을 때는 깊이를 기록하는 방식이 아닌 각 Node에 Balance Factor를 기록해두고 사용하는 방식으로 구현한다.

명령어는 LOAD를 통해 입력 파일로부터 항공편 정보들을 가져오는 것으로 시작하여, ADD 명령어를 통해 입력 파일이 아닌 명령어로부터 직접적으로 추가하거나, 항공편을 관리할 수 있고, SEARCH\_BP 명령어를 통해 B+ Tree에 저장된 항공편의 데이터들을 확인할 수 있다. 단일 데이터를 확인하거나 여러 데이터들을 모아 확인하는 기능 또한 구현해야 한다. SEARCH\_AVL 명령어를 통해 AVL Tree에 저장된 데이터를 확인할 수도 있다. 이후에는 VLOAD를 통해 AVL Tree에 저장된 값들을 Print\_Vector로 옮긴 후, VPRINT를 통해 특수하게 정렬된 기준에 따라 정렬하고 출력한다.

LOAD 명령을 제외한 다른 명령들은 여러번 수행되어도 큰 문제가 발생하지 않는다. 명령을 모두 수행한 후에는 프로그램을 종료하는 EXIT명령어를 사용할 수 있다.

**Flowchart**



이진탐색 또는 split함수, 라이브러리 내장함수를 사용하는 경우, 이미 작성된 순서도거나, 내장함수 내 알고리즘과 일치하지 않을 수 있으므로, 순서도를 작성하지 않고 언급만 하였다.

Manager 내부의 클래스들과 이어져있는 방식으로 순서도를 작성하여 명령어 수행과 Manager 내부 변수들의 값 호출에 대해 표현하였다. 멀리있는 같은 기능의 경우 색깔로 표시하여 중복 내용의 기재와 불편함을 해결하였다.

**Algorithm**

입출력

Manager에서 명령어의 입력을 위해서는 프로젝트 1에서 사용했던 split 함수를 그대로 사용하였다. 구분자는 공백에서 \t(탭)으로 바뀌었지만, 구분자 지정이 가능한 함수이기에 그대로 사용하여도 문제없이 작동한다. 항공편 데이터의 입출력에는 split함수를 사용하지 않고, stream을 이용해 입출력을 받도록 하였다. 입력 중간에 자료형이 다른 입력이 있어, split 함수를 사용하지 않았다.

BpTree

스켈레톤 코드에서 삽입 과정에서 사용되는 코드가 제공되었다. 노드에 이미 공간이 없는 경우를 표시하는 함수가 있었고, 노드를 분할하는 함수 또한 기본 형태만 제공되었다. B Tree의 경우 기본적으로 모든 Leaves의 Degree가 같기 때문에 이를 구현하기 위해 새로운 값이 들어올 때, 이를 올바른 위치에 넣은 후, Node가 넘친다면 그 Node를 쪼개고 부모 Node로 올려버린다. 이후 부모 Node에서도 넘칠 경우, 이를 반복한다. 반복으로 조건 확인 후 수행하여도 되긴 하지만, 스켈레톤 코드에서 함수 형태로 제공되었기에 재귀적인 방식으로 구현하도록 하였다. BTree와 다르게 BpTree는 데이터노드 Leaves가 존재하므로, 데이터 노드를 나누는 함수와 인덱스 노드를 쪼개는 함수가 따로이며, 데이터 노드를 나눈 후에는 인덱스 노드를 확인하는 함수가 실행된다.

따라서, 삽입 시에는 먼저 이진 탐색을 통해 올바른 데이터 노드까지 값을 이동하여 삽입하고, 조건 함수를 통해 해당 노드를 나눌 필요가 있는지 조건을 확인한다. 차수가 3이므로, 한 노드에 값이 3개 들어있게 된다면 나눠야 한다.

