

# **Data Structure Project**

## **Project #2**

**담당교수 : 이기훈 교수님**

**제출일 : 2018. 11. 11.**

**학과 : 컴퓨터 정보 공학과**

**학번 : 2017202008**

**이름 : 남 건 우**

## 1. Introduction

본 프로젝트는 주식 정보 검색 시스템을 구축하는 것이다. 이 시스템은 다양한 종목들의 주식 정보를 저장하고, 다양한 기준을 적용하여 효율적으로 검사하는 프로그램이다. AVL tree와 B+ tree를 사용해서 프로그램을 구축하는 것으로, 주식들의 주가데이터를 수익률과 종목 이름 순서로 정렬하여 각 자료 구조에 저장한다. B+ tree는 전체 데이터 출력 기능, 특정 수익률의 범위 안에 있는 종목들을 검색하는 기능을 가지고 있다. 또한, AVL tree는 검색한 데이터를 Heap을 통해 수익률 순으로 출력하는 기능을 가지며 B+ tree와 마찬가지로 전체 데이터 출력, 검색 기능을 갖고 있다.

주식 정보 검색 시스템은 'stock\_list.txt' 텍스트 파일을 통해 저장되어 있는 주식 정보를 읽어 두 개의 자료구조에 이 데이터들을 저장한다. 텍스트 파일은 고유번호, 종목 이름, 시가, 종가, 거래량 순서로 저장되어 있고 시가 종가의 계산을 통해 수익률을 계산하여 저장한다. 이 때, 항상 소수점 둘째 자리까지 반올림하여 저장하는데 텍스트 파일의 중복된 고유번호의 입력은 없다고 가정한다.

B+ tree는 수익률을 기준으로 객체 정보를 저장하고 AVL tree는 종목 이름을 기준으로 정렬한다. B+ tree는 Base class BpTreeNode를 통해 BpTreeIndexNode와 BpTreeDataNode는 상속된다. BpTreeIndexNode는 비 단말 노드로 m(ORDER)의 차수를 갖고 BpTreeData Node는 단말노드로 수익률의 값과 vector를 갖는다. 이 두 종류의 노드는 map 컨테이너 형태를 지닌다.

다음은 해당 프로젝트를 구현하면서 사용한 함수의 이름이다. 7가지의 함수의 이름은 다음과 같고 기능은 이러하다.

### (1) LOAD

처음 main문 안에서 Run 함수는 한개의 매개 변수로 "command.txt"를 받는다. 이후, Manager.cpp에서 'command.txt' 파일에 있는 명령어들을 읽게 된다. 주식 정보 검색 프로그램의 종목들의 정보를 불러오는 명령어로 "stock\_list.txt"(텍스트 파일)을 읽어 두 개의 자료 구조 B+ tree, AVL tree를 구성하는데 사용한다. 고유번호의 범위는 반드시 1000~9999에 있고 주가 데이터 이외의 다른 문자열의 입력은 없다고 한다. 중복된 고유번호를 입력하고 3개의 주가 데이터와 고유 번호, 종목 중에서 하나라도 입력을 하지 않는 경우의 수는 없다고 한다. 예외 처리는 텍스트 파일이 존재하지 않거나, 기존에 트리가 구성되어 있으면 'log.txt'(출력 파일)에 에러 코드를 출력하게 된다. LOAD 함수는 LOAD라는 명령어를 읽을 때, 구현이 된다. strcmp함수를 써서 각 명령어를 파악하여 함수들을 실행시키게 되는데, LOAD가 두 번 'command.txt' 텍스트 파일에 들어 있을 경우 에러 코드를 뽑아 내야 하므로 이를 구별할 수 있는 flag인 identifier로 'load'를 사용한다. 또한 Manager의 LOAD함수의 return 값을 IsWork로 받아줌으로써 'log.txt'(출력 파일)

에 결과를 입력해주게 된다. LOAD함수 안에서 strtok함수를 이용하여 " "를 기준으로 잘라 StockData에 대입해준다. 이렇게 대입된 데이터들은 avl->insert()와 bp->insert()의 첫번째 인자로 들어가 작업을 수행하게 된다. 이것이 성공적으로 끝이 나면 true를 반환해 성공 코드를 출력하는 것이다. 반면에 실패하면 에러 코드를 출력하게 된다. 이렇게 반환된 값은 IsWork를 통해 받아 성공 코드를 출력해야 하는지 에러 코드를 출력해야 하는지 결정을 한다.

## (2) SEARCH\_AVL

'SEARCH\_AVL 이름'이라는 형태를 가지고 그 이름을 가진 주가 데이터를 검색하는 명령어로 종목의 이름을 검색하여 'log.txt'(출력 파일)에 출력한다. 한글이 아닌 이름은 없다 가정하고 입력한 이름의 데이터가 존재하지 않거나, 이름을 미입력 하게 된다면 에러 코드를 출력한다. SEARCH\_AVL 함수는 strtok를 사용하여 뒤에 있는 종목의 이름을 받아준다. 이렇게 받아진 이름은 AVLtree.cpp 파일에 있는 SEARCH\_AVL함수에 들어가게 되고 종목을 기준으로 sorting된 AVL tree에서 자신의 값을 찾게 된다. 만약 받아진 이름이 해당 이름과 비교 되어지는 이름보다 작으면 왼쪽으로 가고 크면 오른쪽으로 간다. 이러한 과정들을 반복하면서 이루어지다가 자신과 같은 이름을 가진 노드를 만나게 되면 이런 과정을 중지시키고 함수를 종료하여 true를 반환하고 해당 이름을 가진 노드의 데이터를 전부 출력시켜준다. 또한 이렇게 찾은 이름의 노드의 데이터는 Heap이라는 vector안에 들어가는데 이 vector의 자료 형은 pair< pair<double, int>, StockData\* >이다. 이렇게 만들어준 vector들은 RANK라는 명령어가 나올 때 사용된다.

## (3) SEARCH\_BP

SEARCH\_BP옆에 있는 2개의 인자를 받아 함수를 실행한다. 입력한 수익률 범위에 속하는 종목들을 B+-tree에서 검색한 후, 해당 종목의 정보를 수익률에 대한 내림차순으로 'log.txt'(출력 파일)에 출력한다. 이 때, 수익률의 범위 란, -1~100까지 숫자를 의미한다. 수익률을 입력 할 때, 소수점 둘째자리까지 입력을 하며 범위에 맞지 않는 수익률의 입력과 시작보다 끝이 작은 경우는 없다고 설정한다. 이렇게 틀을 맞추면서 입력한 조건에 부합하는 종목이 존재하지 않거나, 시작이나 끝 범위 중 어느 하나라도 입력되지 않는 다면 에러 코드를 출력하게 된다.

## (4) RANK

AVL tree의 SEARCH\_AVL을 실행했던 종목들을 수익률의 내림차순으로 정렬하되, 수익률이 같을 때에는 고유번호의 오름차순으로 출력하게 된다. SEARCH\_AVL을 실행하게 되었을 때, Heap에 데이터들이 들어가게 되고 그 데이터들을 이용하여 RANK함수를 실행하게 된다. Heap에 있었던 종목들의 수익률 정보가 삭제되면서

'log.txt'(출력 파일)에 출력한다. 또한 한국어가 아닌 다른 나라의 언어 입력은 없다고 가정한다. Heap에 저장되어 있는 데이터가 없을 경우, 'log.txt'(출력 파일)에 에러 코드를 출력한다.

(5) PRINT\_AVL

AVL tree에 저장되어 있는 모든 정보를 출력하는 명령어로 'log.txt'(출력 파일)에 종목 이름을 오름차순으로 나열한다. 이 때, AVL tree에 저장되어 있는 정보가 없을 경우, 'log.txt'(출력 파일)에 에러 코드를 출력한다.

