

데이터 구조 설계 2차 프로젝트 보고서

2021202085 전한아솔

1. Introduction-프로젝트 소개

이번 프로젝트는 B+ Tree, AVL Tree, STL vector를 이용하여 항공권의 정보를 저장, 검색, 출력하는 시스템을 구현하는 프로젝트이다.

항공권 정보 데이터는 항공사 명, 항공편 명, 도착지, 좌석 수, 상태 정보를 포함한다.

먼저, LOAD 명령어는 input_data.txt에 저장된 항공권 정보 데이터를 불러와서 B+ Tree에 저장하는 역할을 한다.

VLOAD 명령어는 AVL Tree에 저장된 모든 데이터 정보를 print_vector(STL vector)에 불러오는 명령어로, 재귀를 사용하지 않고 queue나 stack을 사용하여 벡터에 저장한다.

ADD 명령어는 인자로 전달된 항공권의 정보를 B+ Tree에 직접 추가하기 위한 명령어로, 항공사 명, 항공편 명, 도착지, 상태를 입력받는데, 해당 데이터의 항공편 명이 B+에 이미 존재한다면 상태에 따라서 좌석 수를 하나 감소시킬지를 결정한다. B+ 트리에 존재하지 않는다면 새롭게 B+ 트리에 삽입을 진행하며, 이때 좌석 수는 항공사에 맞는 최대 좌석 수로 고정한다.

좌석 수가 감소하는 상황은 Canceled -> boarding, Boarding -> boarding, Delayed -> delayed 총 3가지의 형태일 때이고, Boarding -> Canceled는 좌석 수가 감소하지 않고 상태만 변경한다. 좌석 수가 감소하는 3가지의 형태도 상태가 바뀌면 상태 정보가 바뀔 수 있다. 이 4가지의 상황을 제외하고는 모두 에러를 출력해야한다.

SEARCH_BP 명령어는 B+ 트리에 저장된 데이터를 검색하여 출력하는 명령어로 2가지 형태가 존재한다. 먼저, 인자가 1개인 경우는 해당 항공편 명과 동일한 데이터 정보를 출력한다. 인자가 2개인 경우는 해당 알파벳으로 시작하는 모든 항공권을 출력한다. 예를 들어서 C와 K가 인자로 입력되면 C~K로 시작하는 항공편 명을 가진 모든 항공권을 출력한다.

SEARCH_AVL 명령어는 인자로 전달된 항공편 명과 동일하고 AVL 트리에 존재하는 항공편 명의 항공권 정보를 출력하는 역할을 한다.

VPRINT 명령어는 2가지 조건 중 하나를 택하여 벡터를 정렬하고 출력한다.

먼저 A 조건은 1. 항공사 명 오름차순, 2. 항공사 명이 동일하다면 도착지 명 오름차순, 3. 도착지 명도 동일하다면 상태 정보 내림차순으로 정렬하고 출력한다.

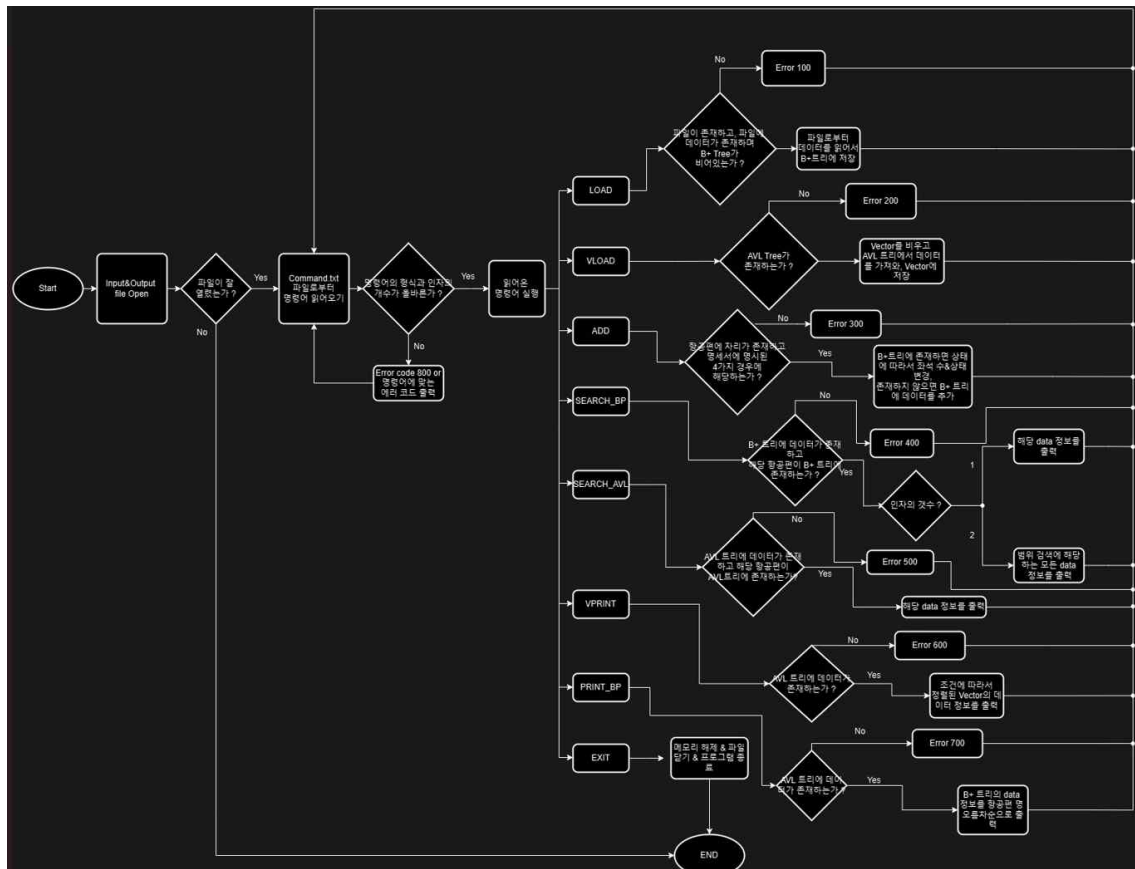
B 조건은 1. 도착지 명 오름차순, 2. 도착지 명이 동일하다면 상태 정보 오름차순, 3. 상태 정보도 동일하다면 항공사 명 내림차순으로 정렬하고 출력한다.

