데이터 구조 실습 보고서

학 번: 2018202014

이 름: 박지원

실습 분반: 목 34

Introduction

본 프로젝트는 B+ tree, AVL tree, STL vector를 이용하여 코로나 19 예방접종 관리 프로그램을 구현한다.
예방 접종 관리 데이터에는 이름, 백신 명, 접종 횟수, 나이 지역 명이 있으며 이는 VaccinationData 노드에서 모아서 관리한다. 이를 이용하여 접종 대상자 및 접종 완료자에 대한 정보를 제공하여 여러 데이터 구조에서 관리한다.
이 데이터를 관리하는 데 사용하는 여러 명령어들이 있으며, 이 명령어들은 "command.txt"파일에 의해서 관리된다.

백신 접종 데이터는 VaccinationData class에서 관리하고 백신의 종류에는 Pfizer, Moderna, AstraZenca, Janssen이 있다. 이 준 Janssen은 접종 횟수가 1이면 접종 완료자이고, 나머지는 접종 횟수가 2이면 접종 완료자라 한다.
input_data.txt에는 백신 접종 대상자들에 대한 정보가 있고, LOAD 명령어를 통해 B+ tree에 저장된다. input_data.txt에는 접종 완료자가 포함 될 수 없다.

BpTree는 B+ tree형태의 데이터 구조로, 접종 대상자 및 접종 진행자 등 모든 인원을 관리하며 검색, 추가, 접종 횟수 증가 등을 지원한다. 이러한 BpTree의 특징 및 주의할 점은 아래와 같다.

- B+ tree의 차수는 3이다.
- 주어진 input_data.txt에 저장된 데이터를 읽은 후, 접종 대상자는 B+ tree에 저장
- ADD 명령어로 추가되는 데이터를 읽은 후, B+ tree에 저장한다. B+ treed에 없으면 node를 새로 추가하며, 존재하는 경우 접종 횟수만 증가 시킨다. ADD로 추가된 데이터가 이미 접종 횟수가 2인 접종 완료자인 경우 예외처리를 한다.
- B+ tree에 저장되는 데이터는 VaccinationData class로 선언되어 있다.
- 이름 정보는 항상 고유하며, 중복으로 입력되는 경우는 없다고 가정한다.
- B+ tree는 이름을 기준으로 정렬하며, 대소문자를 구별한다.
- B+ tree는 데이터 노드와 인덱스 노드로 구성되어 있으며, 데이터 노드에서 데이터 관리는 map으로 한다.

AVLTree는 AVL tree형태의 데이터 구조로, 접종 완료자에 대한 데이터만 관리하며, 검색, 추가 등의 기능을 지원한다. 이러한 AVLTree의 특징 및 주의할 점은 아래와 같다.

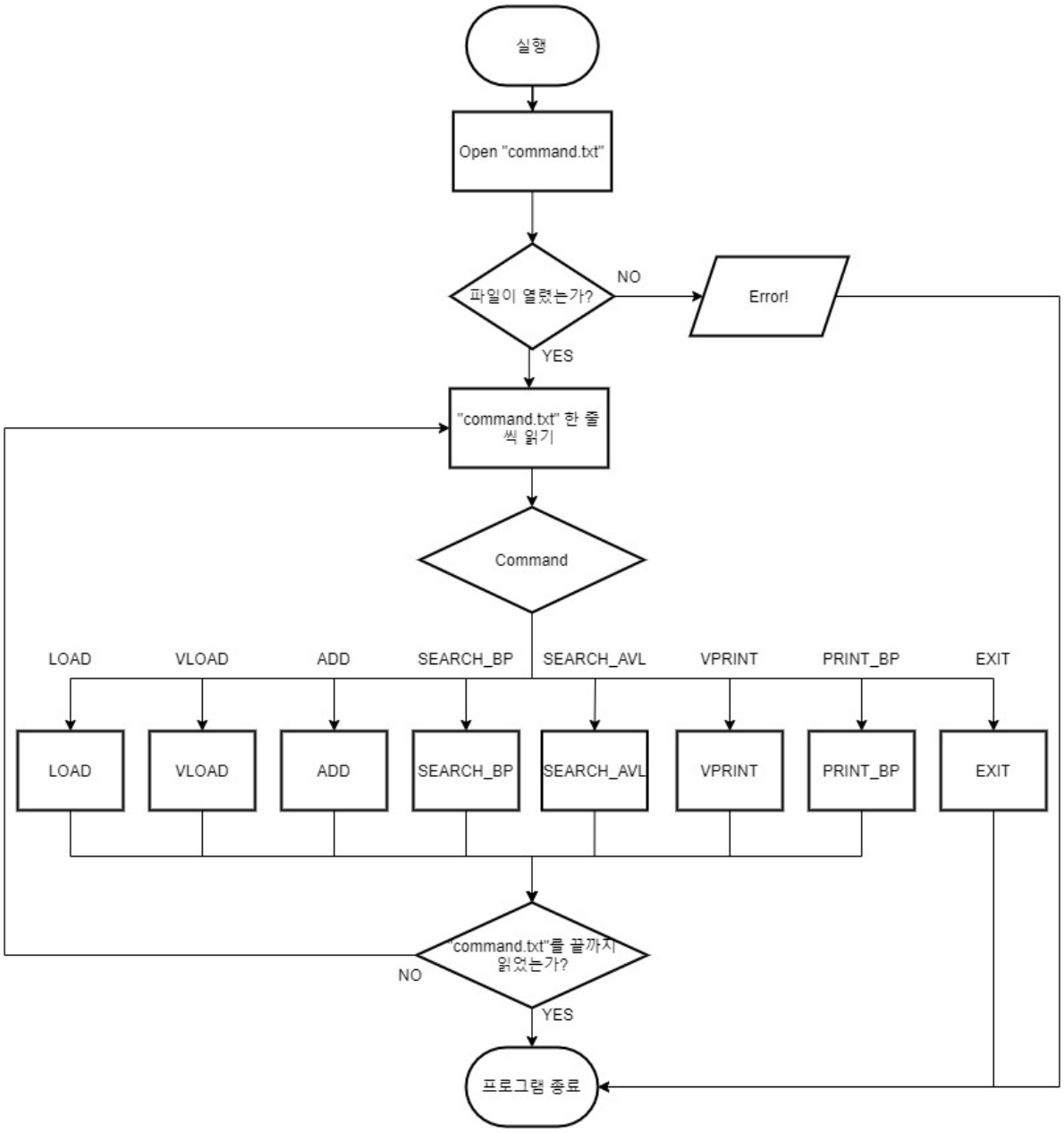
- ADD 명령어로 B+ tree에 저장된 데이터의 접종 횟수를 업데이트 후에, 접종 완료자는 AVL tree에 저장한다. 이때, 해당 데이터는 B+ tree에서 삭제하지 않고 그대로 유지한다.

- AVL tree에 저장되는 데이터는 VaccinationData class로 선언되어 있다.
- AVL tree는 이름을 기준으로 정렬하며, 대소문자를 구별하지 않는다.
- AVL tree는 각 노드마다 balance factor를 가지고 있으며, 이를 활용하여 트리의 균형을 유지한다.