(6) PRINT\_BP

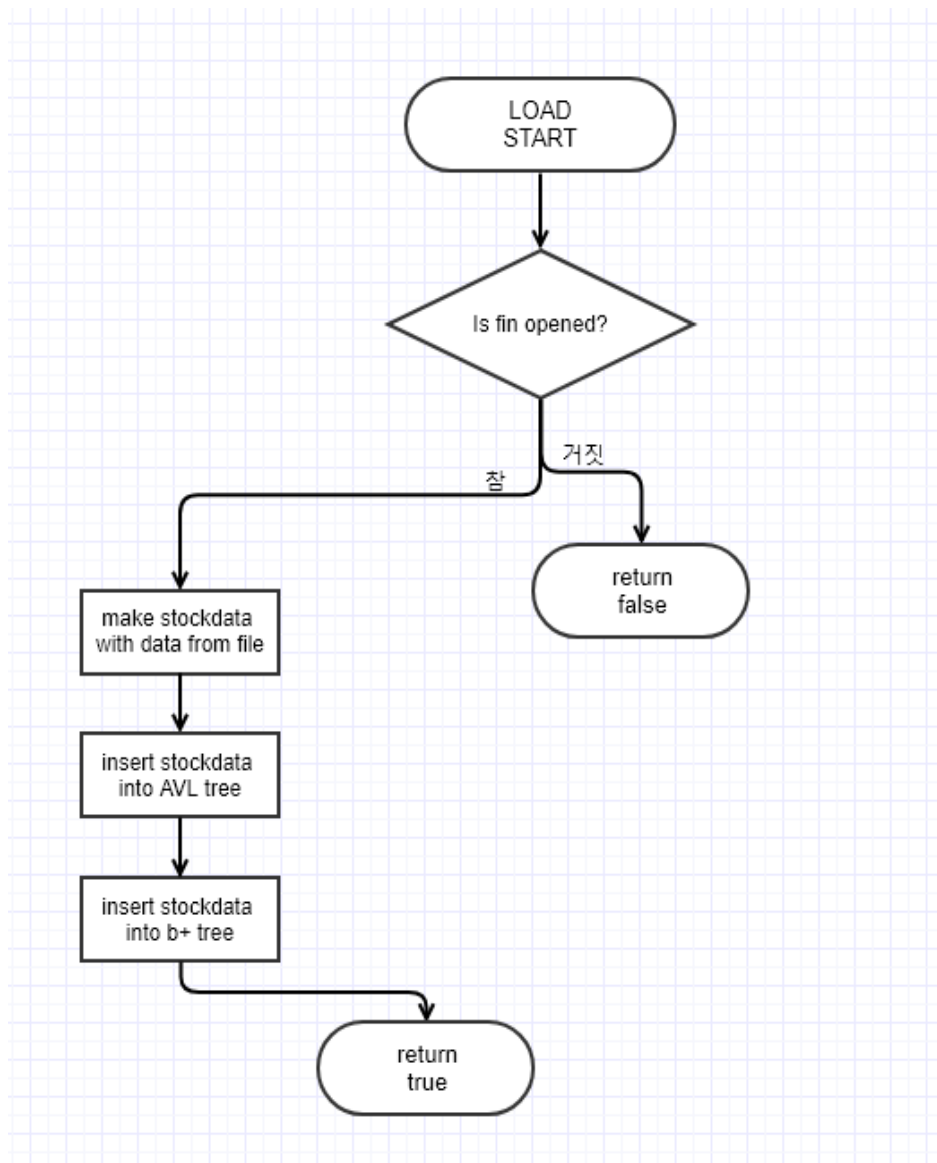
B+ tree에 저장되어 있는 모든 정보를 출력하는 명령어로 'log.txt'(출력 파일)에 수익률의 내림차순으로 정렬한다. 수익률이 같을 경우, 고유번호의 내림차순으로 정렬한다. 이 때, B+ tree에 저장되어 있는 정보가 없을 경우, 'log.txt'(출력 파일)에 에러 코드를 출력한다.

(7) EXIT

주식 정보 검색 프로그램을 종료하는 명령어로, 종료 시 할당된 메모리를 모두 해체한다.

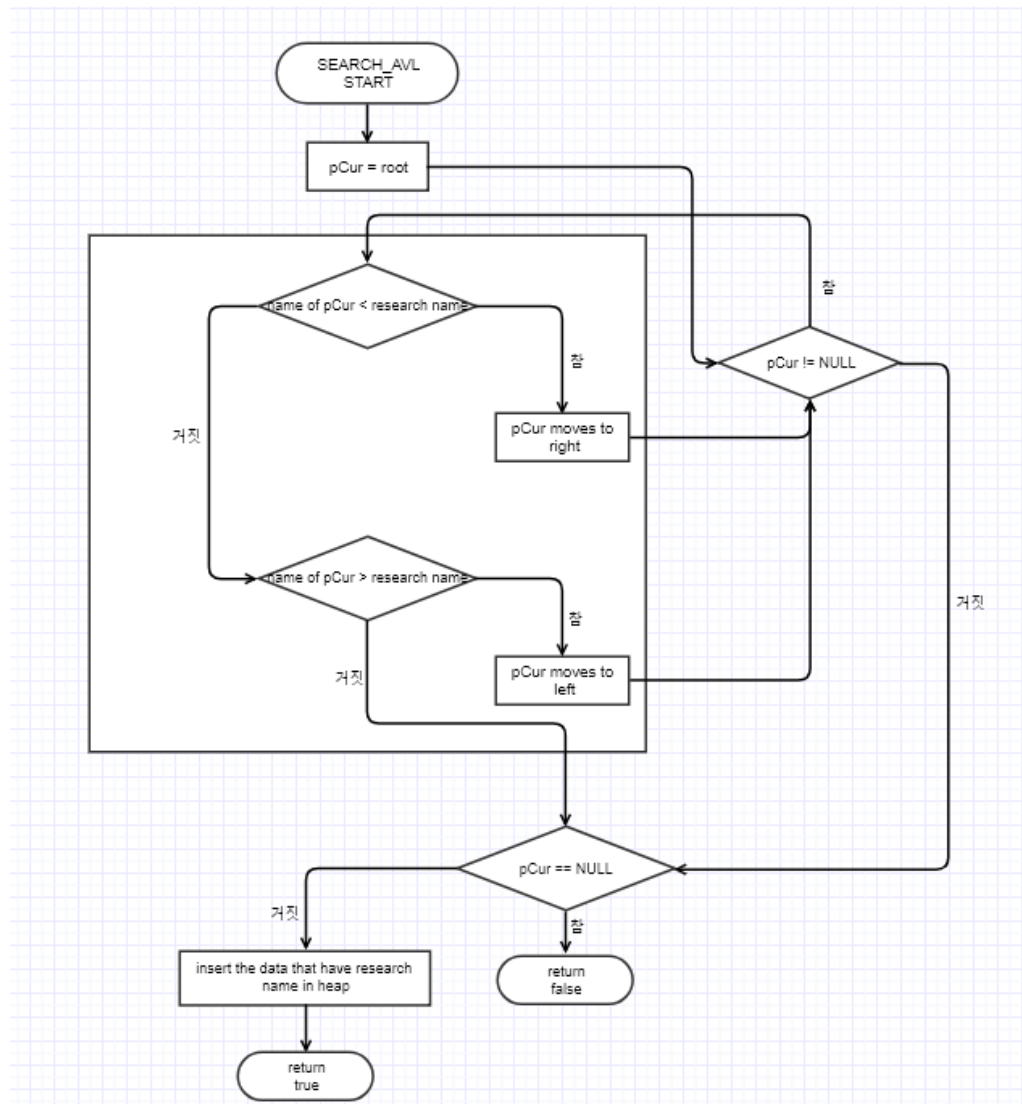
## 2. Flow Chart

### (1) LOAD



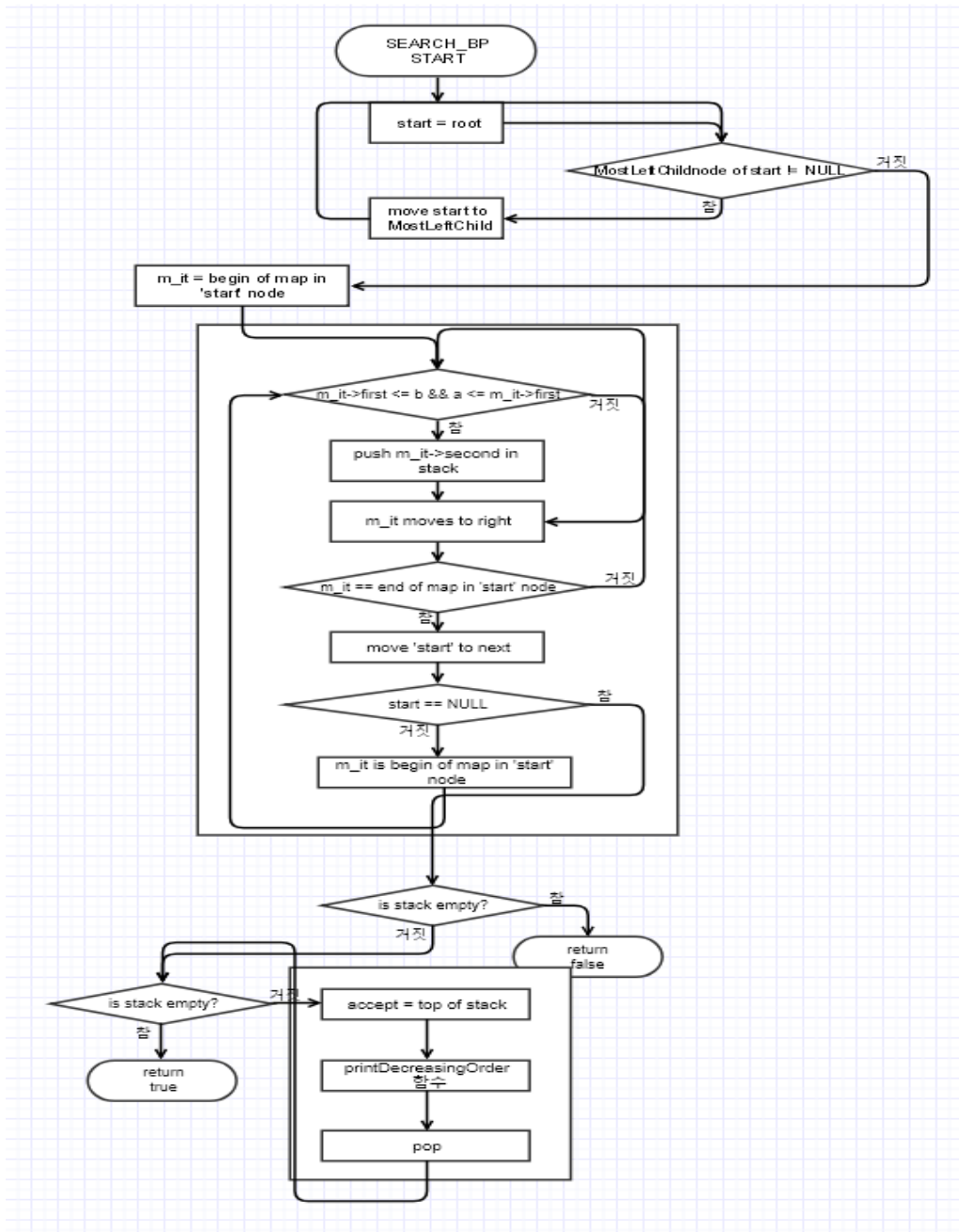
LOAD 함수는 파일과의 스트림이 제대로 연결되었는지 확인하는 'stream fin'을 사용하여 파일과의 연결 여부를 판단한다. 'fin'이 1이라면 연결이 잘되었다는 것을 의미하고 0이라면 연결되지 않은 것을 의미한다. 연결이 잘 되었다면 파일로부터 받아온 data를 이용하여 stockdata를 만들고 이것을 2개의 자료구조, AVL tree 와 B+tree에 insert시켜주고 true를 반환하게 된다. 하지만 연결이 잘되지 않았다면 파일로부터 데이터를 받아올 수 없는 상태가 됨으로 거짓을 반환한다.