PRINT_BP 명령어는 B+ 트리에 저장된 데이터를 항공편 명 기준, 오름차순으로 전부 출력하는 명령어이다.

EXIT 명령어는 메모리를 해제하고 프로그램을 종료하는 명령어이다.

이렇게 총 7개의 명령어를 수행할 수 있고 B+ 트리와 AVL 트리, 벡터를 사용하는 프로젝트를 구현하면 된다. 이번 프로젝트도 1차 프로젝트와 마찬가지로 파일 입·출력을 사용하고 명령어에 입력하는 인자 개수가 맞는지 않은 경우나 잘못된 명령어를 입력하는 경우, 에러를 출력한다. 또한 과제 명세서에 명시된, 명령어마다 에러를 출력하는 조건이 존재하는데, 해당 조건에 해당한다면 명령어에 맞는 에러 코드를 출력하도록 구현한다. 또한 메모리 누수가 발생하지 않도록 동적으로 할당한 메모리를 모두 해제하는 것도 잘 고려해서 구현한다.

2. Flowchart-프로젝트의 전반적인 흐름과 동작을 설명



파일이 제대로 열리지 않는다면 프로그램은 종료된다. 제대로 파일이 열렸다면, Command.txt 파일로부터 명령어를 한 줄씩 읽어와서 해당 명령어를 실행한다. 이때, 명령어가 틀렸거나 인자의 개수가 맞지 않는다면 명령어를 실행하지 않고 해당 에러에 맞는 에러 코드를 출력하고 다음 명령어를 실행하도록 했다.

제대로 인자와 형식이 맞아서 명령어를 실행하면, 2차로 명령어가 제대로 실행될 조건을 확인

한다. 이때 조건에 부합하지 않으면 각 명령어의 에러 코드를 출력한다. 조건과 부합한다면 해당 명령어에 맞는 동작을 수행하도록 구현했다.

3. Algorithm-프로젝트에서 사용한 알고리즘의 세부적인 동작과 아이디어를 설명

<B+ Tree>

B+ 트리는 균형 이진 탐색 트리의 일종으로 B 트리를 변형하여 모든 데이터를 리프 노드에만 저장하고 인덱스 노드는 오직 탐색을 위하여 사용한다는 특징을 가진다.

먼저, 데이터 노드(리프 노드)는 실제 데이터를 저장하는 노드로, 모든 데이터 노드는 포인터를 통해서 서로 연결되어 있다. 이는 양방향 연결 리스트의 형태이다. 그리고 각 리프 노드는 트리의 데이터가 정렬된 상태로 저장된다. 따라서 데이터 노드에 연결된 다른 데이터 노드를 차례대로 따라가면서 탐색한다면 빠른 속도를 보장할 수 있다.

B+ 트리에 데이터를 삽입하는 과정은 먼저 데이터 노드에 데이터를 삽입한다. 이번 과제에서 구현해야 하는 B+ 트리의 차수는 3이므로 한 노드가 가지는 데이터의 개수가 3개를 초과하면 분할이 일어나야 한다.

한 노드의 데이터가 3개가 존재하면 처음 데이터로 새로운 노드를 만들고, 현재 노드의 데이터는 삭제, 다음 데이터를 부모 노드에 삽입하고 현재 노드의 데이터를 삭제하는 방식으로 노드의 분할을 구현했다.

먼저, 데이터 노드의 분할을 설명하면, 한 데이터 노드가 가지는 데이터가 3개를 초과하면 오름차순으로 정렬된 데이터를 기준으로 왼쪽 데이터와 오른쪽 데이터는 데이터 노드로 분할되며, 중간에 존재하던 데이터는 바로 위 인덱스 노드(부모 노드)에 포함된다. 그리고 해당 인덱스 노드와 데이터 노드를 연결하는데, 이 연결하는 부분은 이후에 경우를 나누어서 설명하겠다.

인덱스 노드의 분할은 데이터 노드와 마찬가지로 데이터의 개수가 3을 넘으면 왼쪽 데이터와 오른쪽 데이터가 각각 다른 인덱스 노드로 분할되며, 중간에 존재하던 데이터는 해당 인덱스 노드보다 높이가 하나 높은 인덱스 노드(부모 노드)에 포함된다. 이렇게 3개로 나뉜 데이터가 포함된 인덱스 노드들도 새롭게 연결해주어야 하는데, 이 부분도 경우를 나누어서 설명하겠다.

이 내용은 하나의 노드가 분할되는 과정이고, B+ 트리의 모든 노드에 존재하는 데이터의 개수는 3을 초과할 수 없으므로 분할이 일어난 노드는 부모 노드의 주소를 타고 올라가서 더 이상 분할이 일어나지 않을 때까지 반복적으로 분할을 진행하여 한 노드의 데이터 개수를 조절한다.

B+ 트리에 삽입하는 로직은 여러 가지 경우를 가진다. 먼저 첫 번째는 루트가 비어있는, 즉

B+ 트리가 비어있는 경우이다. 해당 경우는 B+ 트리가 존재하지 않으므로 노드를 동적으로 할당하여 데이터를 삽입하고 해당 노드는 root로 지정한다.

만약 데이터의 개수가 3을 초과한다면 분할이 되어야 하는데, 앞서 설명했던 방식으로 분할을 진행하고 위로 승격된 인덱스 노드는 새로운 root 노드가 되며 새로운 root의 LeftChild 노드는 왼쪽 데이터를 가지는 노드, 새로운 root의 데이터에서 포인터로 가리키는 노드는 오른쪽 데이터를 가지는 노드가 된다.

B+ 트리가 비어있지 않다면 루트가 데이터 노드인지, 인덱스 노드인지에 따라서 나뉜다. 루트가 데이터 노드라면 루트 노드에 데이터를 삽입하고 데이터 노드에 존재하는 데이터의 개수를 확인한다. 인덱스 노드와 데이터 노드는 모두 BpTreeNode 클래스를 상속받아서 구현했는데, BpTreeNode 클래스에 멤버로 bool형 변수를 하나 선언하여, 해당 노드가 데이터 노드라면 true, 인덱스 노드라면 false를 가지도록 각 클래스의 생성자에서 코드를 작성했다. 따라서 이를 이용하여 루트 노드의 종류를 알 수 있도록 구현했다.