Flowchart

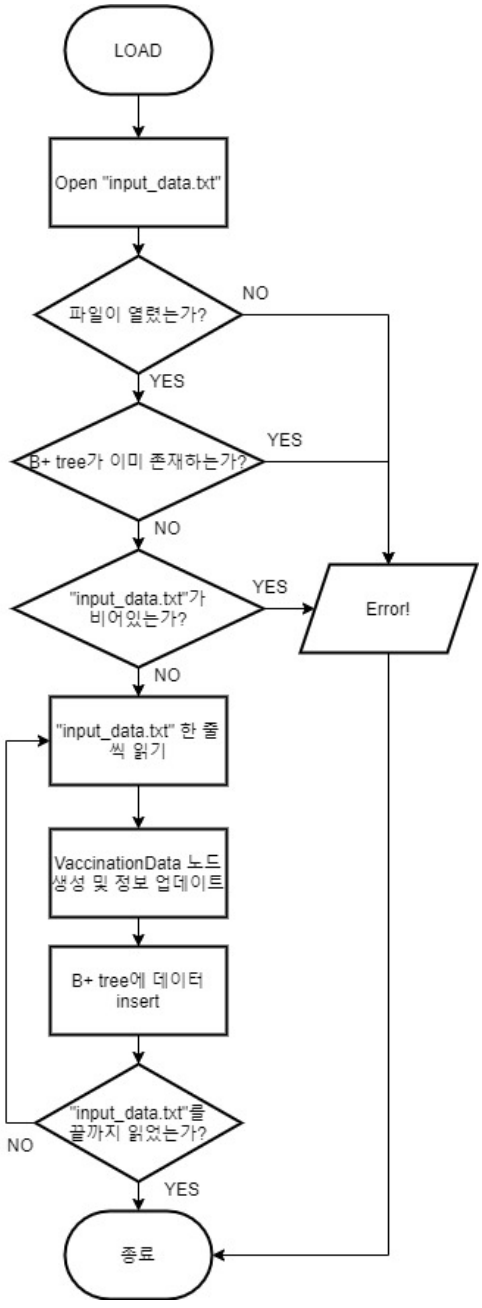
<manager>

프로그램을 전체적으로 관리하는 Manager 클래스의 run은 "command.txt"에서 명령어를 순차적으로 읽어와 그에 해당하는 동작을 수행한다. Flow chart는 아래와 같다.



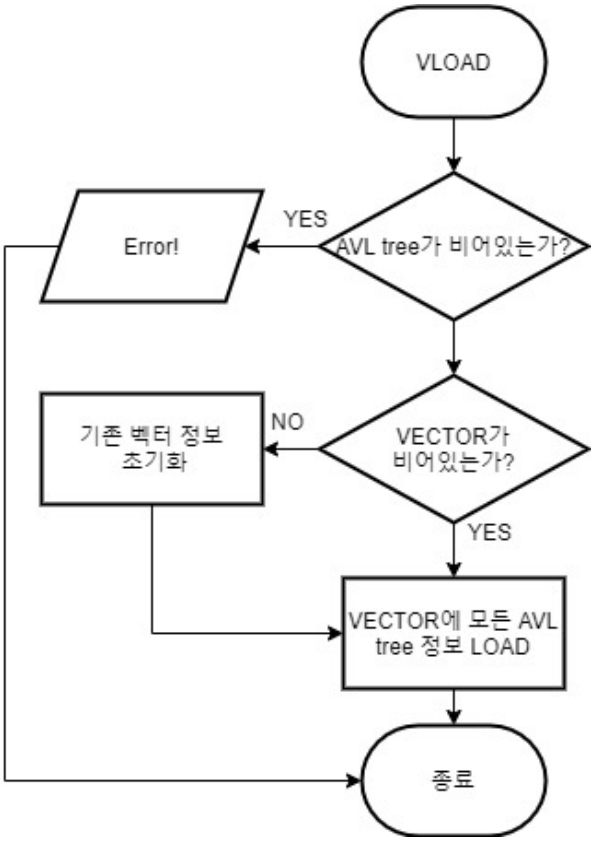
<LOAD>

LOAD 명령어는 "input_data.txt"에서 백신 접종 데이터를 읽어와 b+ tree에 저장한다. B+ tree가 이미 존재하거나 파일이 열리지 않은 경우, 또는 "input_data.txt"가 비어 있으면 에러 코드를 출력한다. Flow chart는 아래와 같다.



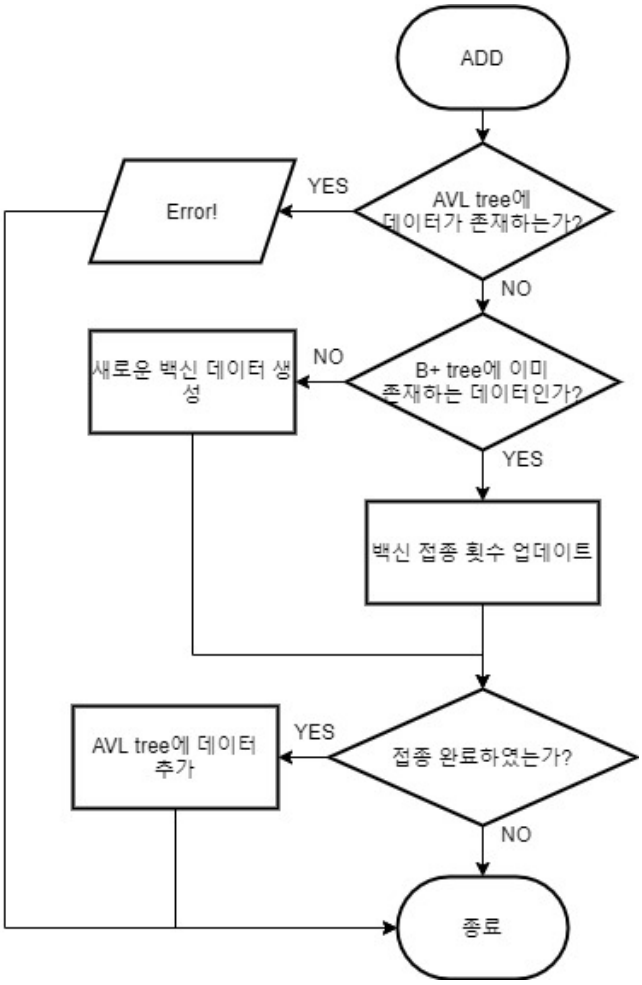
<VLOAD>

VLOAD는 AVL tree의 모든 데이터를 Vector에 저장하는 명령어이다. AVL tree가 비어있거나 Vector가 비어 있으면 에러 코드를 출력한다. Flowchart는 아래와 같다.



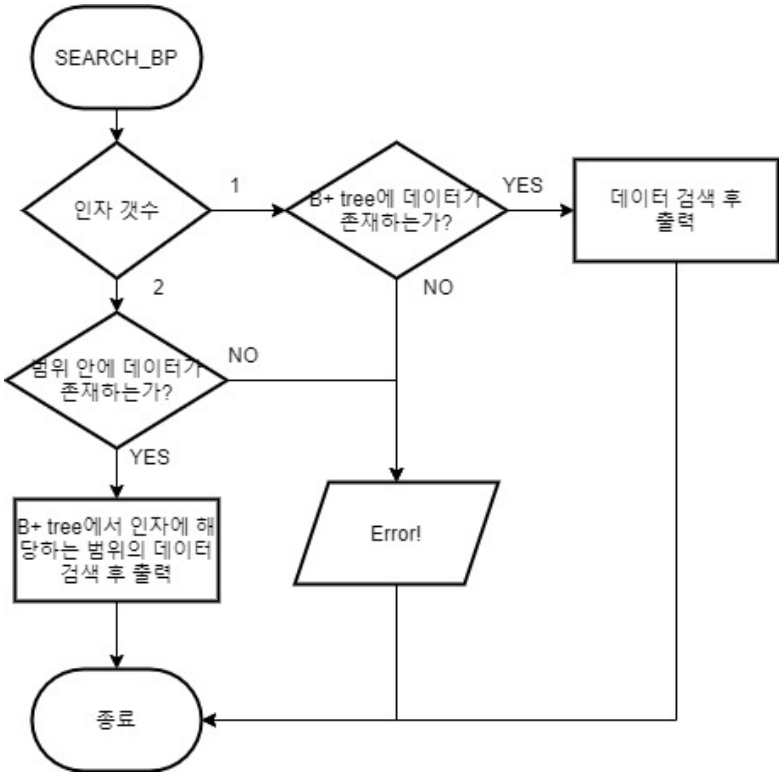
<ADD>

ADD는 B+ tree나 AVL tree에 백신을 접종 한 후 데이터를 반영하는 명령어이다. ADD를 통해 B+ tree에 백신 접종 횟수를 업데이트 하거나 새로운 데이터를 추가한다. 만약, 접종 완료하면 AVL tree에 데이터를 추가한다. AVL tree에 이미 해당 데이터가 존재하는 경우 에러 코드를 출력한다. Flowchart는 아래와 같다.