## (2) SEARCH\_AVL



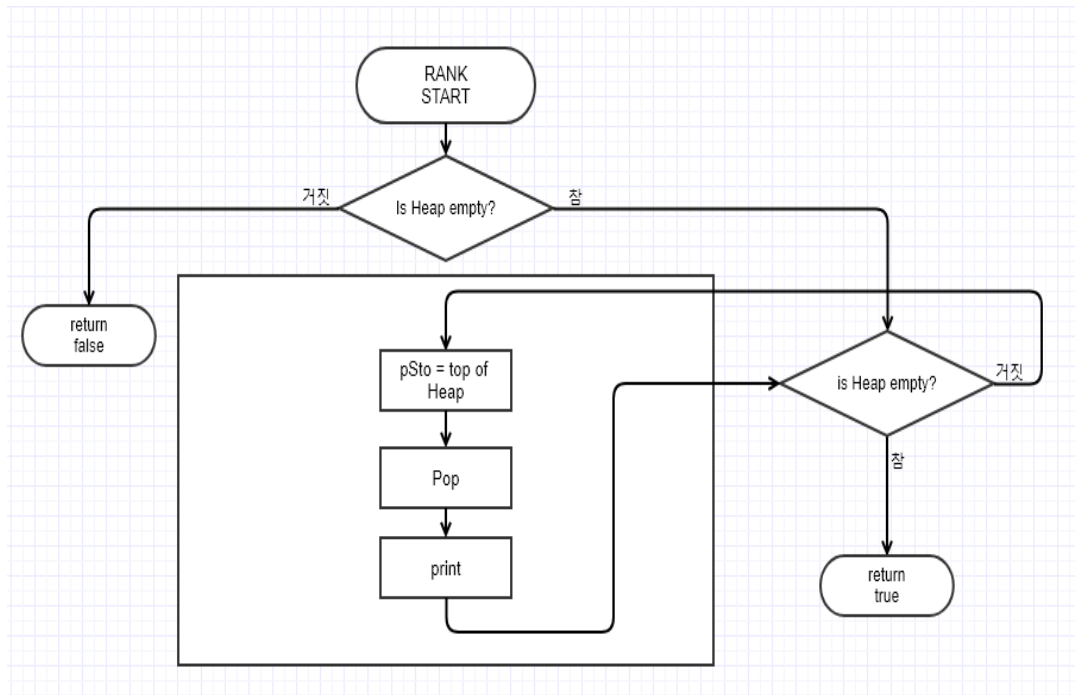
pCur은 AVL tree를 순회하는 AVLnode이다. pCur에 root를 넣어주면 root를 가리키게된다. 이렇게 첫번째 요소를 가리킨 pCur은 자신이 가지고 있는 이름과 앞서 받은 찾을 노드의 대소 관계를 비교하여 이동한다. 만약 pCur이 가리키고있는 노드의 이름이 찾을 노드보다 작으면 왼쪽으로 이동하고 크면 오른쪽으로 이동하게 된다. 이 과정을 반복하게 되면서 찾을 노드의 위치를 파악하게 되는데 만약 pCur이 NULL이 될 때까지 찾지 못한다면 찾는 노드가 없다는 의미임으로 false를 반환한다. 하지만 NULL이 되기 전 까지 pCur이 가리키고있는 노드의 이름이 찾을 노드와 같을 때, 반복문을 탈출하고 이 경우 pCur이 NULL이 아니다.NULL이 아니란 말은 노드를 찾았다는 의미임으로 heap에 찾은 노드의 데이터를 삽입해주고 true를 반환하여 종료합니다.

(3) SEARCH\_BP



Bptreeindexnode를 거쳐 Bptreedatanode로 내려가 vector를 받아 stack에 넣어 준 후, 나중에 넣은 걸 처음으로 나오게끔 만들어주면 내림차순이 가능하다. 또한 vector에 있는 component 또한 내림차순으로 정렬 해주기 위해서는 뒤에서부터 출력 해주면 되기 때문에 printDecreasingOrder 함수를 실행시켜준다. Bptreedatanode로 내려가기 위해서는 반복문을 통해 start를 MostLeftChildnode로 이동시키고 start의 MostLeftChildnode가 NULL이 될 때까지 반복문을 실행한다.

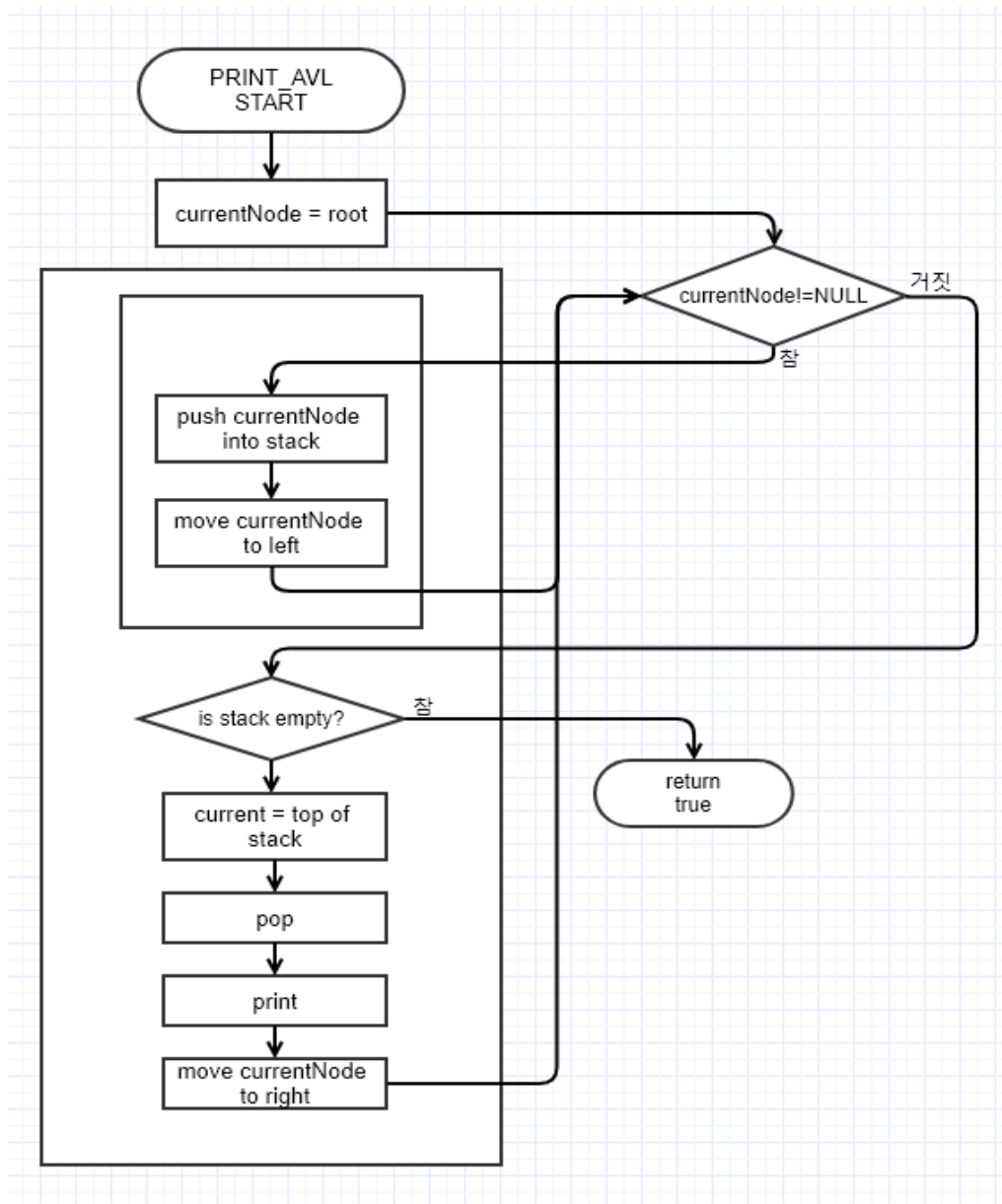
#### (4) RANK



SEARCH\_AVL에서 찾아진 노드들이 Heap에 저장되어 있다. 이 Heap에서 하나씩 제거해주면서 저장되어 있던 원소들을 빼주어야 한다. 그럼으로 처음에 Heap안이 비었는지 확인해야하고 아무것도 없으면 false를 반환하고 있다면 반복 문 안으로 들어 가게 된다. Heap이 빌 때까지 Heap의 제일 위를 pSto에 넣어주고 Heap의 맨 위를 Pop (삭제)시킨 후 print를 함으로써 하나를 출력하고 하나를 제거해주는 방식을 택한다. Heap이 빌 때까지 이 과정이 반복됨으로 pSto는 Heap의 가장 위에 있는 것을 가리키고 삭제시킨 후 출력한다. 만약 이 과정이 잘 실행되어 Heap이 비었다면 true를 반환하게 된다.