루트 노드가 인덱스 노드인 경우는 루트 노드부터 시작해서 현재 노드가 가지는 map의 원소와 삽입할 데이터를 비교하여 데이터 노드에 도달할 때까지 내려가, 알맞은 데이터 노드에 삽입을 해주어야 한다.

삽입할 데이터가 현재 노드(인덱스 노드)의 첫 번째 원소보다 작다면 LeftChild로 이동한다. 앞선 상황에 해당하지 않을 때, iterator를 이용하여 map을 순회한다. begin부터 시작하여 노드가 가지는 원소가 1개라면 남은 경우의 수는 노드가 가리키는 포인터(자식 노드)로 이동하는 방법뿐이다. 하지만 노드가 가지는 원소가 2개라면 iterator를 증가시켜 2번째 원소로 이동한다. 해당 원소와 비교하여 삽입할 데이터가 더 크다면 2번째 원소가 가리키는 포인터로 이동한다. 이 경우도 아니라면 3개의 자식 중 가운데 자식으로 이동해야 하므로 iterator를 하나 감소하고 iterator가 가리키는 포인터로 이동한다. 이런 방식으로 데이터 노드에 도달할 때까지 반복하다가 도달한 데이터 노드에 데이터를 삽입하면 된다.

이렇게 데이터를 삽입하고 난 후 데이터의 개수를 확인하여 노드를 분할 한다.

루트 노드가 인덱스 노드인 상황에서 데이터 노드를 분할 할 때에는 내가 분할 할 데이터 노드의 prev가 nullptr인지 아닌지에 따라서 나뉜다.

먼저 prev가 nullptr이라면 새로운 데이터 노드의 next를 현재 데이터 노드로 연결하고 현재 데이터 노드의 prev를 새로운 데이터 노드로 연결한다. 만약 아니라면 새로운 데이터 노드는 현재 데이터 노드와 현재 데이터 노드의 prev 사이에 연결되도록 구성한다. 이렇게 데이터 노드가 분할되면 중간 데이터를 인덱스 노드(부모 노드)에 추가해야 한다.

인덱스 노드에 추가하는 상황도 데이터를 탐색하는 방법과 비슷하게 3가지로 나뉜다. 추가해야 할 데이터가 노드의 첫 번째 원소보다 작은 경우(왼쪽에 추가하는 경우라고 표현), 두 번째 원소보다 큰 경우(오른쪽에 추가하는 경우라고 표현), 둘 다 아닌 경우(중간에 추가하는 경우라고 표현)로 생각할 수 있다.

먼저, 왼쪽에 추가하는 경우는 LeftChild를 새로운 데이터 노드에 연결되도록 바꾸면 된다.

오른쪽에 추가하는 경우는 추가하기 전, 마지막 원소가 가리키던 포인터를 새로운 데이터 노드로 바꾼다. 새로 추가되는 데이터는 second로 현재 데이터 노드의 주소를 가리키고 있으므로 따로 변경할 필요가 없다.

마지막으로 중간에 추가하는 경우는 노드의 첫 원소가 가리키는 포인터가 현재 노드인데, 이 포인터를 새로운 데이터 노드의 주소를 가리키도록 하면 된다. 데이터 노드들 사이의 next와 prev는 인덱스 노드의 추가하기 전에 모두 고려하여 포인터를 수정하였으므로 고려하지 않아도 된다.

이렇게 데이터 노드를 분할하고 여러 가지 경우에 맞추어 인덱스 노드에 데이터까지 삽입했다면 해당 인덱스 노드를 분할 할지 판단한 후 분할을 진행한다. 데이터 노드를 분할하는 코드에서 while을 이용하여 노드가 nullptr이 될 때까지, 부모 노드를 타고 올라가서 반복하며 인덱스 노드의 분할을 확인하고 처리하도록 구현했다.

공통으로 적용되는 코드를 먼저 작성한 후 경우를 나누었으므로 먼저 적용되는 코드 부분의 로직을 설명하겠다.

현재 노드의 첫 번째 원소로 새로운 인덱스 노드를 만든다. 이 새로운 인덱스 노드의 LeftChild는 원래 현재 인덱스 노드가 가지는 LeftChild로 설정하고 LeftChild의 부모를 새로운 인덱스 노드로 설정한다. 그리고 현재 인덱스에서 첫 번째 원소를 삭제하도록 해서 초기의 공통으로 적용되는 코드를 작성했다.

인덱스 노드가 분할되는 코드는 해당 인덱스 노드가 root인지, 아닌지에 따라서 나뉜다.

먼저, 루트 노드에 해당한다면 현재 인덱스 노드가 분할되어 추가로 생기는 2개의 인덱스 노드 중, 가운데 데이터를 가지는 인덱스 노드는 새로운 루트 노드가 되어야 한다. 그리고 이는 루트가 인덱스 노드인 상황에서 분할되면 새로운 루트로 인덱스 노드가 생기고 나머지 2개의 데이터 노드가 LeftChild와 root의 second가 가리키는 노드가 되는 것처럼, 새로운 루트 노드의 LeftChild는 왼쪽 데이터를 가지는 새로운 인덱스 노드가 되고, 루트 노드가 가지는 원소의 second는 현재 인덱스 노드가 된다. 루트로 승격하지 않은 두 인덱스 노드의 부모는 루트 노드가 된다.

분할되는 노드가 루트가 아닌 경우는, 데이터 노드를 분할해서 부모 노드에 추가하는 방식과 유사하게 왼쪽, 중간, 오른쪽에 추가하는 경우로 나누어 삽입한다.

먼저, 왼쪽에 추가하는 경우는 부모 노드의 LeftChild를 새로운 인덱스 노드로 지정한다.

오른쪽에 추가하는 경우는 부모 노드의 마지막 원소가 가리키는 노드를 새로운 인덱스 노드로 지정한다.

가운데에 추가하는 경우는 첫 원소가 가리키는 노드를 새로운 인덱스 노드로 지정한다.