데이터 노드를 분할하는 함수는 오른쪽 노드를 새로 생성하고 데이터노드의 특성에 맞추어 왼쪽 노드와 양방향 연결 리스트로 연결한다. 이후 오른쪽 노드에 왼쪽 노드의 데이터를 모두 복사한다. 이후 왼쪽 노드를 clear하고 오른쪽노드에서 첫 번째 값을 복사하고 그 노드의 첫 번째 값을 지운다. 기본적으로 데이터노드 분할 시 1/2로 분할되고, 중간값이 인덱스 노드로 가며, 양쪽을 가리키게 되기 때문에 하나씩 복사하여 옮기는 것보다 삭제 개수나 값 가져오는 방식이 명료하다. 이후 부모 노드로 올리는 과정이 필요한데, 이에 부모 노드가 존재하는지 확인하고, 존재하지 않다면 기존이 root이므로, 새로운 Node를 생성해 왼쪽Node는 MostLeftChild로, 오른쪽 노드는 map에서 이은 후 root로 만들어준다. 만약 기존 부모가 있다면 올린 후, 오른쪽 노드만을 이어주게 된다. 기존 Node를 가리키고 있던 인덱스 노드가 이미 존재하기 때문에, 오른쪽 노드를 가리키는 Node만이 새로 요구되기 때문이다. 이후 해당 노드의 초과 여부를 확인하고 초과하였다면 그 노드를 기준으로 인덱스 노드 분할 함수를 호출한다.

인덱스 노드 분할 시에는 데이터노드와 다르게 3개 중 중간값을 부모로 올리고, 양쪽 값은 남기게 된다. 따라서, 하나씩 값을 돌며, 첫 값은 미리 저장, 세번째 값은 따로 오른쪽 노드 생성 후 넣어준다. 이후 다시 순회하며, 중간값위치에 반복자를 두고, 부모 노드가 존재하는지 확인한다. 존재할 경우, 그 값을 부모노드로 올려주고, 그 값에 대해 오른쪽 노드를 이어준다. 만약 존재하지 않는다면, 새로 부모노드를 생성하고 기존 Node를 MostLeftChild로 이어준 후, 오른쪽 노드를 map에서 이어준다. 이후 부모노드에서 초과 여부를 확인하고 초과하였다면 그 노드를 기준으로 인덱스 노드 분할 함수를 호출한다.

탐색 시에는 이진 탐색 후 해당 데이터노드에 값이 있는지 확인한 다음 그 데이터노드를 반환하는 함수를 사용하였다. 범위탐색의 경우, 해당 데이터노드를 시작으로 데이터노드들이 서로 양방향 연결리스트처럼 이어져 있다는 사실을 통해 시작 지점부터 해당 범위 종료 시까지 탐색을 수행하도록 하였다.

AVL Tree

AVL Tree는 데이터를 삽입할 때, 균형이 맞지 않는다면 회전을 수행하여, 탐색 시간이 밑이 2인 log N이 되도록 하는 트리다. BpTree에서 항공편의 좌석 수가 0이 되면 본 트리로 복사하게 된다.

삽입

AVL Tree는 균형잡힌 이진 트리이기에 탐색 과정은 이진 트리에서 요소를 찾는 것과 같은 방식으로 수행한다. 따라서, 삽입과 문제에서 요구된 GetVector 함수의 구현만 보고서에 기입하도록 한다.

삽입 시, AVL Tree는 이진 트리 구성 규칙에 따라 신규 값을 적절한 위치까지 가져간다. 이후, 해당 값의 노드를 이은 후, 트리의 균형이 맞는지 확인한다. 확인하는 방법은 다양한 것이 있지만, 본 문제에서는 각 Node에 Balance Factor를 저장하는 방식을 사용하고 있다. 마지막으로 균형이 +-1인 Node부터 이진 탐색에 따라 삽입된 Node까지 이동한다. 이동하며, 각 Node의 균형 또한 추가된 Node의 방향에 따라 BF값이 변경된다.