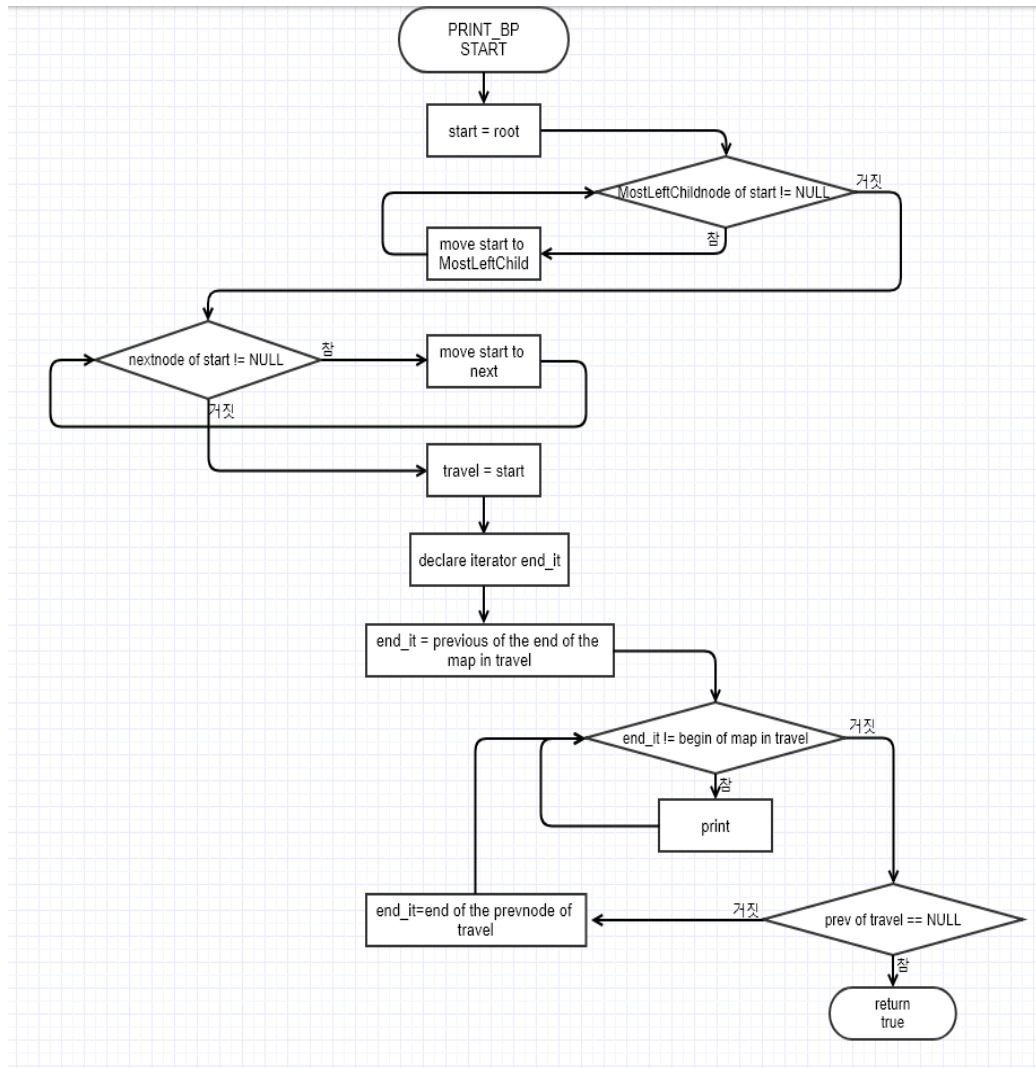


(5) PRINT\_AVL



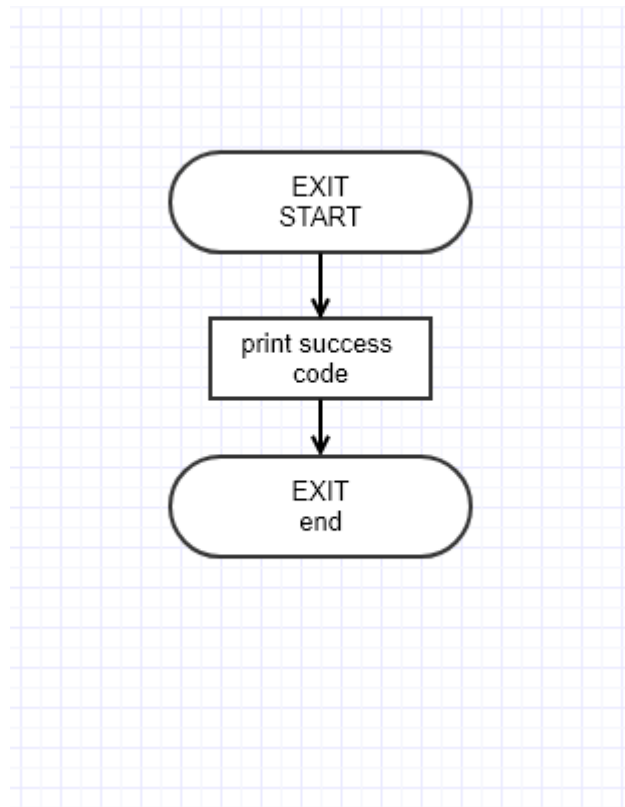
AVL tree에 저장되어있는 모든 데이터를 출력시켜 주기 위해서는 currentNode를 사용해 root부터 순회 시켜 준다. 오름차순으로 정렬 시켜 주기 위해서는 비재귀를 통한 inorder방법을 사용하였다. currentNode가 NULL이 아닐 때 까지 스택에 currentNode을 push하고 왼쪽으로 currentNode을 보내준다. 이때 currentNode가 NULL이되면 stack이 빌 때까지 currentNode에 top을 받고 없애주고 출력하며 currentNode를 오른쪽으로 보낸다. 이것을 반복해서 stack을 사용한 inorder를 구현할 수 있다.

(6) PRINT\_BP



Start는 root에서 시작하여 B+ tree의 datanode에 있는 vector의 내용을 출력하는 것을 도와주는 순회 노드이다. Start의 MostLeftChild가 NULL이 될 때까지, start를 MostLeftChild로 보내준다. 그리고 start의 nextnode가 NULL이 될 때까지, next로 start를 이동시켜준다. Start->getNext가 NULL인 경우 start와 같은 travel을 생성해준다. 'end\_it'을 선언해준 후, travel에 있는 map의 끝의 전의 값을 대입시키고 travel에 있는 첫번째요소거 'end\_it'이 될 때 까지 반복문을 돌려주는데 만약 'end\_it'이 'begin'에 도착한다면 한번 더 출력을 해준 후 true를 반환한다.

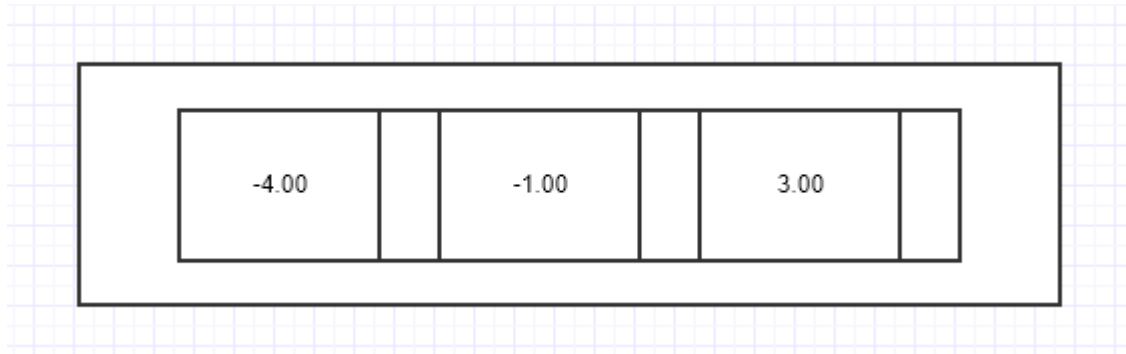
(7) EXIT



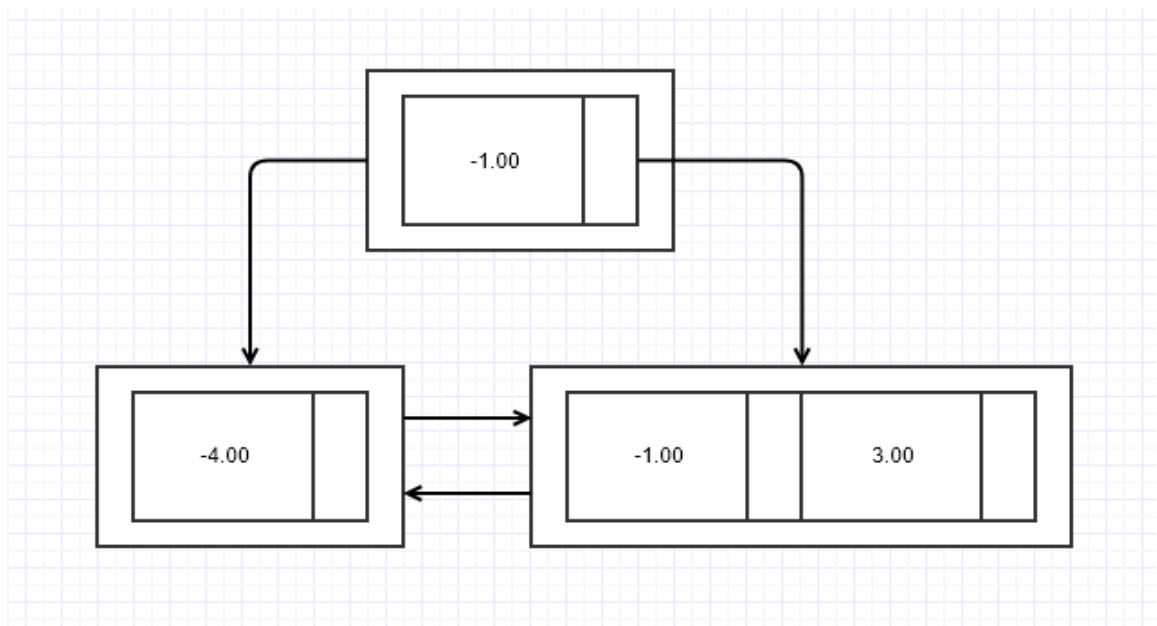
Command.txt에 EXIT이 입력되었을 경우, success code 출력 후 Exit을 탈출한다.