세 경우 모두 적용되는 로직은 새로운 인덱스의 부모 노드를 현재 노드의 부모 노드로 지정한다. 그리고 부모 노드에 가운데 존재하는 데이터를 삽입한다. 이때 가운데 존재하는 데이터는 현재 인덱스 노드의 주소를 second로 가져야 한다.

아까 현재 인덱스 노드의 데이터를 하나 삭제했으므로 현재 인덱스 노드가 첫 번째 원소가

가리키는 노드를 LeftChild로 설정하고 현재 인덱스 노드의 원소를 삭제하는 과정을 통해서 인덱스 노드의 분할을 구현했다.

<AVL Tree>

AVL 트리는 Binary search tree에서 최악의 경우, 한쪽으로 치우쳐진 데이터가 삽입되어서 $O(n)$ 의 시간복잡도를 가진다. 이렇게 되면 연결 리스트와 다를바가 없게 되므로 이런 BST의 단점을 보완하기 위해서 고안된 자료구조이다.

AVL 트리는 각 노드마다 왼쪽 서브 트리의 높이-오른쪽 서브 트리의 높이를 Balance factor(BF)로 가진다. 이는 양쪽 서브 트리의 높이 차이가 1이 되도록 유지하려는 특징 때문인데, BF가 -1, 0, 1이 아닌 경우에는 트리의 균형이 깨짐, 즉 트리가 치우쳐졌음을 의미하므로 트리의 노드를 바꾸어 회전하는 과정을 거쳐서 다시 균형을 맞추도록 설계한다.

트리가 비어있다면 새로 만든 노드는 루트가 되고 삽입을 종료하고 루트가 비어있지 않다면 AVL도 BST의 일종이므로 현재 노드의 데이터와 비교하여 왼쪽 or 오른쪽 자식으로 이동한다. 이때, 노드의 BF가 0이 아니라면 이는 곧 균형이 깨질 후보임을 의미하므로 따로 저장하고(a) 새로운 노드를 알맞은 위치에 삽입한다.

아까 저장해둔 노드와 새로 삽입한 노드 사이 경로에 존재하는 노드의 BF를 새로 업데이트한다. 이후 아까 저장해둔 노드의 BF가 0이거나 ± 1 이라면 불균형이 발생하지 않았으므로 삽입을 종료한다. 이때 오른쪽 불균형이라면 -1을, 왼쪽 불균형이라면 1을 저장해놓는다.

불균형이 발생한 경우는 회전이 수행되어야 한다. 아까 저장한 불균형을 의미하는 수(d)가 1이라면 처음은 왼쪽 불균형이다. 그리고 a의 자식 노드(b)의 BF를 확인한다. 1이라면 이는 LL 회전이 필요하다. a의 자식 노드를 b, b의 자식 노드를 c라고 하자.

LL 회전은 트리를 오른쪽으로 회전하는 동작으로, b가 새로운 서브의 root가 되고 a는 b의 오른쪽 자식이 된다.

b의 BF가 -1이라면 이는 LR 회전이 필요하다. LR 회전은 RR 회전 후 LL 회전을 수행하는 동작으로, b와 c로 RR 회전의 동작을 수행하고 a와 c로 LL 회전의 동작을 수행한다.

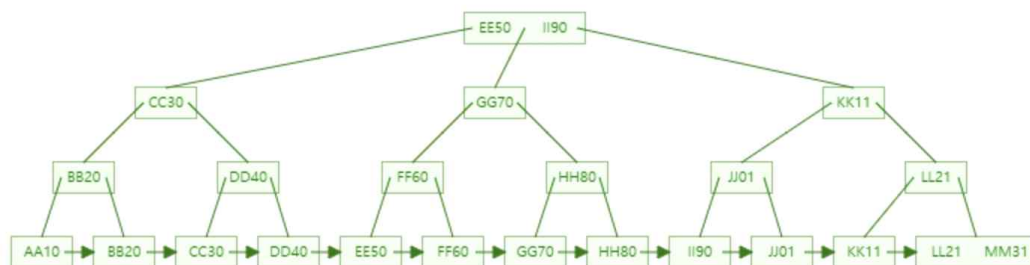
d가 -1이라면 처음 부분이 오른쪽 불균형이라는 의미이다. 마찬가지로 b의 BF를 확인한다. -1이라면 이는 RR 회전이 필요하다. RR회전은 트리를 왼쪽으로 회전하는 동작으로 b가 새로운 서브 트리의 root가 되고 a는 b의 왼쪽 자식이 된다.

d가 1이면 RL 회전이 필요하다. RL 회전은 LR 회전과 반대로 LL 회전 후 RR 회전을 수행하는 동작으로 b와 c로 LL 회전의 동작을 수행하고 a와 c로 RR 회전의 동작을 수행한다.

4. Result Screen-모든 명령어의 결과화면과 예외 처리의 동작을 설명

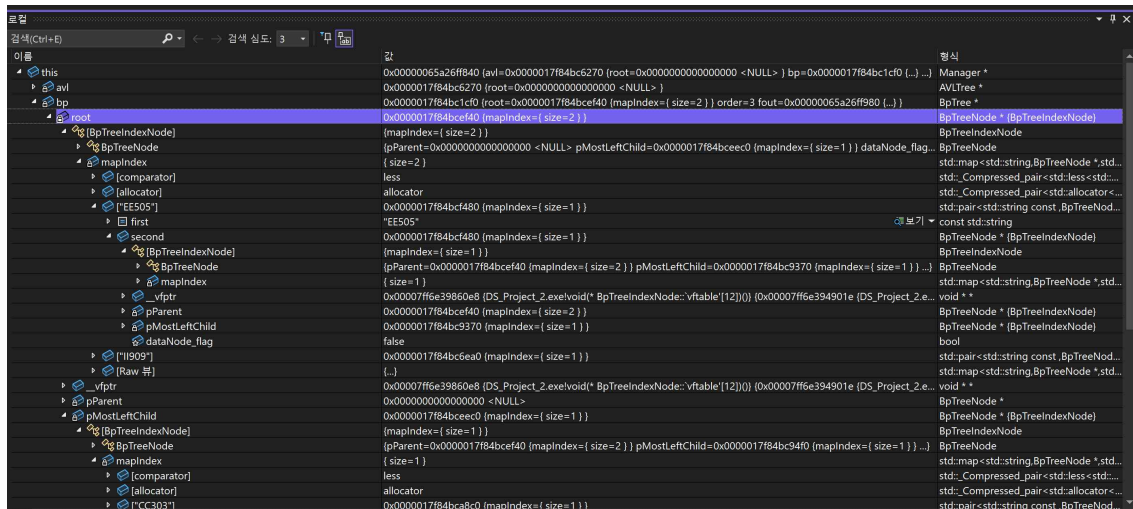
```
KoreanAir AA101 LAX 1 Boarding  
JEJU BB202 CJU 1 Delayed  
ASIANA CC303 CJU 1 Boarding  
JeanAir DD404 NRT 1 Boarding  
AirKwangwoon EE505 LHR 1 Cancelled  
KoreanAir FF606 CJU 1 Delayed  
JEJU GG707 CJU 1 Boarding  
ASIANA HH808 MUC 1 Delayed  
JeanAir II909 NRT 2 Boarding  
AirKwangwoon JJ010 LHR 5 Boarding  
KoreanAir KK111 CJU 3 Cancelled  
JEJU LL212 CJU 5 Boarding  
ASIANA MM313 MUC 4 Departure
```