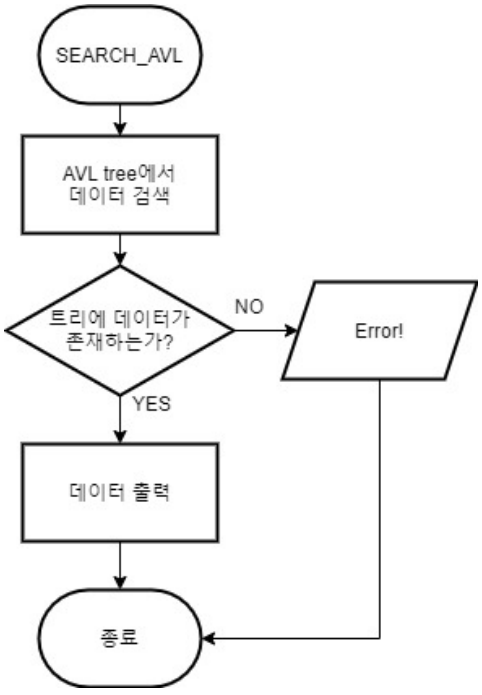
<SEARCH_BP>

SEARCH_BP는 B+ tree에서 인자에 해당하는 데이터를 찾거나 인자에 해당하는 범위의 모든 데이터를 출력하는 명령어이다. 찾는 데이터가 존재하지 않는 경우 에러 코드를 출력한다. Flowchart는 아래와 같다.



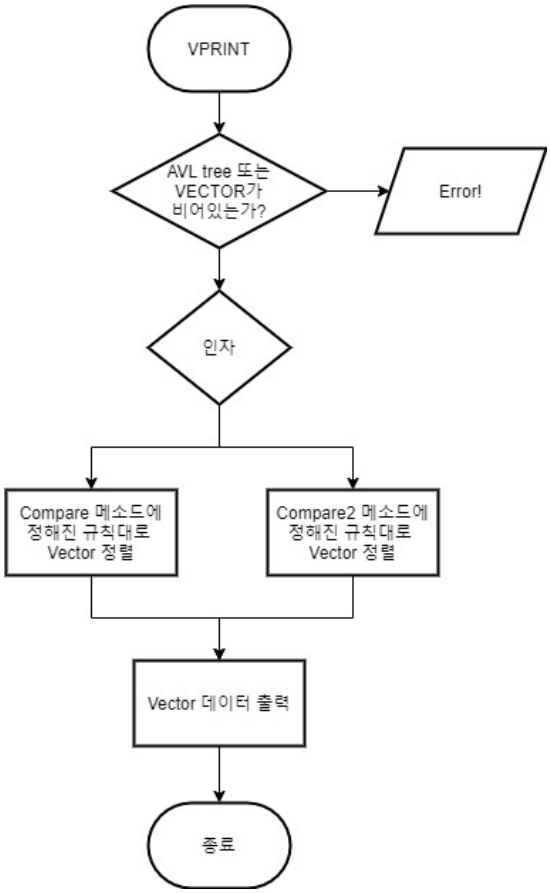
<SEARCH_AVL>

SEARCH_AVL은 AVL tree에서 데이터를 검색 및 출력을 하는 명령어이다. 만약, 트리에 데이터가 존재하지 않는 경우 에러 코드를 출력한다. Flowchart는 아래와 같다.



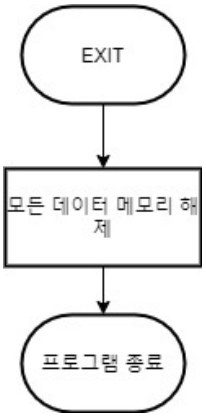
<VPRINT>

VPRINT는 VLOAD를 통해 Vector에 저장한 데이터를 정해진 기준에 따라 정렬 및 출력하는 명령어이다. AVL tree나 Vector가 비어 있는 경우 에러 코드를 출력한다. Flow char는 아래와 같다.



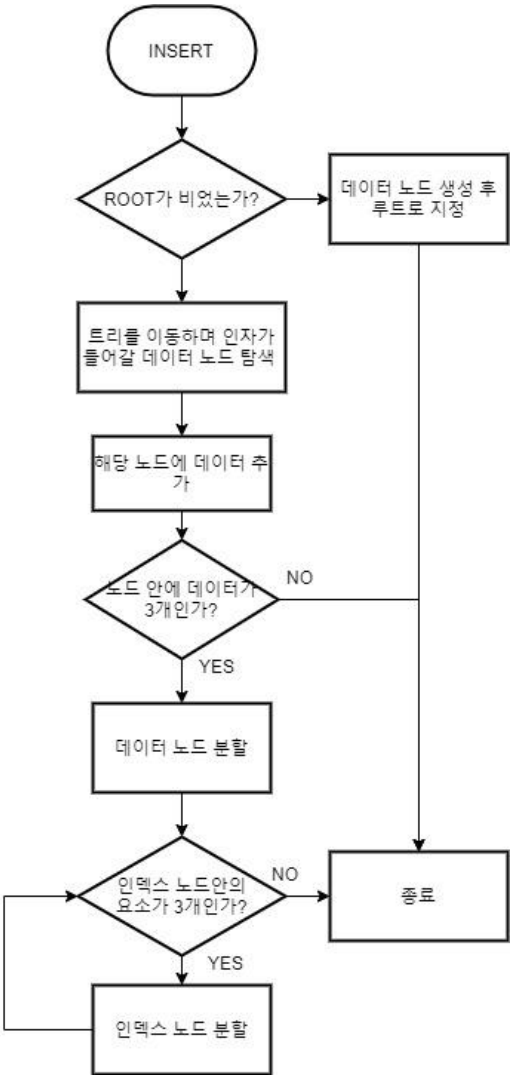
<EXIT>

EXIT는 프로그램 내에 사용한 모든 메모리를 해제하고 프로그램을 종료하는 명령어이다. Flowchart는 아래와 같다.

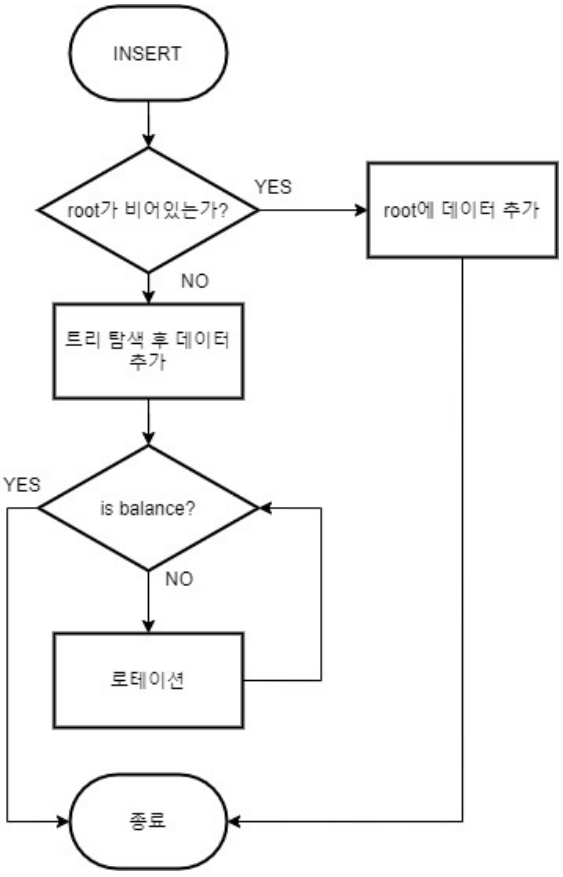


추가적으로 B+ tree와 AVL tree에 데이터를 추가하는 방법은 아래와 같다.