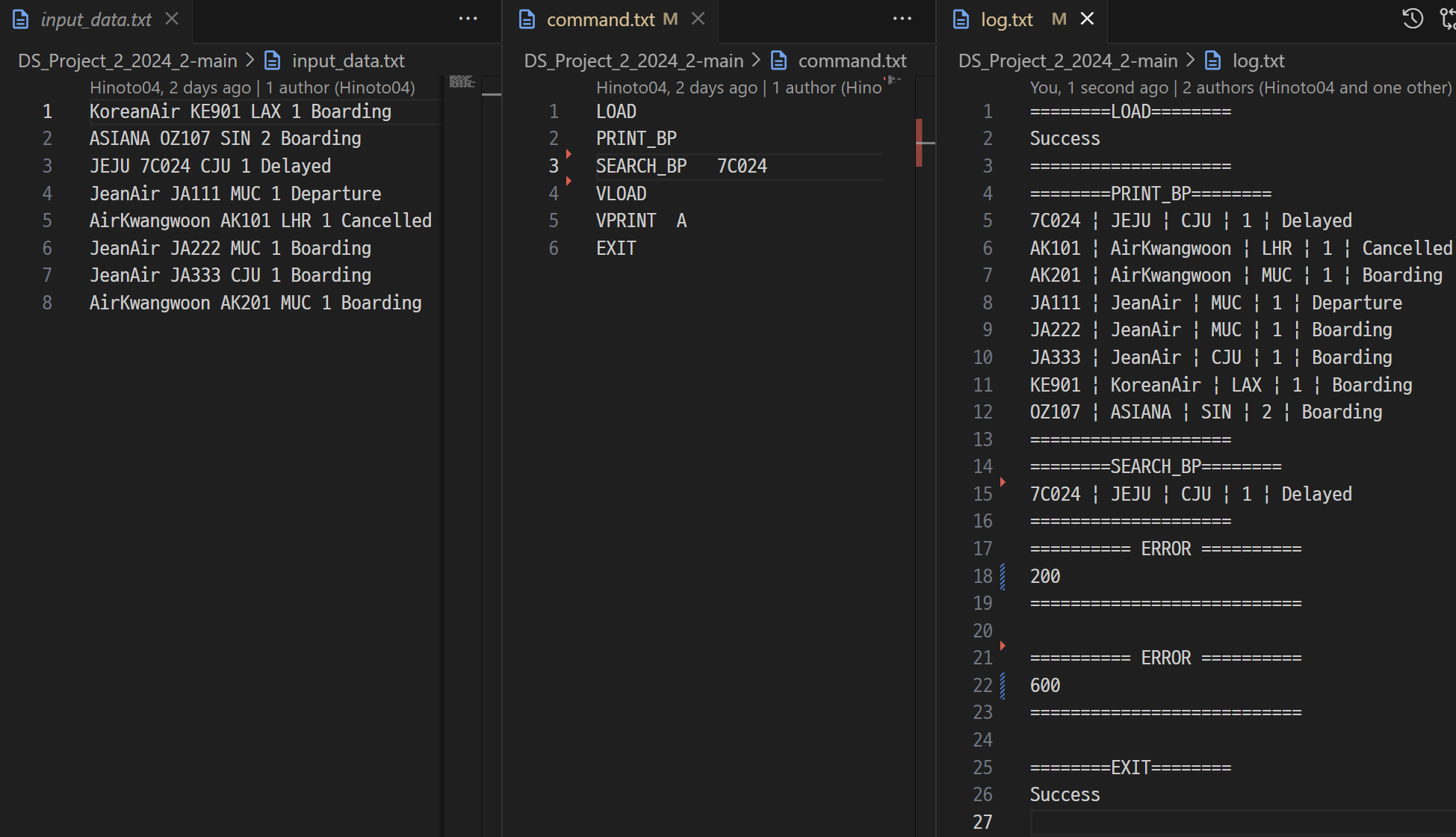
만약 처음으로 BF가 +-1인 Node에 새 Node방향과 같은 방향을 더했을 때, 2가 된다면 그 노드는 균형이 잡혀있지 않은 것으로 회전을 요구한다. 회전 시에는 회전을 하는 노드와 회전 방향과 순서에 따라 (LL, LR, RR, RL) 다음(2번)과 다다음(3번) 노드의 주소를 미리 저장해둔다. 이후, LL, RR이라면 2번 Node를 부모로 잡고, 다른 노드들을 자식으로 설정한 후, 1번 Node의 자식중 원래 2번 Node가 있던 자리에 2번 Node의 원래 자식 중 가까운 쪽을 할당한다. LR, RL의 경우에는 3번 Node를 부모로 잡고, 2번 Node와 원래 Node중 2번 Node에서 먼 쪽을 자식으로 설정한다. 이후 2번 Node 기준으로 3번 Node가 있던 자리에 3번 Node의 자식 중 가까운 쪽을 할당한다. 이 회전 과정을 통해 AVL Tree는 항상 균형잡혀 있을 수 있다.