LOAD 명령어를 통해 B+ 트리로 삽입될 input.txt 파일이다.



Visualization을 통하여 해당 B+ 트리를 시각화한 결과이다.

<LOAD>



LOAD 명령어를 실행한 후, 디버깅을 통해 bp 객체(B+ 트리)를 생성한 결과이다. Visualization과 같은 모양의 트리가 생성됨을 볼 수 있다.



<SEARCH_BP>

```
SEARCH_BP AA101
SEARCH_BP NN414
SEARCH_BP C    J
SEARCH_BP N    Q
SEARCH_BP JJ010
SEARCH_BP A    M
SEARCH_BP GG707
SEARCH_BP MM313
```

command.txt에 적혀있는 SEARCH_BP 명령어이다.


```

===== SEARCH_BP =====
AA101 | KoreanAir | LAX | 1 | Boarding
=====

===== ERROR =====
400
=====

=====SEARCH_BP=====
CC303 | ASIANA | CJU | 1 | Boarding
DD404 | JeanAir | NRT | 1 | Boarding
EE505 | AirKwangwoon | LHR | 1 | Cancelled
FF606 | KoreanAir | CJU | 1 | Delayed
GG707 | JEJU | CJU | 1 | Boarding
HH808 | ASIANA | MUC | 1 | Delayed
II909 | JeanAir | NRT | 2 | Boarding
JJ010 | AirKwangwoon | LHR | 5 | Boarding
=====

===== ERROR =====
400
=====

===== SEARCH_BP =====
JJ010 | AirKwangwoon | LHR | 5 | Boarding
=====

=====SEARCH_BP=====
AA101 | KoreanAir | LAX | 1 | Boarding
BB202 | JEJU | CJU | 1 | Delayed
CC303 | ASIANA | CJU | 1 | Boarding
DD404 | JeanAir | NRT | 1 | Boarding
EE505 | AirKwangwoon | LHR | 1 | Cancelled
FF606 | KoreanAir | CJU | 1 | Delayed
GG707 | JEJU | CJU | 1 | Boarding
HH808 | ASIANA | MUC | 1 | Delayed
II909 | JeanAir | NRT | 2 | Boarding
JJ010 | AirKwangwoon | LHR | 5 | Boarding
KK111 | KoreanAir | CJU | 3 | Cancelled
LL212 | JEJU | CJU | 5 | Boarding
MM313 | ASIANA | MUC | 4 | Departure
=====

===== SEARCH_BP =====
GG707 | JEJU | CJU | 1 | Boarding
=====

===== SEARCH_BP =====
MM313 | ASIANA | MUC | 4 | Departure
=====

```

log.txt에 출력된 결과로, AA101은 B+ 트리에 존재하는 항공편이므로 항공편의 데이터를 출력한 결과를 볼 수 있다.

NN414는 B+ 트리에 존재하지 않는 항공편이므로 항공편의 데이터가 출력되지 않고 에러를 출력한 결과를 볼 수 있다.

SEARCH C J의 명령어의 결과로, C~J의 항공편을 가지는 모든 노드의 정보를 출력한 결과를 볼 수 있다.

나머지 결과도 모두 SEARCH_BP 명령어 맞는 동작을 하는 결과를 볼 수 있다.

<ADD>

```
ADD JeanAir OO444 NRT Cancelled
ADD KoreanAir FF606 CJU Delayed
ADD ASIANA HH808 MUC Delayed
ADD KoreanAir AA101 LAX Cancelled
ADD AirKwangwoon EE505 LHR Cancelled
ADD JEJU GG707 CJU Boarding
ADD JEJU BB202 CJU Delayed
ADD AirKwangwoon JJ010 LHR Boarding
ADD ASIANA MM313 MUC Delayed
ADD ASIANA CC303 CJU Boarding
ADD KoreanAir KK111 CJU Cancelled
ADD JEJU LL212 CJU Boarding
ADD JeanAir DD404 NRT Boarding
ADD KoreanAir AA101 LAX Bodarding
```

command.txt에 존재하는 ADD 명령어이다.

```
===== ADD =====
OO444 | JeanAir | NRT | 5 | Cancelled
=====

===== ADD =====
FF606 | KoreanAir | CJU | 0 | Delayed
=====

===== ADD =====
HH808 | ASIANA | MUC | 0 | Delayed
=====

===== ADD =====
AA101 | KoreanAir | LAX | 1 | Cancelled
=====

===== ERROR =====
300
=====

===== ADD =====
GG707 | JEJU | CJU | 0 | Boarding
=====

===== ADD =====
BB202 | JEJU | CJU | 0 | Delayed
=====

===== ADD =====
JJ010 | AirKwangwoon | LHR | 4 | Boarding
=====

===== ERROR =====
300
=====

===== ADD =====
CC303 | ASIANA | CJU | 0 | Boarding
=====

===== ERROR =====
300
=====

===== ADD =====
LL212 | JEJU | CJU | 4 | Boarding
=====

===== ADD =====
DD404 | JeanAir | NRT | 0 | Boarding
=====

===== ADD =====
AA101 | KoreanAir | LAX | 0 | Boarding
=====
```

처음 AA101의 경우는 B+ 트리에 존재하나, 현재 상태가 Boarding이고 ADD 명령어에서 입력한 상태는 Cancelled이다. 따라서 좌석 수는 감소하지 않고 상태만 바뀌어야 하므로 좌석 수는 그대로 1을 유지하고 상태만 바뀌어서 출력된 결과를 볼 수 있다.