<B+ tree>



<AVL tree>



Algorithm

이 프로젝트에서 사용한 알고리즘은 크게 2가지로, AVL tree와 B+ tree 이다.

<B+ tree>

B+ tree는 m개의 자식과 m-1개의 데이터를 갖는 다원 트리로 기존 다른 tree의 약점을 보완하기 위해 만들어졌다.

B+ tree의 가장 큰 특징은 다른 tree 들은 보통 데이터를 루트 노드부터 가지 노드, 잎 노드까지 각자 데이터를 갖고 있지만, B+ tree는 노드의 종류가 Data node와 Index node 두 가지로 나뉜다.

따라서, B+ tree는 다른 tree에게 없는 규칙이 있다.

첫 번째는 모든 Data node는 동일한 레벨에 존재하며 잎 노드에 존재하고 Data node는 순서대로 서로를 가리키며 연결 되어있다.

두 번째는 Data는 Data node만 가지며 Index node는 다른 노드를 가리키는 키 값만을 가진다.

세 번째는 만약 node가 가질 수 있는 정보의 양을 초과하면 분열하여 트리를 재구성 한다.

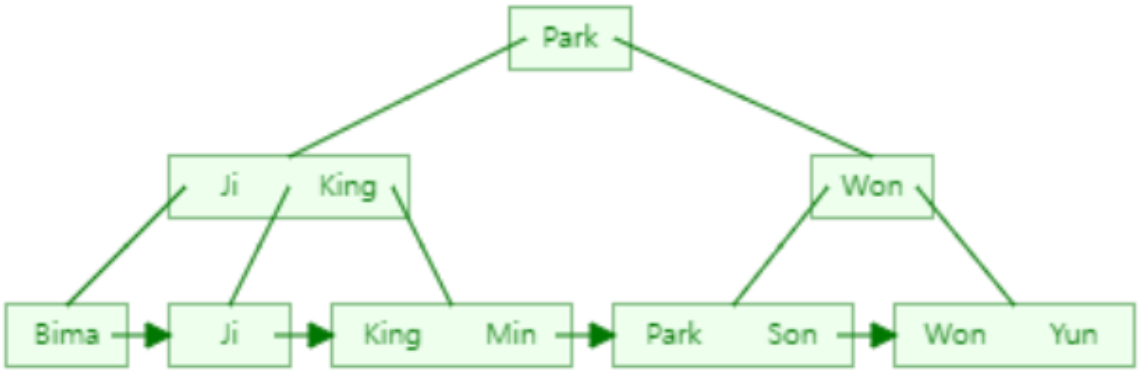
그리고 이 프로그램에서는 요소들이 Map으로 관리된다.
Index node는 이름을 Key 값으로 갖고 요소로는 그 Key 값 보다 큰 Key를 가진 node를 갖는다.
Data node는 이름을 Key 값으로 갖고 요소로는 그 Key에 해당하는 이름의 VaccinationData를 갖는다.

다음은 B+ tree에서 구현한 함수들 이다.

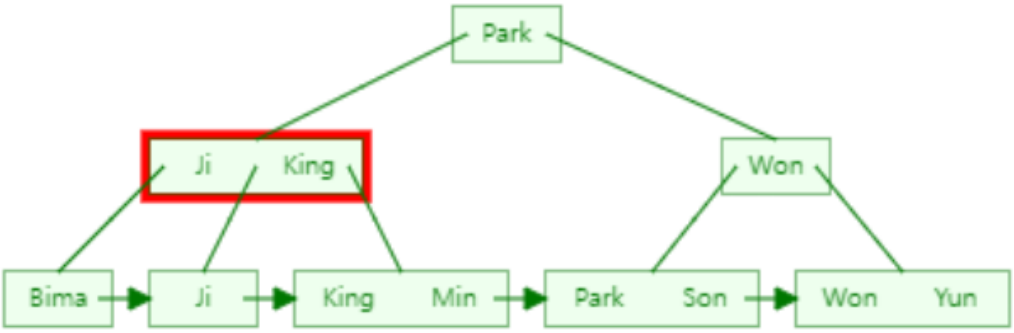
반환 타입	함수	설명
bool	Insert(VaccinationData*)	tree에 데이터 삽입
bool	exceedDataNode(BpTreeNode*)	DataNode의 요소가 초과 했는가?
bool	exceedIndexNode(BpTreeNode*)	IndexNode의 요소가 초과 했는가?
void	splitDataNode(BpTreeNode*)	DataNode 분할 및 tree 재구성
void	splitIndexNode(BpTreeNode*)	IndexNode 분할 및 tree 재구성
void	SearchRange(string, string)	범위 내 데이터 탐색 및 출력
void	Print();	tree 안의 모든 데이터 출력
BpTreeNode*	searchDataNode(string)	특정 데이터 탐색

1. Insert

Insert는 명령어 ADD나 LOAD가 들어왔을 때, 데이터를 B+ tree에 추가하는 함수이다.
이 프로그램에서는 Insert를 구현한 방법은 아래와 같다.
1) 데이터를 추가할 Data Node를 찾는다.
B+ tree에 아래와 같이 데이터가 저장되어 있다고 가정하자.

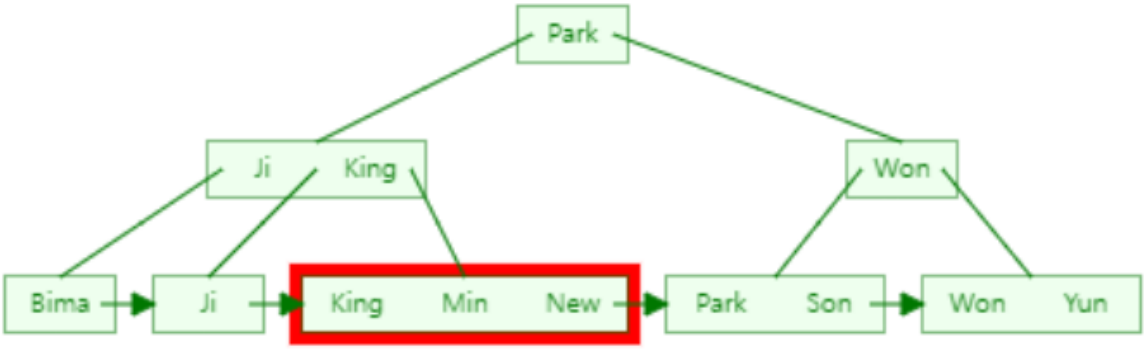


여기서 New라는 이름을 가진 VaccinationData를 추가 한다고 하면, 우선 Data를 추가할 Data Node를 찾아야 한다. 이 프로그램의 Key는 이름 사전 순 기준 이므로 Park와 비교 했을 때, Park보다 New가 우선 이므로 아래와 같이 MostLeftChild node로 내려가게 된다.



다음 Index node의 요소가 여러 개 이므로 해당 node의 map을 사전 순으로 탐색하게 된다.
그 결과 New가 King보다 사전 상 뒤에 있으므로 Map에서 Key가 King인 요소가 가리키는 Node로 내려간다.
그리고 해당 node가 자식 노드가 없으므로 Data node(잎 노드)임을 알 수 있다.

2) 찾은 Data Node에 Data를 추가한다.
알맞은 Data Node를 찾으면 아래 그림과 같이 Data를 Map을 추가해준다.



3) Data Node의 요소가 3개면 분할 하고 재구성한다.
exceedDataNode 함수로 Data Node 요소가 초과했는지 판별하고 만약 초과 하였다면,
splitDataNode 함수로 Data Node를 분할 하고 tree를 재구성한다.
두 함수는 뒤에서 각각 설명한다.