GetVector 함수는 본 문제에서 제시된 기능올, AVL Tree에서 값을 Print\_Vector로 옮기는 기능이다. 단지, 재귀로 구현해서는 안되며, Queue, Stack STL을 사용해야 한다. 작성한 프로그램에서는 Queue를 사용하여 구현하였다.

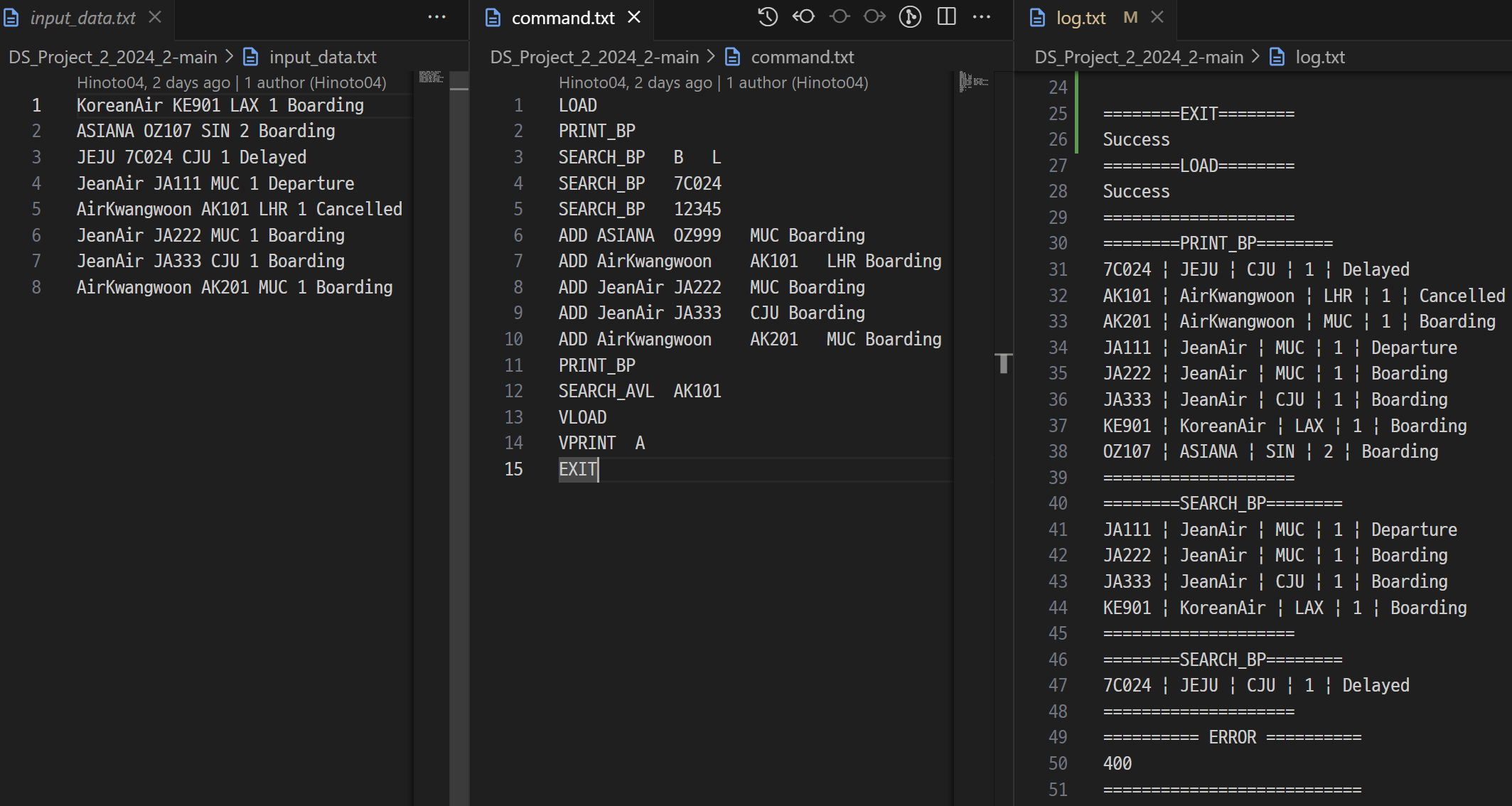
먼저 Queue에 root를 넣는다. 이후 Queue에서 요소를 빼고, 그 자식들 중 NULL이 아닌 것들을 Queue에 넣는다. 이를 반복하면, AVL Tree내의 모든 요소를 Print\_vector에 복사할 수 있다. 이후 Print\_Vector는 정렬을 요하는데, 제공된 라이브러리에 있는 sort에 비교 기준 함수를 따로 넣어 3가지 조건 하의 정렬 방식을 구현하였다.