### 3. Algorithm

.(1) B+ tree 삽입 (Assume bporde=3)



Datanode가 3개가 되면 splitdatanode 함수가 실행이 되는데 가운데의 수익률이 복사되고 indexnode가 새로생기고 가운데의 map component부터 map->end()까지 새로운 data node를 만들어 이동시킨다. 다음은 splitdatanode가 실행되었을 때, 나타나는 B+ tree의 형태이다.

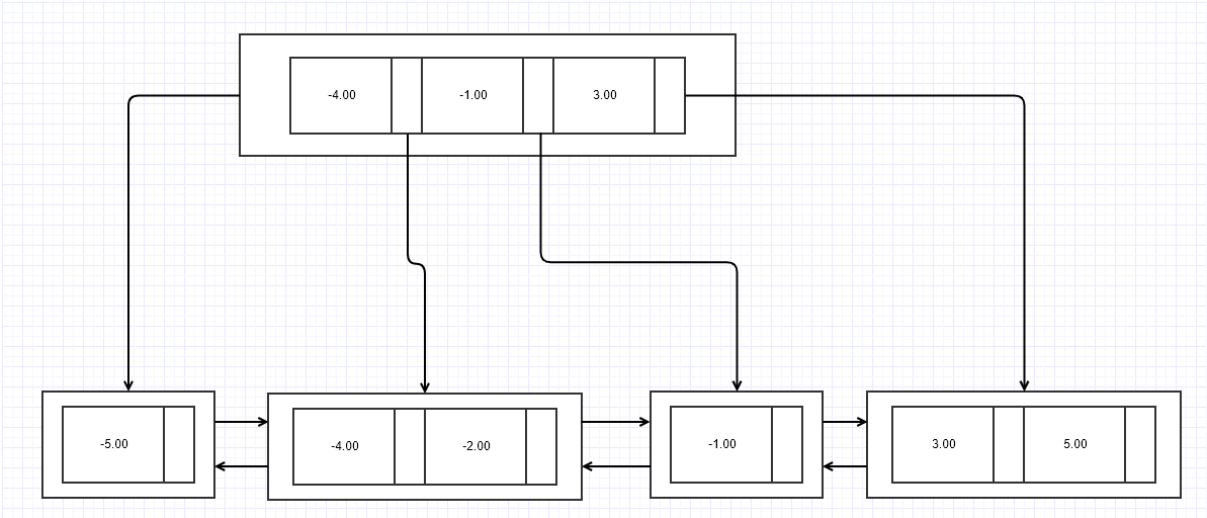


다음은 splitdatanode 함수를 실행한 결과의 B+ tree이다. 가운데의 위치 처럼 올라가는 map의 위치는 다음과 같은 식으로 일반화 된다.

$$\text{ceil}((\text{order} - 1) / 2.0) + 1$$

이 위치에 있는 map의 component의 first는 새롭게 할당되는 indexnode로 복사되어 처음에 있었던 datanode를 MostLeftChild로 가리켜주고 Indexnode.second는 새롭게 할당된

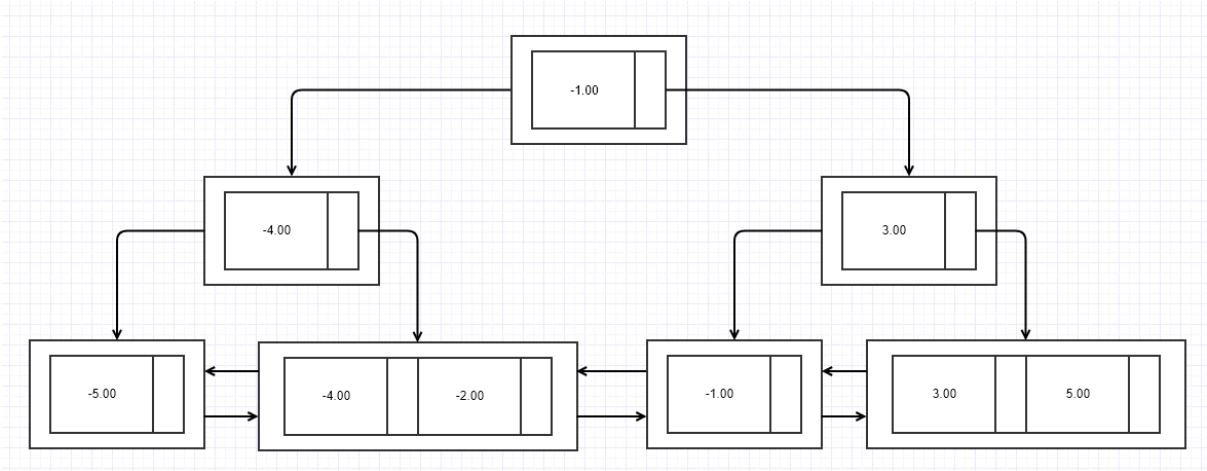
datanode를 가리켜준다. 다음은 splitindexnode 함수를 실행하기 전 B+ tree의 상태이다.



위의 indexnode의 map의 size가 bpointer 보다 많을 때, split이 일어나는데 위로 올라가는 map의 component 위치는 다음과 같은 식을 가진다.

$$\text{ceil}(\text{order} / 2.0)$$

다음은 splitindexnode 함수를 실행시켰을 때, B+ tree의 상태를 나타낸 상태이다.

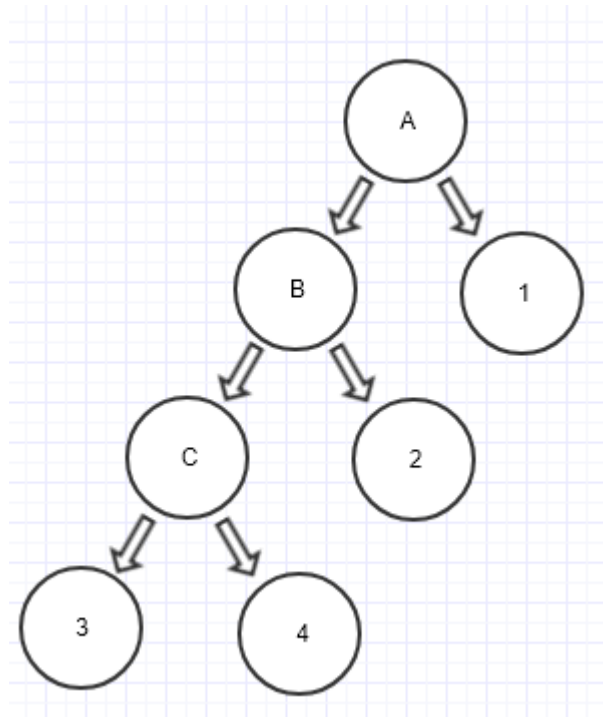


Indexnode에 있는 split이 일어나는데 위로 올라가는 map의 component의 second를 다음 component의 MostLeftChild로 옮겨준다. 새로 할당된 indexnode는 남아있는 indexnode와 새로 할당된 childindexnode를 가리키게 된다.

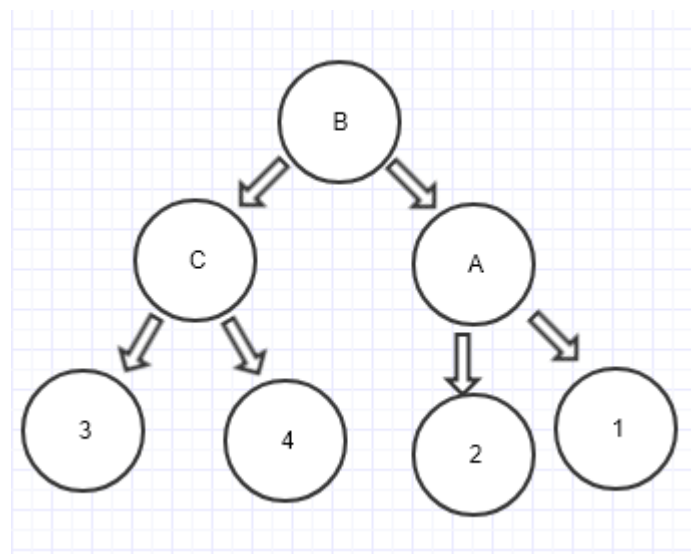
## (2) AVL tree 삽입

AVL tree는 balance factor에 의해서 트리의 모양이 조정되는 tree를 일컫는다. Balance factor의 값이 -1, 0, 1이 아닌 다른 수라면 balance factor를 -1, 0, 1이 되는 노드로 만들어주어야 한다. 이렇게 상황에 따라 만들어 줄 수 있는 방법에는 총 4가지의 방법이 있다.