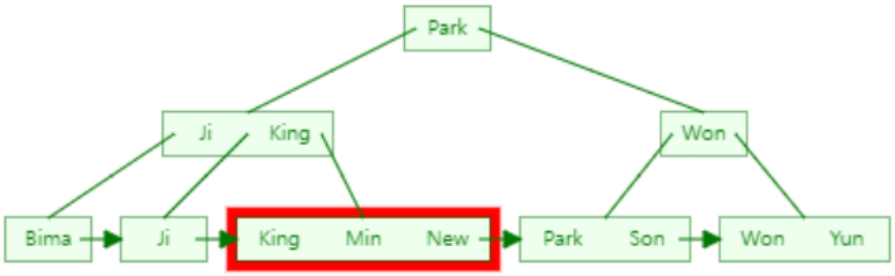
4) Data Node를 분할하고 재구성 했을 때 Index Node의 요소가 3개면 분할 하고 재구성한다.
exceedIndexNode 함수로 Index Node 요소가 초과했는지 판별하고 만약 초과 하였다면,
splitIndexNode 함수로 Index Node를 분할 하고 tree를 재구성한다.
만약, 재구성 했을 때 parent Index Node 요소가 초과했다면, 다시 분할 하고 tree를 재구성한다. 이는, 초과한 Index Node가 없을 때까지 반복한다.
위 두 함수는 뒤에서 각각 설명한다.

2. exceedDataNode

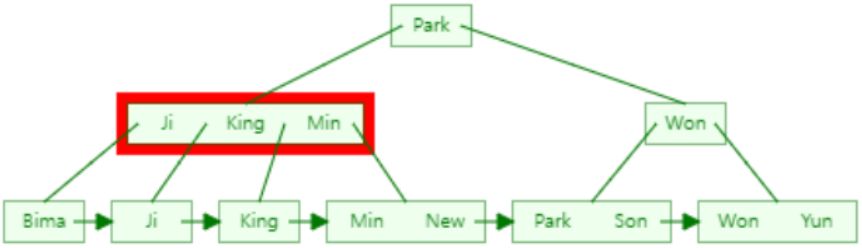
인자로 받은 DataNode의 요소가 3개인지 판별한다. 3개이면 true를 반환하고 2개 이하면 false를 반환한다.

3. splitDataNode

아래와 같이 Data Node를 분할하고 재구성 한다.



(분할 전)



(분할 후)

이 프로그램에서 분할을 분할하는 노드가 root일 때 와 아닐 때로 나누어 아래와 같이 구현 하였다.

(Root 일 경우)

1. 새로운 root 노드(index node)와 left child 노드(data node)를 생성한다.
2. root의 첫 번째 데이터는 left child 노드에 넣고 두 번째 데이터는 새로운 root 노드에 넣는다.
3. root의 첫 번째 데이터를 지운다.
4. 새로운 root 노드의 key 요소(right child node)는 root로 지정하고 MostLeftChild(left child node)는 left child 노드로 지정한다.
5. root와 left child 노드의 부모를 새로운 root 노드로 지정한다.

(Root가 아닐 경우)

1. left child 노드(data node)를 생성한다.
2. 기존 Data node의 첫 번째 데이터는 left child 노드에 넣고 두 번째 데이터는 Data node의 parent 노드 Map에 삽입한다.
3. Data node의 첫 번째 데이터를 지운다.
4. Parent 노드에 넣은 데이터 Key의 요소는 Data node로 지정한다.

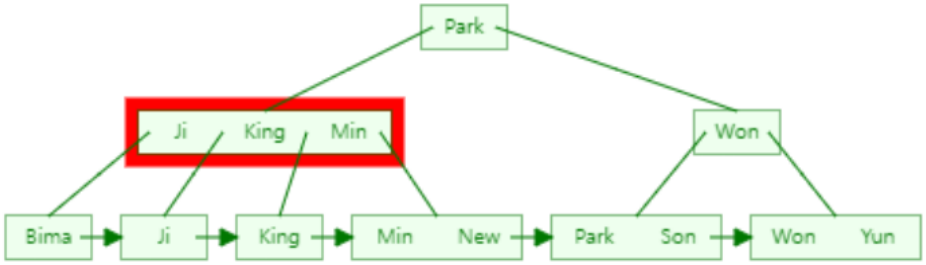
5. Parent 노드에 넣은 데이터의 이전 데이터의 Key의 요소는 Left child node로 지정한다. 만약 데이터가 첫 번째라면 MostLeftChild로 지정한다.
6. 만약 parent 노드의 데이터가 3개 이상이면 splitIndexNode함수를 호출한다.

4. exceedIndexNode

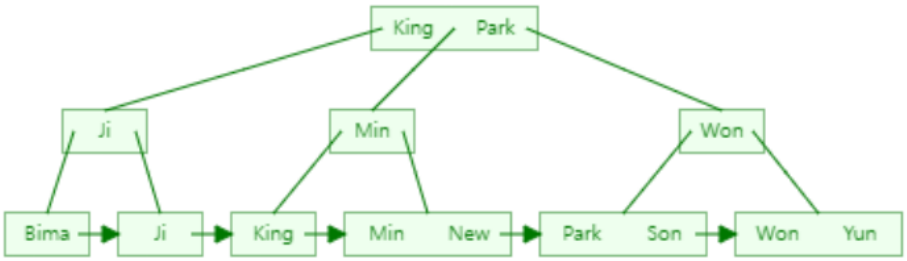
인자로 받은 IndexNode의 요소가 3개인지 판별한다. 3개이면 true를 반환하고 2개 이하면 false를 반환한다.

5. splitIndexNode

아래와 같이 Index Node를 분할하고 재구성한다.



(분할 전)



(분할 후)

splitDataNode와 마찬가지로 분할 할 노드가 root 일 때와 아닐 때로 구별하여 구현하였다.

(root 일 경우)

1. 새로운 root(index node)와 left child 노드(index node)를 생성한다.
2. root의 첫 번째 데이터는 left child 노드에 넣고 두 번째 데이터는 새로운 root 노드에 넣는다.
3. root의 첫 번째 데이터를 지운다.
4. 새로운 root 노드의 key 요소(right child node)는 root로 지정하고 MostLeftChild(left child node)는 left child 노드로 지정한다.
5. root와 left child 노드의 부모를 새로운 root 노드로 지정한다.

(root 가 아닐 경우)

1. left child 노드(Index node)를 생성한다.
2. 기존 Index node의 첫 번째 데이터는 left child 노드에 넣고 두 번째 데이터는 Index node의 parent 노드 Map에 삽입한다.
3. Index node의 첫 번째 데이터를 지운다.
4. Parent 노드에 넣은 데이터 Key의 요소는 Index node로 지정한다.
5. Parent 노드에 넣은 데이터의 이전 데이터의 Key의 요소는 Left child node로 지정한다. 만약 데이터가 첫 번째 데이터라면 MostLeftChild 노드로 지정한다.
6. 만약 parent 노드의 데이터가 3개 이상이면 splitIndexNode함수를 호출한다.
7. 1~6을 tree가 안정적일 때까지 반복한다.

6. SearchRange

SearchRange는 인자로 string start, string end를 받고 start와 end에 해당하는 알파벳 범위의 데이터를 모두 출력한다.

출력 방법은 Key의 앞 글자와 start를 비교하고 트리를 이동하며 범위에 해당하는 Data Node를 찾는다.

그 후 Data Node의 Map의 모든 Key의 앞 글자를 비교하며 범위에 해당하면 Key의 요소로 있는 VaccinationData를 출력한다.

만약, Map안의 모든 데이터를 비교했다면 다음 Data Node로 넘어가서 위 행위를 반복한다.

이는 데이터가 end의 범위를 넘어 갈 때까지 반복한다.