OO444의 경우, B+ 트리에 존재하지 않는다. 따라서 B+ 트리에 데이터를 삽입한다. JeanAir의 항공사 명을 가지므로 좌석수는 5로 고정하여 데이터를 추가하고 해당 데이터의 정보를 출력한 결과를 볼 수 있다.

FF606은 B+ 트리에 존재하며 좌석 수가 1이고, Delayed->Delayed의 상태를 가진다. 따라서 좌석 수를 하나 감소시킨다. 좌석 수가 0이 되었으므로 AVL 트리에 추가하고, 업데이트된 정보를 출력한 결과를 볼 수 있다.

EE505의 경우 B+ 트리에 존재하고 좌석 수가 1이다. 하지만 Cancelled->Cancelled의 상태를 가지므로, 명세서에 명시된 4가지의 상태에 해당하지 않는다. 따라서 에러를 출력한 결과를 볼 수 있다.

마지막 AA101의 경우, 맨 처음 ADD 명령어로 인해서 상태가 Cancelled로 바뀌었고 ADD 명령어의 입력은 Boarding이므로 좌석 수가 감소하고 상태도 변경된 결과를 출력한다.

<PRINT_BP>

```
===== PRINT_BP =====
AA101 | KoreanAir | LAX | 0 | Boarding
BB202 | JEJU | CJU | 0 | Delayed
CC303 | ASIANA | CJU | 0 | Boarding
DD404 | JeanAir | NRT | 0 | Boarding
EE505 | AirKwangwoon | LHR | 1 | Cancelled
FF606 | KoreanAir | CJU | 0 | Delayed
GG707 | JEJU | CJU | 0 | Boarding
HH808 | ASIANA | MUC | 0 | Delayed
II909 | JeanAir | NRT | 2 | Boarding
JJ010 | AirKwangwoon | LHR | 4 | Boarding
KK111 | KoreanAir | CJU | 3 | Cancelled
LL212 | JEJU | CJU | 4 | Boarding
MM313 | ASIANA | MUC | 4 | Departure
OO444 | JeanAir | NRT | 5 | Cancelled
=====
```

LOAD를 통해서 B+ 트리로 삽입한 데이터를 항공편 명 기준, 오름차순으로 정렬하여 출력한 결과이다.

```

===== PRINT_BP =====
AA101 | KoreanAir | LAX | 6 | Boarding
BB202 | JEJU | CJU | 5 | Delayed
CC303 | ASIANA | CJU | 7 | Boarding
DD404 | JeanAir | NRT | 5 | Boarding
EE505 | AirKwangwoon | LHR | 6 | Cancelled
FF606 | KoreanAir | CJU | 7 | Delayed
GG707 | JEJU | CJU | 5 | Boarding
HH808 | ASIANA | MUC | 7 | Delayed
JJ010 | AirKwangwoon | LHR | 6 | Boarding
KK111 | KoreanAir | CJU | 7 | Cancelled
LL212 | JEJU | CJU | 5 | Boarding
MM313 | ASIANA | MUC | 7 | Delayed
OO444 | JeanAir | NRT | 5 | Cancelled
=====

```

위에 설명한 내용과 동일한 command.txt에서 LOAD 명령어만 제외한 결과이다. B+트리가 비어있으므로 ADD 명령어로 입력한 데이터는 모두 B+ 트리에 삽입된다. 또한 모두 각 항공사에 맞추어 좌석 수로 고정된 결과를 볼 수 있다.

<SEARCH_AVL>

```

SEARCH_AVL    FF606
SEARCH_AVL    KK111
SEARCH_AVL    AA101
SEARCH_AVL    OO444

```

command.txt에서 사용한 명령어이다.

```

===== SEARCH_AVL =====
FF606 | KoreanAir | CJU | 0 | Delayed
=====

===== ERROR =====
500
=====

===== SEARCH_AVL =====
AA101 | KoreanAir | LAX | 0 | Boarding
=====

===== ERROR =====
500
=====

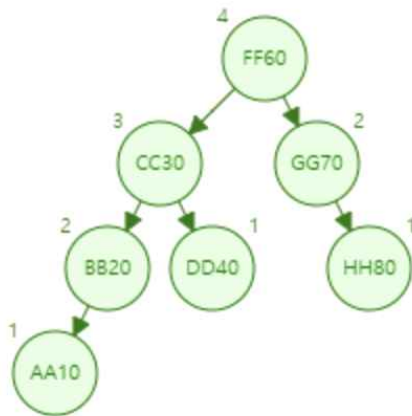
```

B+ 트리에서 좌석 수를 감소시키고, 좌석 수가 0이 된 항공편은 AVL 트리에 삽입한다. 따라서 AVL 트리에 존재하는 항공편의 정보는 좌석 수가 모두 0이다. AVL 트리에 존재하지 않는 항공편 명을 입력하면 에러를 출력하는 결과를 볼 수 있다.