첫 번째 방법은 LL rotation이 있다. 이 방법은 다음과 같은 상황에서 사용하며 -1, 0, 1이 아닌 balance factor를 -1, 0, 1로 만들어 준다.

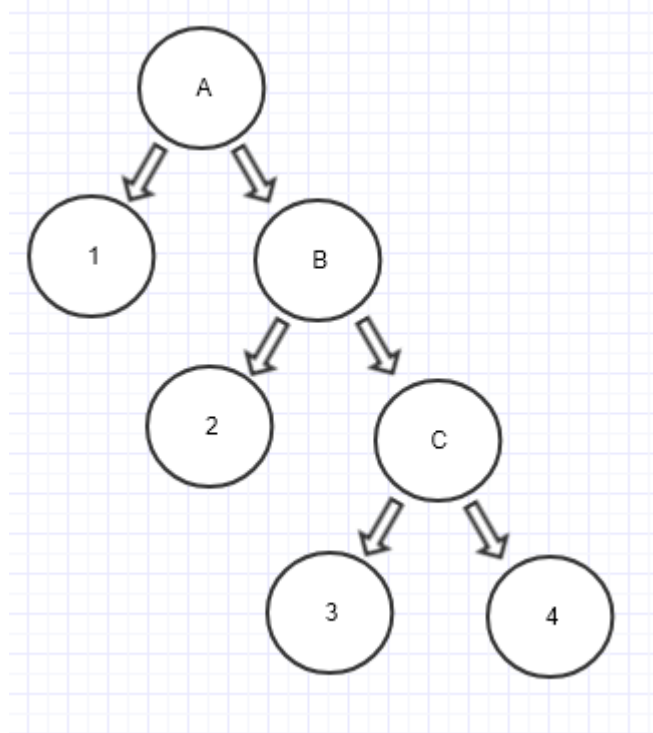


다음은 LL rotation을 사용하고 사용한 후의 tree는 다음과 같이 변형된다. 이에 따라, balance factor이 바뀐다. 'C'의 bf는 0, 'B'의 bf는 1, 'A'의 bf는 2이다.

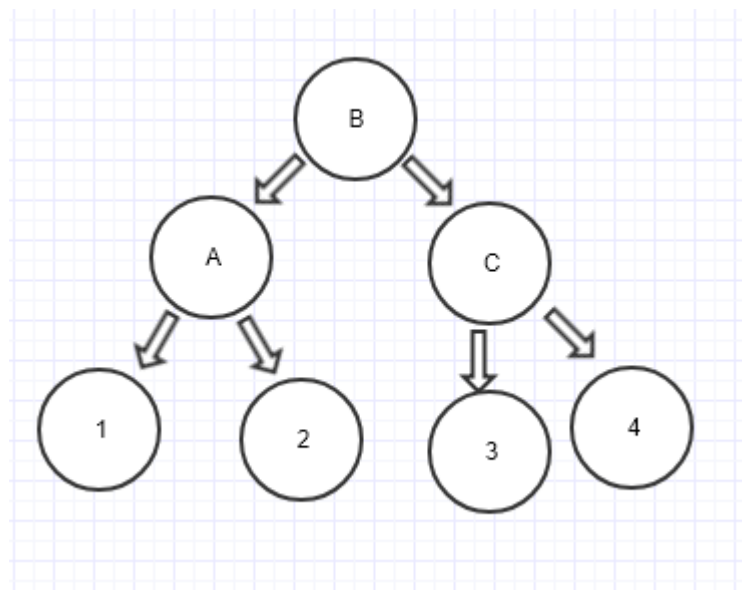


다음과 같이 LL rotation을 사용하여 모든 A, B, C의 bf를 0으로 바꿔 주었다.

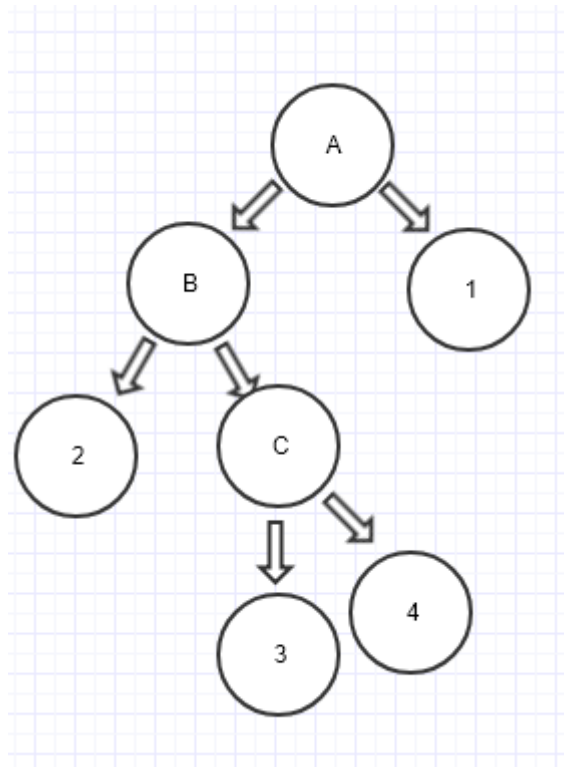
두 번째 방법은 RR rotation이 있다. 이 방법은 다음과 같은 상황에서 사용하며 -1, 0, 1이 아닌 balance factor을 -1, 0, 1로 만들어 준다.



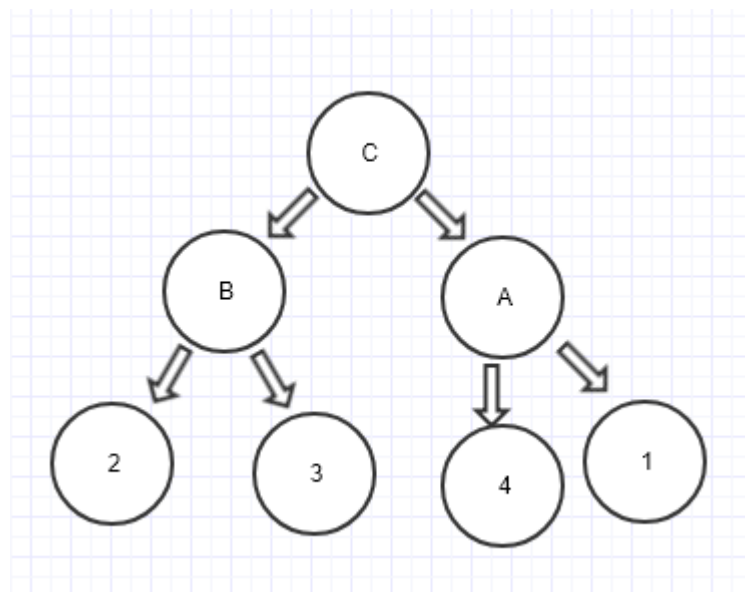
다음은 RR rotation을 사용하고 사용한 후의 tree는 다음과 같이 변형된다. 이에 따라, balance factor이 바뀐다. 'C'의 bf는 0, 'B'의 bf는 -1, 'A'의 bf는 -2이다.



세 번째 방법은 LR rotation이 있다. 이 방법은 다음과 같은 상황에서 사용하며 -1, 0, 1이 아닌 balance factor을 -1, 0, 1로 만들어 준다.



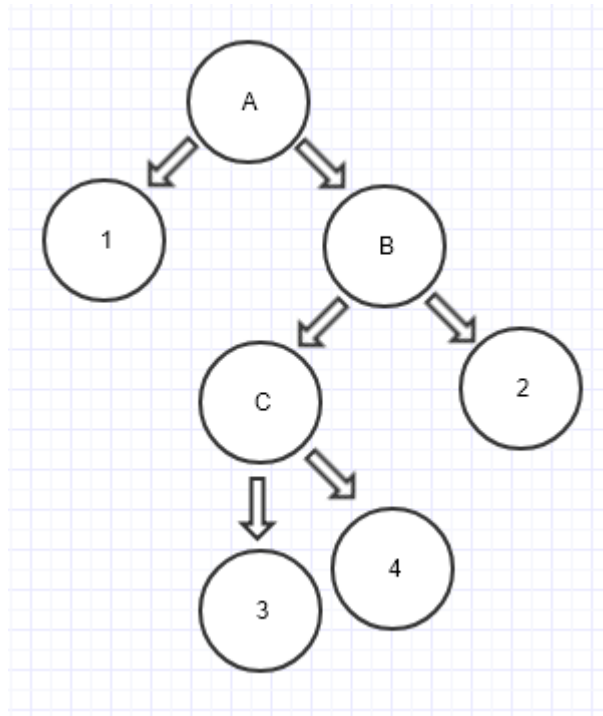
다음은 LR rotation 사용하고 사용한 후의 tree는 다음과 같이 변형된다.