7. Print

Print 함수는 B+ tree안의 모든 데이터를 출력한다.

함수의 동작은 아래와 같다.

1. Data Node가 나올 때까지 MostLeftChild로 이동한다.
2. Data Node 안의 모든 데이터를 출력한다.
3. 다음 Data Node로 이동한다.
4. 2,3을 다음 Data Node가 없을 때까지 반복한다.

8. searchDataNode

searchDataNode는 이름을 인자로 받고 해당하는 이름을 가진 백신 접종 정보를 찾는다.

함수의 동작은 아래와 같다.

1. Root부터 노드의 Key 값과 이름을 비교하며 노드를 이동한다.
2. 마지막에 도착한 Data Node에 이름에 해당하는 Key 값이 있는지 확인한다.
3. 만약 해당하는 Key가 있다면 현재 Data Node를 반환하고, 해당하는 Key가 없다면 NULL을 반환한다.

마지막으로, B+ tree의 메모리 해제는 queue를 이용하여 레벨 순회를 모방해 구현 하였다.
메모리 해제 방법은 아래와 같다.

1. Queue를 사용하여 맨 위 레벨부터 queue에 집어 넣는다.
2. Front 노드의 자식 노드들을 queue에 넣는다.
3. Pop을 진행하고 pop한 노드의 메모리를 해제한다.
4. 2,3을 Index Node를 전부 삭제할 때까지 반복하면 queue에는 모든 Data Node만 남게 된다.
5. queue에서 하나씩 POP 하며 Data Node의 데이터를 먼저 메모리 해제하고 해당 노드도 메모리를 해제한다.
6. 모든 노드의 메모리를 해제한 뒤 B+tree의 메모리를 해제한다.

<AVL tree>

AVL tree는 기존 이진 탐색 트리에서 노드가 한 쪽으로 치우쳐져 있을 때 트리를 탐색할 때, 효율성이 떨어져서 이 문제를 보안하여 만들어졌다.
AVL tree는 Balance Factor라는 변수가 노드 마다 존재한다.
Balance Factor는 왼쪽 자식 노드와 오른쪽 자식 노드를 비교해 한 쪽으로 치우침에 대해 표시해준다.
Balance Factor가 1이면 왼쪽 자식 노드 레벨이 1 더 높은 것이고 Balance Factor가 -1이면 오른쪽 자식 노드가 레벨이 1 더 높은 것이다. 만약 Balance Factor가 0이면, 두 자식 노드가 균형을 이뤘다고 볼 수 있다.
AVL tree는 데이터를 추가하고 -1<=Balance Factor<=1 범위를 넘어가게 되면 로테이션이라는 동작을 통해 Balance Factor가 범위 안에 들어오도록 tree를 재구성한다.

이 프로그램의 AVL tree에서 사용한 함수는 아래와 같다.

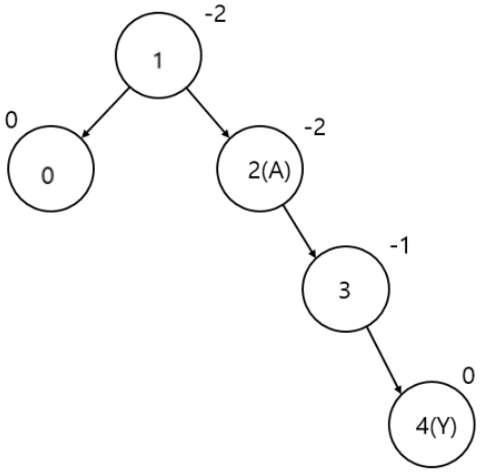
반환 타입	함수	설명
bool	Insert(VaccinationData*)	tree에 데이터 삽입
VaccinationData*	Search(string)	tree안의 데이터 탐색
void	GetVector(vector<VaccinationData*>& v)	tree안의 모든 데이터 Vector에 저장

1. Insert

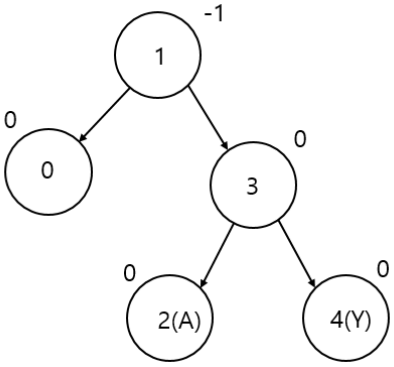
AVL tree의 Insert는 기본 BST의 Insert와 비슷하다 하지만 AVL tree는 노드를 삽입하고 추가적으로 트리의 balance가 맞지 않으면 로테이션을 진행해 Balance를 맞춰준다.
Balance를 맞춰주는 로테이션 동작에는 RR, LL, RL, LR 총 4가지가 있다.

1) RR

RR은 새로 추가한 노드 Y의 위치가 balance가 맞지 않는 노드 A의 오른쪽 자식 노드의 오른쪽 자식 노드 인 경우이다. 아래는 RR 로테이션은 진행하는 예시이다.



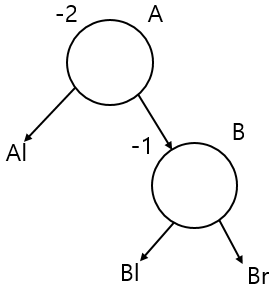
(재구성 전)



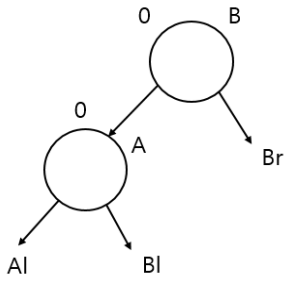
(재구성 후)

위 그림과 같이 RR 로테이션은 A 노드가 Y의 부모 노드의 왼쪽 자식 노드가 되고 A의 부모 노드의 오른쪽 자식 노드는 Y의 부모 노드가 된다. 그리고 그에 맞춰서 Balance Factor도 업데이트를 진행한다.

만약 자식 노드가 있는 경우에는 아래와 같이 노드가 이동하며 자식 노드도 이동하게 된다.



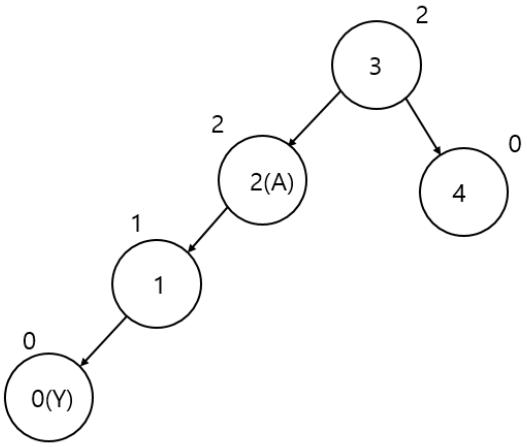
(재구성 전)



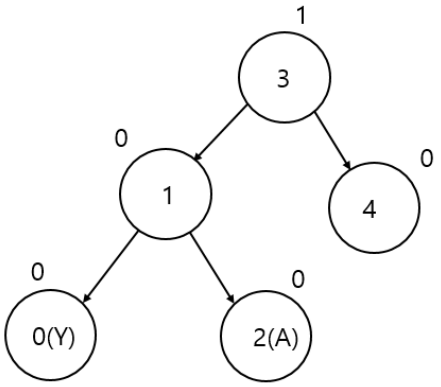
(재구성 후)

2) LL

LL은 새로 추가한 노드 Y의 위치가 balance가 맞지 않는 노드 A의 왼쪽 자식 노드의 왼쪽 자식 노드 인 경우이다. 아래는 LL 로테이션은 진행하는 예시이다.