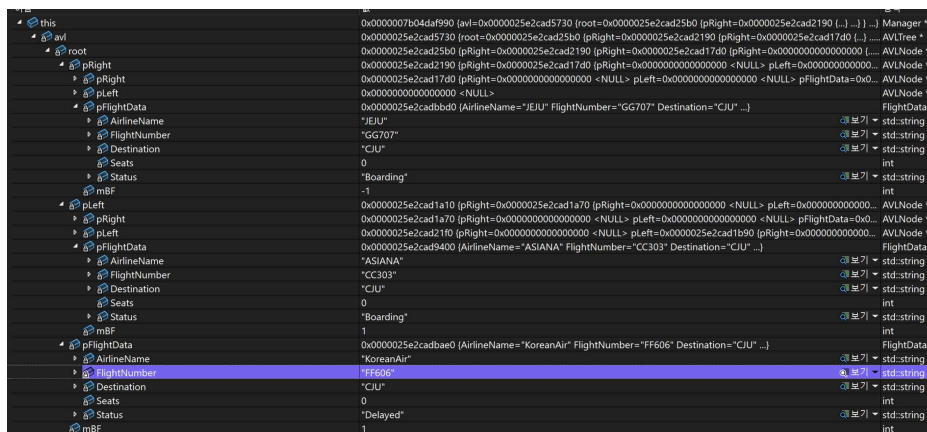
<VLOAD>

FF606
HH808
GG707
BB202
CC303
DD404
AA101

디버깅을 통하여 B+ 트리에서 AVL 트리로 삽입되는 항공편 명을 출력한 화면이다.



해당 삽입 순서로 AVL 트리를 구성하면 이런 형태의 AVL트리가 형성된다.



AVL 트리를 생성한 후 디버깅을 통하여 트리를 보면 Visualization으로 만든 트리와 동일한 구조를 지님을 볼 수 있다.

this	0x0000005a568ff3b0 (avl=0x00000219e8725c80 (root=0x00000219e8721da0 (pRight=0x00000219e8722400 (...)...)...) Manager *
avl	0x00000219e8725c80 (root=0x00000219e8721da0 (pRight=0x00000219e8722400 (pRight=0x00000219e8722160 (...)...)...) AVLTree *
bp	0x00000219e8721980 (root=0x00000219e872ef80 (mapIndex={ size=2 }) order=3 fout=0x0000005a568ff4f0 (...)...) BpTree *
Print_vector	{ size=7 } std::vector<FlightData *,std::allocator<...
[capacity]	9 unsigned __int64
[allocator]	allocator std::Compressed_pair<std::allocator<...
[0]	0x00000219e871b3b0 (AirlineName="KoreanAir" FlightNumber="AA101" Destination="LAX" ...) FlightData *
[1]	0x00000219e87292c0 (AirlineName="JEJU" FlightNumber="BB202" Destination="CJU" ...) FlightData *
[2]	0x00000219e8729440 (AirlineName="ASIANA" FlightNumber="CC303" Destination="CJU" ...) FlightData *
[3]	0x00000219e872a810 (AirlineName="JeanAir" FlightNumber="DD404" Destination="NRT" ...) FlightData *
[4]	0x00000219e872bb20 (AirlineName="KoreanAir" FlightNumber="FF606" Destination="CJU" ...) FlightData *
[5]	0x00000219e872bc10 (AirlineName="JEJU" FlightNumber="GG707" Destination="CJU" ...) FlightData *
[6]	0x00000219e872bd00 (AirlineName="ASIANA" FlightNumber="HH808" Destination="MUC" ...) FlightData *
[Raw #]	(Mypair=allocator) std::vector<FlightData *,std::allocator<...

Stack을 이용하여 중위 순회의 방법으로 AVL 트리를 순회해서 항공편 정보를 벡터에 추가했다. 따라서 벡터에는 항공편 명 기준, 오름차순으로 정렬되어 삽입된 결과를 볼 수 있다.

<VPRINT>

this	0x000000c4066ff790 (avl=0x0000028e74785a00 (root=0x0000028e747826a0 (pRight=0x0000028e747823a0 (...)...)...) Manager *
avl	0x0000028e74785a00 (root=0x0000028e747826a0 (pRight=0x0000028e747823a0 (pRight=0x0000028e74781fe0 (...)...)...) AVLTree *
bp	0x0000028e74781ce0 (root=0x0000028e7478ef80 (mapIndex={ size=2 }) order=3 fout=0x000000c4066ff8d0 (...)...) BpTree *
Print_vector	{ size=7 } std::vector<FlightData *,std::allocator<...
[capacity]	9 unsigned __int64
[allocator]	allocator std::Compressed_pair<std::allocator<...
[0]	0x0000028e74789440 (AirlineName="ASIANA" FlightNumber="CC303" Destination="CJU" ...) FlightData *
[1]	0x0000028e7478bd00 (AirlineName="ASIANA" FlightNumber="HH808" Destination="MUC" ...) FlightData *
[2]	0x0000028e747892c0 (AirlineName="JEJU" FlightNumber="BB202" Destination="CJU" ...) FlightData *
[3]	0x0000028e7478bc10 (AirlineName="JEJU" FlightNumber="GG707" Destination="CJU" ...) FlightData *
[4]	0x0000028e7478a810 (AirlineName="JeanAir" FlightNumber="DD404" Destination="NRT" ...) FlightData *
[5]	0x0000028e7478bb20 (AirlineName="KoreanAir" FlightNumber="FF606" Destination="CJU" ...) FlightData *
[6]	0x0000028e7477b3b0 (AirlineName="KoreanAir" FlightNumber="AA101" Destination="LAX" ...) FlightData *

===== VPRINT A =====

```
ASIANA | CC303 | CJU | Boarding
ASIANA | HH808 | MUC | Delayed
JEJU | BB202 | CJU | Delayed
JEJU | GG707 | CJU | Boarding
JeanAir | DD404 | NRT | Boarding
KoreanAir | FF606 | CJU | Delayed
KoreanAir | AA101 | LAX | Boarding
=====
```

먼저, A의 조건으로 벡터를 정렬하여 출력한 결과이다.