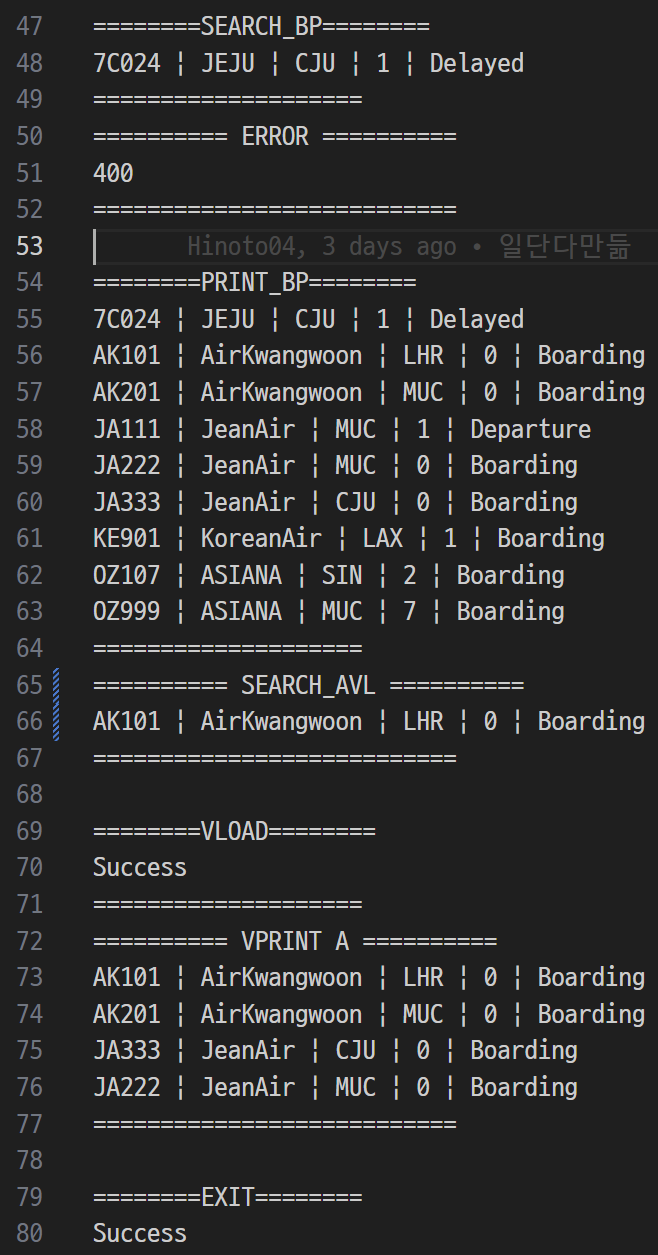
Print\_Vector는 진짜 출력하는 것 뿐인 Vector와 같기 때문에 따로 알고리즘으로 보고서에 적지는 않는다.

**Result Screen**

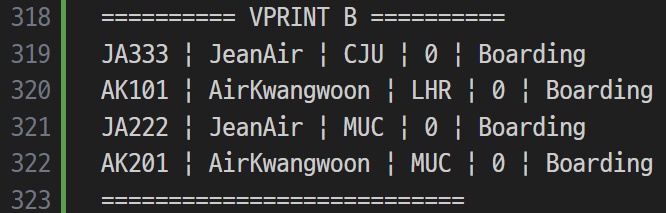


BP 삽입 및 검색 테스트수행 결과이다. ADD 명령을 수행하지 않았기에 AVL Tree에 값이 들어가지 않으며, VLOAD와 VPRINT는 AVL Tree에 값이 없으므로, 오류가 난다. LOAD, PRINT\_BP, SEARCH\_BP는 정상적으로 수행되고 Error 번호가 출력되지 않음을 확인할 수 있다.