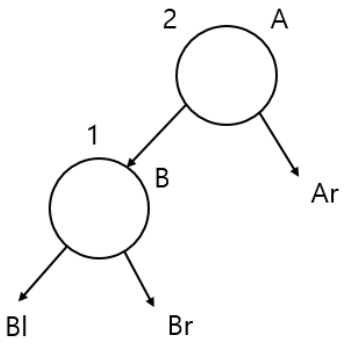
(재구성 전)



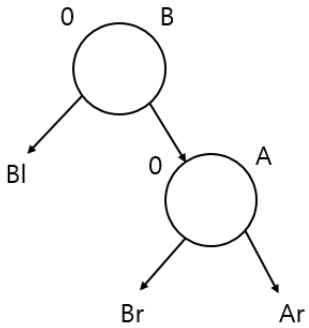
(재구성 후)

위 그림과 같이 LL 로테이션은 RR 로테이션과 반대로 Y 노드의 부모가 A 노드를 오른쪽 자식 노드로 갖고 A노드의 부모가 Y노드의 부모를 왼쪽 자식 노드로 갖는다. 그리고 그에 맞춰서 Balance Factor도 업데이트 해준다.

만약 자식 노드가 있는 경우에는 아래와 같이 노드가 이동하며 자식 노드도 이동하게 된다.



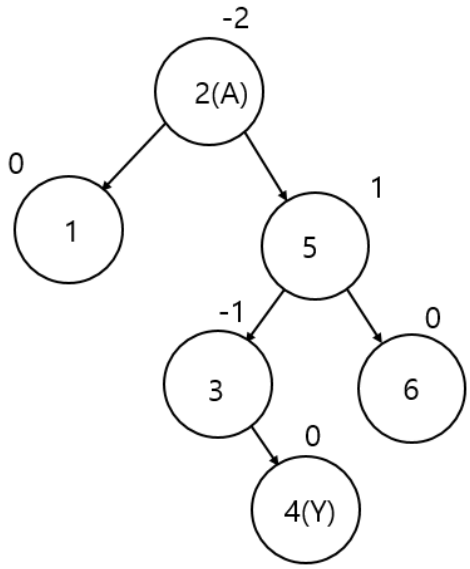
(재구성 전)



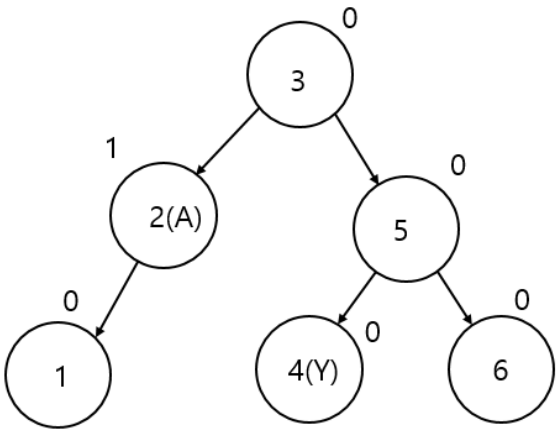
(재구성 후)

3)RL

RL은 새로 추가한 노드 Y의 위치가 balance가 맞지 않는 노드 A의 오른쪽 자식 노드의 왼쪽 자식 노드인 경우이다. 아래는 RL 로테이션의 예시이다.



(재구성 전)



(재구성 후)

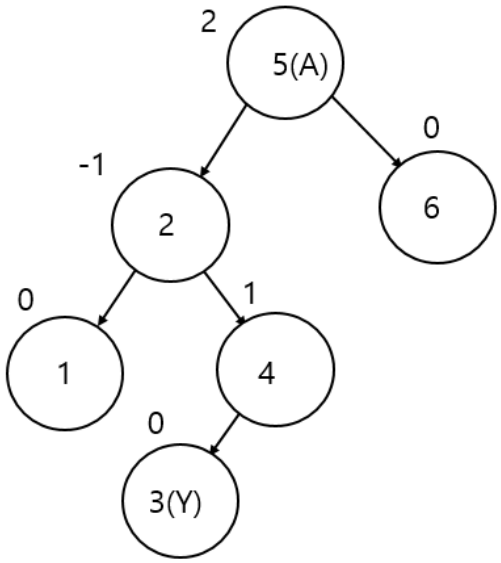
위 그림과 같이 RL 로테이션은 서로 반대되는 로테이션이 두 번 동작해줘야 한다.

위 그림에서는 3번 노드와 5번 노드를 로테이션을 먼저 진행 한 후 다시 3번과 2번을 로테이션을 진행한다.

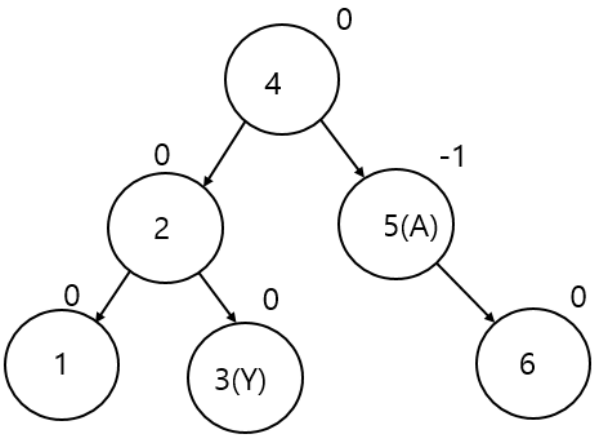
자식 노드의 경우엔 RL은 LL+RR과 같으므로 로테이션을 진행하며 LL과 RR을 동작할 때 와 같이 이동한다.

4)LR

LR은 새로 추가한 노드 Y의 위치가 balance가 맞지 않는 노드 A의 왼쪽 자식 노드의 오른쪽 자식 노드인 경우이다. 아래는 LR 로테이션의 예시이다.



(재구성 전)



(재구성 후)

위 그림과 같이 RL 로테이션은 서로 반대되는 로테이션이 두 번 동작해줘야 한다.
위 그림에서는 2번 노드와 4번 노드를 로테이션을 먼저 진행 한 후 다시 4번 노드와 5번 노드를 로테이션을 진행한다.

자식 노드의 경우엔 RL은 LL+RR과 같으므로 로테이션을 진행하며 LL과 RR을 동작할 때 와 같이 이동한다.

마지막으로 이 프로그램의 AVL tree Insert는 다음과 같이 진행된다.

1. 이름을 기준으로 트리를 탐색하며 노드를 삽입한다.
2. Tree의 balance가 맞는지 확인한다. 만약 맞는다면 함수는 종료한다.
3. Balance가 맞지 않는다면, 경우에 따라서 LL, RR, LR, RL 중 로테이션을 진행한다.

2. Search

Search는 이진 탐색 트리와 같이 트리 안을 이동하며 인자로 받은 이름에 해당하는 데이터를 가진 노드를 찾고 해당 데이터를 반환한다.

3. GetVector

GetVector는 인자로 받은 Vector에 queue를 이용해 레벨 순회를 하며 트리 안의 모든 데이터를 저장한다. GetVector의 동작은 다음과 같다.

1. Root의 자식 노드 들을 queue에 PUSH 한다.
2. Root를 Vector안에 넣는다.
3. front의 자식 노드들을 queue에 PUSH 한다.
4. Front를 Vector에 넣고 POP 한다.
5. 이를 queue가 빌 때까지 반복한다.