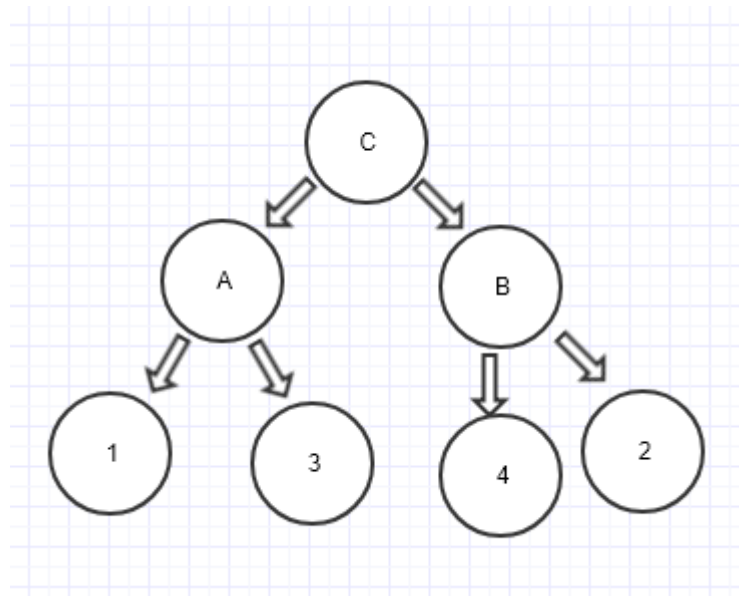
RR rotation을 사용하고 LL rotation을 사용하게되면 다음과 같은 형태의 B+tree가 나오게 된다. LR rotation은 RR rotation을 한 후 재배열된 트리로 다시 LL rotation을 사용한 것과 같다.



세 번째 방법은 RL rotation이 있다. 이 방법은 다음과 같은 상황에서 사용하며 -1, 0, 1이 아닌 balance factor을 -1, 0, 1로 만들어 준다.



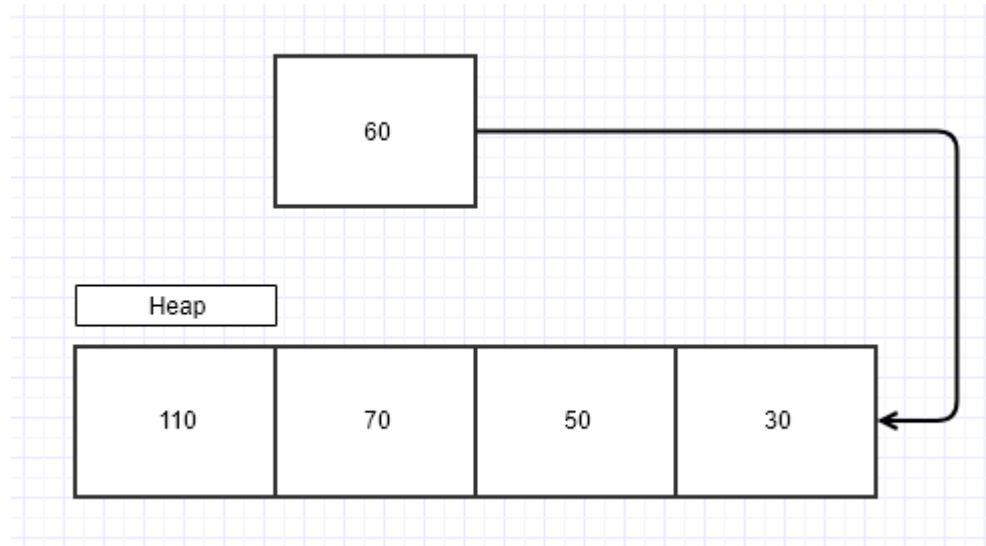
다음은 RL rotation 사용하고 사용한 후의 tree는 다음과 같이 변형된다.



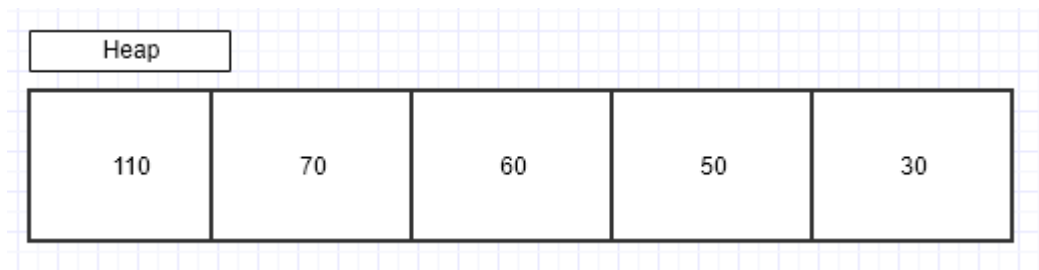
LL rotation을 사용하고 RR rotation을 사용하게되면 다음과 같은 형태의 B+tree가 나오게 된다. RL rotation은 RR rotation을 한 후 재배열된 트리로 다시 LL rotation을 사용한 것과 같다.

### (3) Heap push

Heap에서 어떤 원소를 추가 시켜줄 때에 어떤 원소를 추가 시켜줄지 결정한 후 그것을 맨뒤에 대입하고 heap의 형태로 재정렬시키는 방법을 많이 사용한다. 다음은 방금 설명한 방법을 도식적으로 나타낸 것이다.



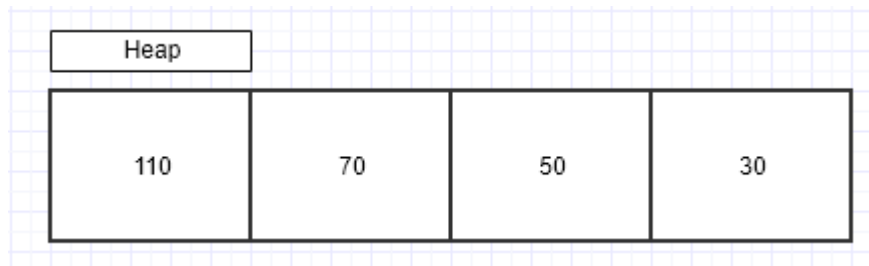
다음은 어떤 한 Heap에 60이라는 수를 넣었을 때, 맨뒤에 대입됨을 도식화하여 나타낸 것이다. 이렇게 맨 뒤에 대입된 원소는 Heap에 들어가게 되고 정렬이 되어야한다.



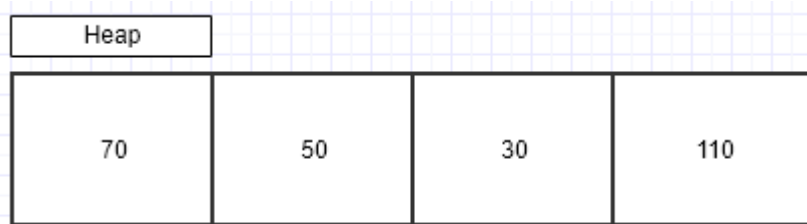
정렬된 heap을 도식화하여 나타낸 것이다. 어떤 원소를 Heap의 맨 뒤에 붙여주는 함수는 `push_back()`이고, `push_heap()`은 조건에 따라 Heap을 정렬시켜주는 함수이다. 이 두 함수를 사용하게되면 Heap push의 기능을 구현할 수 있다.

#### (4) Heap pop

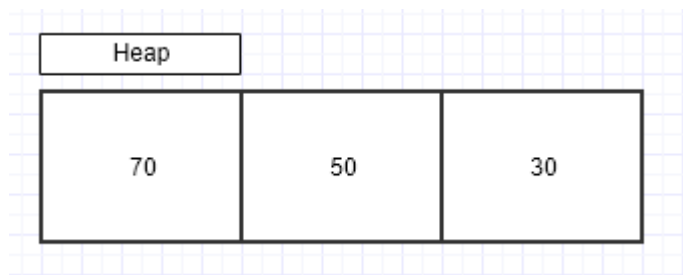
Heap에서 어떤 원소를 삭제 시켜줄 때에 삭제하고 싶은 원소를 찾아준 후 배열의 맨뒤로 보내고 그것을 pop시키는 방법을 많이 사용한다. 이 때, pop되는 원소의 정보를 다른 하나의 포인터 변수에 저장할 시키면 그 정보는 출력이나 다른 작업을 수행하는데에 사용될 수 있다. 다음은 방금 설명했던 방법을 도식적으로 나타낸 것이다.



다음과 같은 Heap이 존재할 때, 삭제할 데이터를 구분해준다. 이때, 110을 삭제하기위해선 이 원소를 벡터의 맨뒤로 보내주어 다음과 같은 그림이 성립한다.



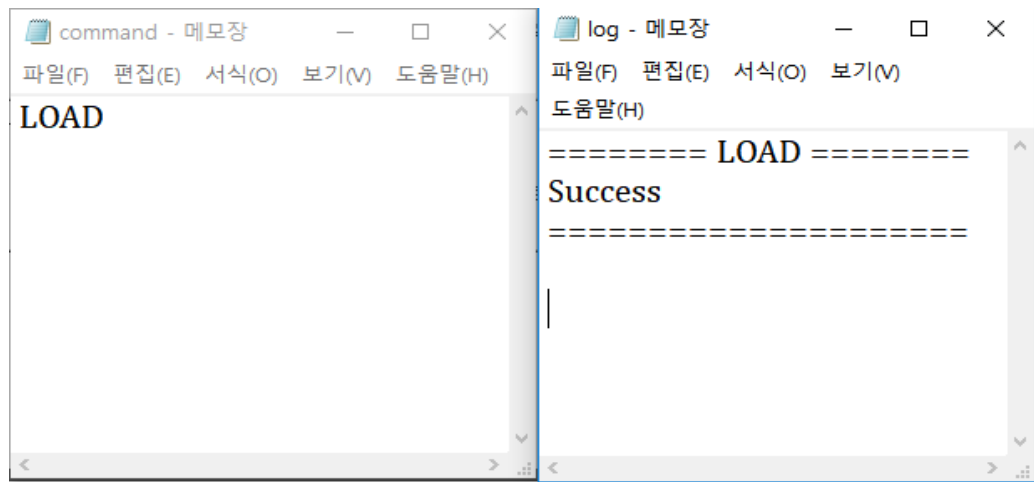
이렇게 구성된 벡터에 맨 마지막 원소를 삭제시켜주는 구현을 기능하게 되면 맨 마지막 원소가 사라져 다음과 같은 형태를 띄게 된다.