조건 A는 항공사 명 오름차순, 도착지 명 오름차순, 상태 정보 내림차순으로 정렬해야 하는데 알맞은 결과가 나타남을 볼 수 있다.

this	0x0000002b4ff4f4d0 (avl=0x00000190cd9c6310 (root=0x00000190cd9c1b60 (pRight=0x00000190cd9c2340 (...)...)...) Manager *
avl	0x00000190cd9c6310 (root=0x00000190cd9c1b60 (pRight=0x00000190cd9c2340 (pRight=0x00000190cd9c22e0 (...)...)...) AVLTree *
bp	0x00000190cd9c1c20 (root=0x00000190cd9ccef80 (mapIndex={ size=2 }) order=3 fout=0x0000002b4ff4f610 (...)...) BpTree *
Print_vector	{ size=7 } std::vector<FlightData *,std::allocator<...
[capacity]	9 unsigned __int64
[allocator]	allocator std::Compressed_pair<std::allocator<...
[0]	0x00000190cd9c9cb10 (AirlineName="JEJU" FlightNumber="GG707" Destination="CJU" ...) FlightData *
[1]	0x00000190cd9c9440 (AirlineName="ASIANA" FlightNumber="CC303" Destination="CJU" ...) FlightData *
[2]	0x00000190cd9c9bb20 (AirlineName="KoreanAir" FlightNumber="FF606" Destination="CJU" ...) FlightData *
[3]	0x00000190cd9c92c0 (AirlineName="JEJU" FlightNumber="BB202" Destination="CJU" ...) FlightData *
[4]	0x00000190cd9c9bb3b0 (AirlineName="KoreanAir" FlightNumber="AA101" Destination="LAX" ...) FlightData *
[5]	0x00000190cd9c9bd00 (AirlineName="ASIANA" FlightNumber="HH808" Destination="MUC" ...) FlightData *
[6]	0x00000190cd9ca810 (AirlineName="JeanAir" FlightNumber="DD404" Destination="NRT" ...) FlightData *
[Raw #]	(Mypair=allocator) std::vector<FlightData *,std::allocator<...
fin	(Filebuffer=(Pcvt=0x0000000000000000 <NULL> _Mychar=0 '\0' _Wrotesome=false ...) std::basic_ofstream<char,std::char_traits<...
fout	(Filebuffer=(Pcvt=0x0000000000000000 <NULL> _Mychar=0 '\0' _Wrotesome=false ...) std::basic_ofstream<char,std::char_traits<...

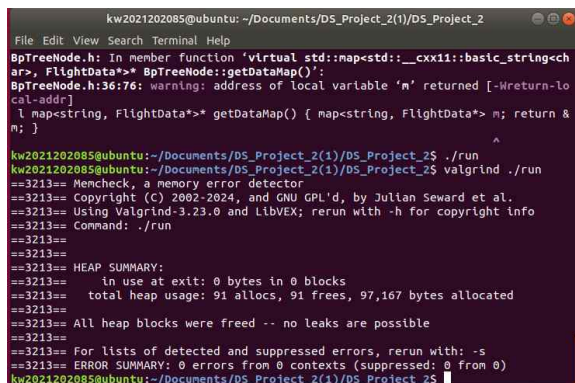

```

===== VPRINT B =====
JEJU | GG707 | CJU | Boarding
ASIANA | CC303 | CJU | Boarding
KoreanAir | FF606 | CJU | Delayed
JEJU | BB202 | CJU | Delayed
KoreanAir | AA101 | LAX | Boarding
ASIANA | HH808 | MUC | Delayed
JeanAir | DD404 | NRT | Boarding
=====

```

B의 조건은 도착지 명 오름차순, 상태 정보 오름차순, 항공사 명 내림차순의 순서로 정렬하는데, 알맞은 결과가 출력되었다.

<EXIT>



```

kw2021202085@ubuntu: ~/Documents/DS_Project_2(1)/DS_Project_2
File Edit View Search Terminal Help
BpTreeNode.h: In member function 'virtual std::map<std::__cxx11::basic_string<char>, FlightData*>*> BpTreeNode::getDataMap()':
BpTreeNode.h:36:76: warning: address of local variable 'm' returned [-Wreturn-local-addr]
    36 |     map<string, FlightData*> getDataMap() { map<string, FlightData*> m; return &
        |                                                                ^
kw2021202085@ubuntu:~/Documents/DS_Project_2(1)/DS_Project_2$ ./run
kw2021202085@ubuntu:~/Documents/DS_Project_2(1)/DS_Project_2$ valgrind ./run
==3213== Memcheck, a memory error detector
==3213== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==3213== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==3213== Command: ./run
==3213==
==3213== HEAP SUMMARY:
==3213==   in use at exit: 0 bytes in 0 blocks
==3213== total heap usage: 91 allocs, 91 frees, 97,167 bytes allocated
==3213== All heap blocks were freed -- no leaks are possible
==3213== For lists of detected and suppressed errors, rerun with: -s
==3213== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
kw2021202085@ubuntu:~/Documents/DS_Project_2(1)/DS_Project_2$

```

EXIT의 명령어는 동적으로 할당한 메모리를 해제하고 프로그램을 종료하는 명령어이다.

리눅스에 Valgrind 명령어를 이용하여 메모리 누수를 확인한 결과, 모두 정상적으로 해제된 결과를 볼 수 있다.

5. Consideration-고찰

B+ 트리를 처음 구현해 봐서 예러도 많이 발생하고 여러 가지 힘든 점이 있었다. 또한 비효율적으로 코드를 작성한 부분도 존재하는데. 특히 ADD 명령어를 구현한 부분에서 B+ 트리에 항공편이 존재하는지 탐색하고, 존재한다면 B+ 트리에서 상태를 업데이트 하기 위해서 한번 더 B+ 트리를 탐색하는 비효율적인 코드를 작성했다. 따라서 B+트리에 존재하는 정보를 탐색하고, 존재한다면 해당 데이터를 반환하는 방식으로 코드를 작성하여 2번 탐색하지 않고 더 효율적으로 코드를 개선할 수 있다고 생각한다.

입력되는 항공권 정보의 항공편 명은 고유하므로 원래의 B+ 트리의 원소가 같은 상황이 발생하지 않는다. 따라서 BPTree 클래스의 searchDataNode 함수에서 if문을 이용하여 데이터를 비교하는 상황에서 두 데이터의 항공편 명이 같은 상황은 고려하지 않았다. 하지만 해당 함수를 B+ 트리에 존재하는 데이터를 찾는 과정에서도 사용했는데, 같은 항공편 명이 들어와서 B+ 트리에서 찾는 상황에서 잘못된 동작이 나타남을 발견했다. 다음 자식 노드로 이동할 때, LeftChild로 이동하는 경우는 해당하지 않으나, 맨 오른쪽 자식으로 이동하는 상황에는

searchDataNode 함수에 조건에는 =을 붙여야 올바른 자식 노드로 이동할 수 있다. 이런 방식으로 해당 코드를 수정하여 잘못된 동작을 고쳤다.