ADD 명령을 추가한 형태이다. 기존 명령의 로그를 지우지 않고 작업하게 될 경우, 문제의 요구사항에 따라 기존 log에 이어져 작성된다.

좌석 수가 0이 된 데이터들이 AVLTREE로 이동하여, SEARCH\_AVL과 VLOAD, VPRINT가 정상 작동함을 알 수 있다.



VPRINT B로 설정하여도 우선순위에 맞게 출력됨을 볼 수 있다.

**Consideration**

BpTree와 AVL트리 구현은 처음 이었다 보니, 특히 BpTree는 스켈레톤 코드에서 제시된 구현방식도 다른 구현방식도 제대로된 코드로 짜본 경험이 부족하다보니, 스켈래톤 코드를 통한 코드 작성 방향 제시를 이해하는 것에 상당히 많은 시간이 들어갔다. cpp에서 구현하기 위해서는 어쩔 수 없는 방향이라고는 생각하고 있으나, 단지 길이가 1 더 긴 다른 벡터에 값을 저장해도 되지 않았을까 생각하기도 한다. 차수가 한정되었으니, 아예 Node 자체를 비슷한 구조로 설정하는 것도 방법이라고 생각한다. Node에 상위 Node의 분할 함수 호출 등이 문제가 되는 경우를 생각해볼 수 있으나, 시도해볼 가치는 있지 않을까 생각한다.

BpTree 구현 처음에는 IndexNode에 Vector를 넣으며, 해당 Node에 들어있는 인덱스와 그 주소, 그리고 자식 Node의 값과 주소를 적어 각 IndexNode는 차수 m에 대해 2m-1의 길이로 선언을 하여 홀수번째에는 자식노드의 주소, 짝수에는 그 자신 Node의 값에 대해 기록되어 있는 형태를 생각하였으나, Node가 초과되는 조건을 확인하는 함수에서 스켈레톤 코드의 의도를 알아차린 바가 있다.

AVL Tree의 경우, 강의자료의 구현을 많이 참고하였는데, 원래 회전의 구현을 함수화하여, LR, RL같은 상황의 경우 아래에서 한 번, 위에서 한 번 서로 반대방향의 회전함수를 호출하는 것으로 구현하고자 하였으나, 생각보다 Node를 다시 트리에서 찾고 돌리는 과정에서 생기는 문제 등이 있어, 강의자료와 같은 조건문을 통한 상황 구별과 미리 회전에 필요한 Node들의 주소 기록을 통한 거의 한 번에 회전하는 방식을 사용하였다.

스켈레톤 코드에서 제시된 virtual function 중 makefile을 통한 build 시, pragma once가 아닌 warning을 항상 발생시키는 코드가 있었다. 반환값이 지역 변수의 주소이기 때문인데, 이 경우 결국 nullptr을 반환한 것과 같은 효과를 보이므로, 해당 함수의 반한값을 nullptr로 설정하고, 해당 warning이 많이 표시되지 않도록 하였다.

GetVector 함수의 구현 시, 재귀 사용 불가가 조건으로 걸려있어, Tree도 결국 Graph라는 사실에 착안해 BFS와 같은 방식을 통한 구현을 아이디어로 하여 구현하였다. 잘 작동하는 것으로 보아 BFS를 통한 전체 탐색이 Tree에서 잘 작동함을 알 수 있었다.

ADD 명령 구현 시, 정해진 4개의 경우 외에는 모두 Block하라는 것으로 이해하였고, 상태 정보 업데이트 부분에 적혀있는 Cancelled -> Boarding에서만 감소하라는 것으로 이해하였었으나, 항공권 정보 데이터에 있는 상태 정보 부분에서 이가 단순히 예시라는 사실을 확인할 수 있었다. 구현 시에는 전제 조건을 먼저 확실히 한 후에 구현해야 함을 알 수 있었다.