Result screen

다음은 검증을 하는데 사용한 Command.txt와 input_data.txt이다.

```
jiwon@ubuntu:~/DS_Project2$ cat command.txt
LOAD
ADD Tom Pfizer 38 Seoul
ADD John Pfizer 17 Seoul
ADD Jiwon Janssen 23 Seoul
ADD Jiwon Janssen 23 Seoul
SEARCH_BP Jiwon
SEARCH_BP C J
SEARCH_AVL Jiwon
VLOAD
VPRINT A
VPRINT B
PRINT_BP
EXITjiwon@ubuntu:~/DS_Project2$
```

```
EXITjiwon@ubuntu:~/DS_Project2$ cat input_data.txt
Denny Pfizer 0 32 Gyeonggi
Tom Pfizer 1 38 Seoul
Emily Moderna 0 21 Incheon
John Pfizer 1 17 Seoul
Erin AstraZeneca 0 51 Daegu
Happy Moderna 1 23 Gwangju
Edward Pfizer 0 98 Busan
Sun AstraZeneca 1 43 Ulsan
Min Moderna 0 38 Daejeon
Yun Pfizer 0 11 Jeju
```

아래는 LOAD 명령어이다. 성공적으로 실행이 되었다.

```
jiwon@ubuntu:~/DS_Project2$ cat log.txt
========== LOAD ==========
Success
==========================
```

아래는 ADD 명령어이다. 3명의 접종자가 잘 추가된 것을 볼 수 있고, 마지막은 Jiwon이 이미 AVL tree에 존재하기 때문에 에러 코드를 출력한다.

```
========== ADD ==========
Tom Pfizer 38 Seoul
============================

========== ADD ==========
John Pfizer 17 Seoul
============================

========== ADD ==========
Jiwon Janssen 23 Seoul
============================

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

아래는 SEARCH_BP이다.
우선 인자가 하나일 때 해당하는 이름을 찾아 그 데이터를 출력한다. 아래와 같은 경우에는 이전에 ADD로 추가한 데이터인 Jiwon을 인자로 명령어를 진행하였고, 결과는 아래와 같다.

```
========== SEARCH_BP ==========
Jiwon Janssen 1 23 Seoul
============================
```

인자가 두개일 때는 인자에 해당하는 범위의 데이터를 모두 출력한다. 아래와 같은 경우에는 C와 J를 인자로 주어 명령어를 실행하였고, 결과는 아래와 같다.

```
========== SEARCH_BP ==========
Denny Pfizer 0 32 Gyeonggi
Edward Pfizer 0 98 Busan
Emily Moderna 0 21 Incheon
Erin AstraZeneca 0 51 Daegu
Happy Moderna 1 23 Gwangju
Jiwon Janssen 1 23 Seoul
John Pfizer 2 17 Seoul
============================
```

SEARCH_AVL은 이름을 인자로 받아 AVL tree에서 해당하는 이름을 찾아 출력한다. 이 전에 ADD로 Janssen을 접종한 Jiwon은 접종 완료 횟수가 1이므로 AVL tree에 들어가게 된다. 따라서, SEARCH_AVL로 Jiwon을 찾아 보았고 결과는 아래와 같다.

```
========== SEARCH_AVL ==========
Jiwon Janssen 1 23 Seoul
============================
```

다음은 VLOAD와 VPRINT 명령어이다. VLOAD로 AVL tree안의 데이터를 모두 저장하고 VPRINT의 인자에 따라서 Vector안의 데이터를 정렬하여 출력한다. 결과는 아래와 같다.

```
========== VLOAD ==========
Success
============================

========== VPRINT A ==========
Jiwon Janssen 1 23 Seoul
John Pfizer 2 17 Seoul
Tom Pfizer 2 38 Seoul
============================

========== VPRINT B ==========
Tom Pfizer 2 38 Seoul
Jiwon Janssen 1 23 Seoul
John Pfizer 2 17 Seoul
============================
```

PRINT_BP는 B+ tree안의 모든 데이터를 출력한다. 아래 결과를 보면 Input_data.txt에 있던 데이터와 ADD로 추가한 데이터 모두 다 잘 출력되는 것을 볼 수 있다.

```
========== PRINT_BP ==========
Denny Pfizer 0 32 Gyeonggi
Edward Pfizer 0 98 Busan
Emily Moderna 0 21 Incheon
Erin AstraZeneca 0 51 Daegu
Happy Moderna 1 23 Gwangju
Jiwon Janssen 1 23 Seoul
John Pfizer 2 17 Seoul
Min Moderna 0 38 Daejeon
Sun AstraZeneca 1 43 Ulsan
Tom Pfizer 2 38 Seoul
Yun Pfizer 0 11 Jeju
============================
```

EXIT는 프로그램내 모든 메모리를 해제하고 프로그램을 종료한다. 아래는 EXIT 결과와 valgrind를 이용한 메모리 누수가 없는지 체크 한 것이다

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

```
==7929== HEAP SUMMARY:
==7929==     in use at exit: 0 bytes in 0 blocks
==7929==   total heap usage: 98 allocs, 98 frees, 235,504 bytes allocated
==7929==
==7929== All heap blocks were freed -- no leaks are possible
==7929==
==7929== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==7929== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

.

고찰

이 프로젝트는 B+ tree와 AVL tree를 이용해 백신 접종 관리 프로그램을 구현하는 것이다. 이 프로그램을 구현하는 데 핵심 키워드는 B+ tree와 AVL tree의 Insert 인 것 같았다. 실제로 구현 중 문제가 생긴 부분 또한 모두 Insert 부분 이었다.

B+ tree 부분에서 문제는 PRINT_BP명령어를 실행 했을 때, 중간에 출력되지 않는 데이터가 발생하였다. 이 문제에 대하여 먼저 PRINT_BP를 검증하였으나 문제가 없어 보였고, 그래서 두 가지 가설을 세워 문제에 접근을 하였다.

첫 번째는 Data Node 끼리의 연결이 올바르지 않아 중간에 데이터가 뛰어 넘기고 연결된 것이었고, 두 번째가 Insert 과정에서 적절한 Data Node를 찾아 데이터를 삽입하는데 이 Data Node를 찾는 과정에서 문제가 발생 했을 것이라고 생각하였다.

그 결과는 전자가 정답 이었다. Data Node를 분할 할 때, 분할 한 두 노드는 연결을 해주었지만 왼쪽 자식 노드와 분할 하기 이전의 Data Node를 연결하지 않아 PRINT_BP를 진행 할 때, 왼쪽 자식 노드를 뛰어넘고 출력하는 결과가 발생 하였다.

그 다음으로 문제가 생겼던 부분은 VPRINT를 검증하는 당시에 발생하였다. VPRINT를 검증할 때, AVL tree에 존재해야 함에도 VPRINT에 출력되지 않는 결과가 나왔고, 그 문제를 해결하기 위해 처음에는 ADD 명령어에서 AVL tree에 Insert로 넘어가는지 확인 하였고, 그 부분은 의도한대로 동작을 하였다. 따라서, AVL tree의 Insert 부분에 문제가 있다고 판단하고, 데이터를 하나씩 넣었을 때 결과를 확인하고, 데이터 구조 설계 강의 자료를 참고해 오류를 찾았다.

그 결과, 오류가 발생한 부분은 로테이션을 할 때, root를 다른 노드로 바꾸지 않아 root가 AVL tree의 가지 노드를 가리키는 상황이 발생하였고, 당연히 root부터 아래로 내려가며 VLOAD를 진행 하므로 Vector에 들어가지 않는 데이터가 발생 하였다.

마지막으로 오류는 아니지만 프로그램을 작성하며 가장 고민했던 부분은 메모리 누수를 잡는 점이다. AVL tree는 구조 자체는 BST와 같기 때문에 후위 순회를 하며 메모리를 해제하면 됐기에 쉬웠지만 B+ tree는 차수가 다르기도 하고 노드를 가리킬 때, Map을 사용하고, 선언 할 때, 참조 변수를 모두 BpNode로 선언했지만, Data Node와 Index Node는 다르기 때문에 노드 안의 데이터를 지우는 명령어도 다르기 때문에 접근하기 힘들었다.

따라서, 처음에 생각한 방법은 PRINT_BP를 수행하듯이 Data Node를 먼저 모두 지우고 Index Node를 지우려 하였지만, 조금 비효율적인 것 같아서 나중에는 VLOAD에서 레벨 순회를 하듯이 데이터를 저장한 것을 떠올리고 queue를 이용해 Index Node메모리를 먼저 해제하고 마지막 잎 노드인 Data Node를 모두 메모리를 해제하고 메모리 누수를 모두 막았다.