조건에 따라 heap의 원소를 벡터의 맨 뒤로 보내주는 함수는 `pop_heap()`이고 Heap의 맨 마지막 원소를 삭제시켜 주는 함수는 `pop_back()`이다. 이러한 두 함수를 사용하게 되면 성공적으로 pop의 기능을 구현할 수 있다.

## 4. Result Screen

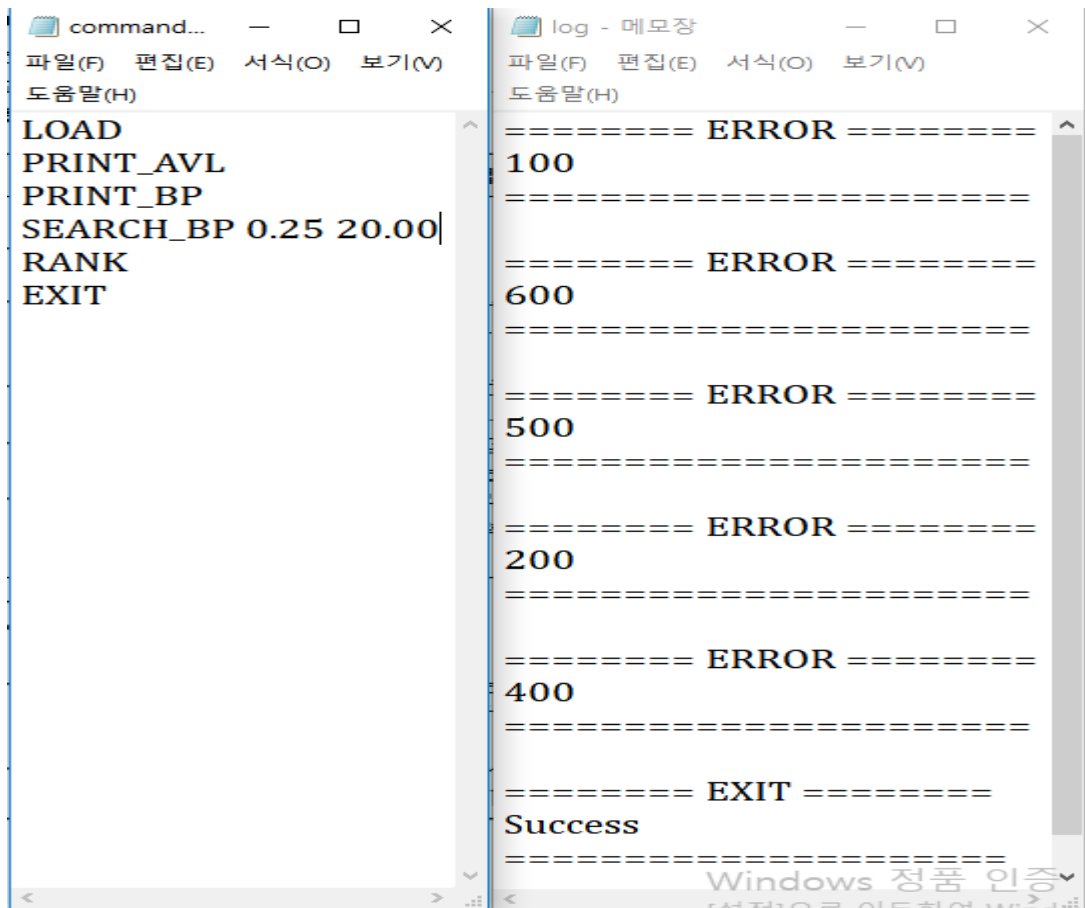
### (1) LOAD



다음은 성공적으로 LOAD가 실행 되었을 때, 출력 파일인 'log.txt'에 입력되는 내용이다.

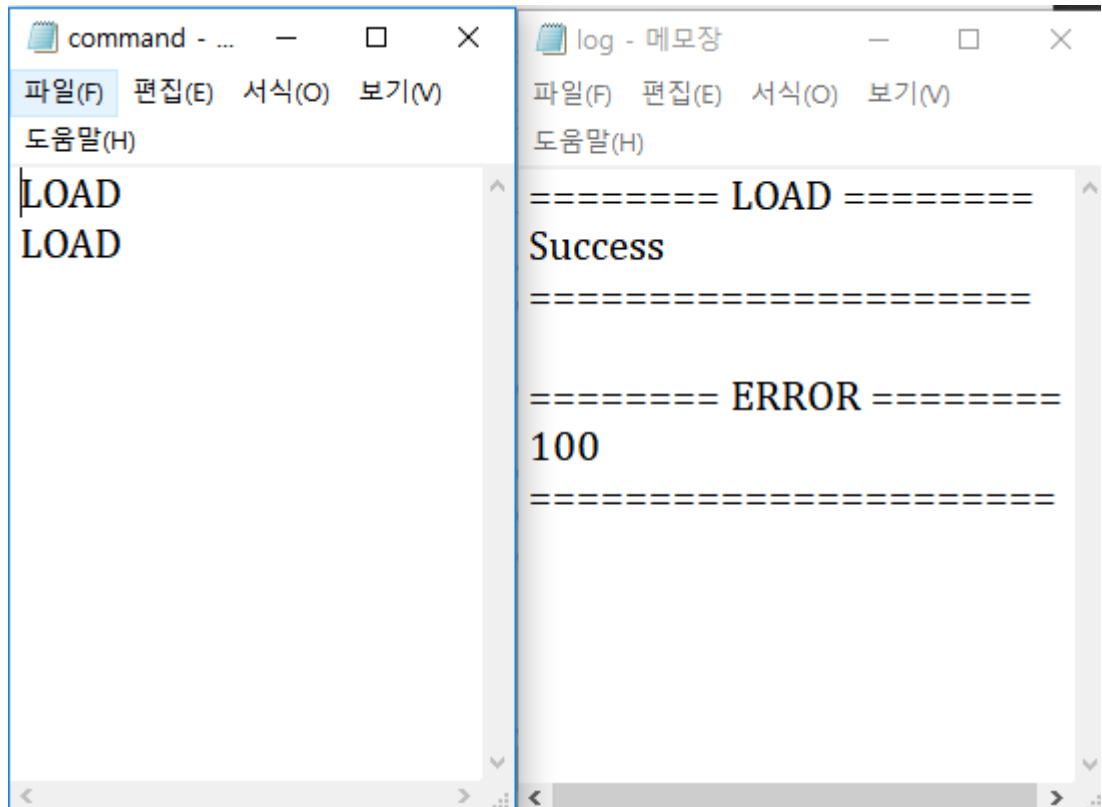
### ※Exception Handling※

#### (1) 텍스트 파일이 존재하지 않는 경우



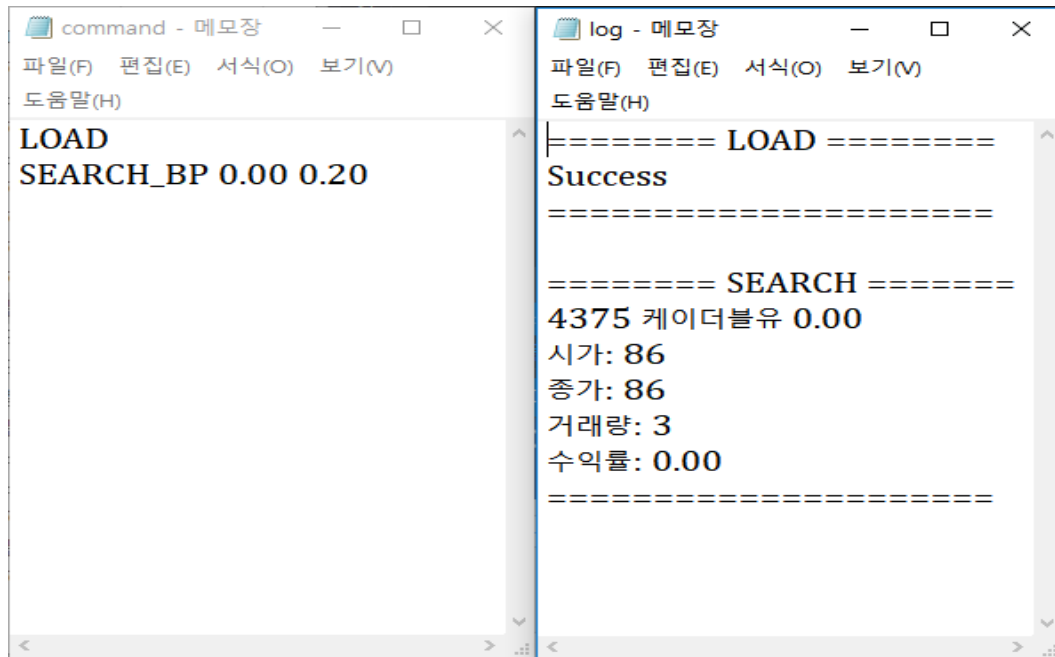
다음은 텍스트 파일("stock\_list.txt")파일이 존재하지 않을 때, 출력 파일인 'log.txt'에 출력되는 내용이다.

(2) 트리가 이미 구성되어 있는 경우



트리가 이미 구성되어 있는 경우, 에러 코드를 출력한다.

(2) SEARCH\_BP



```
command - 메모장
파일(F) 편집(E) 서식(O) 보기(V)
도움말(H)
LOAD
SEARCH_BP 0.00 0.20

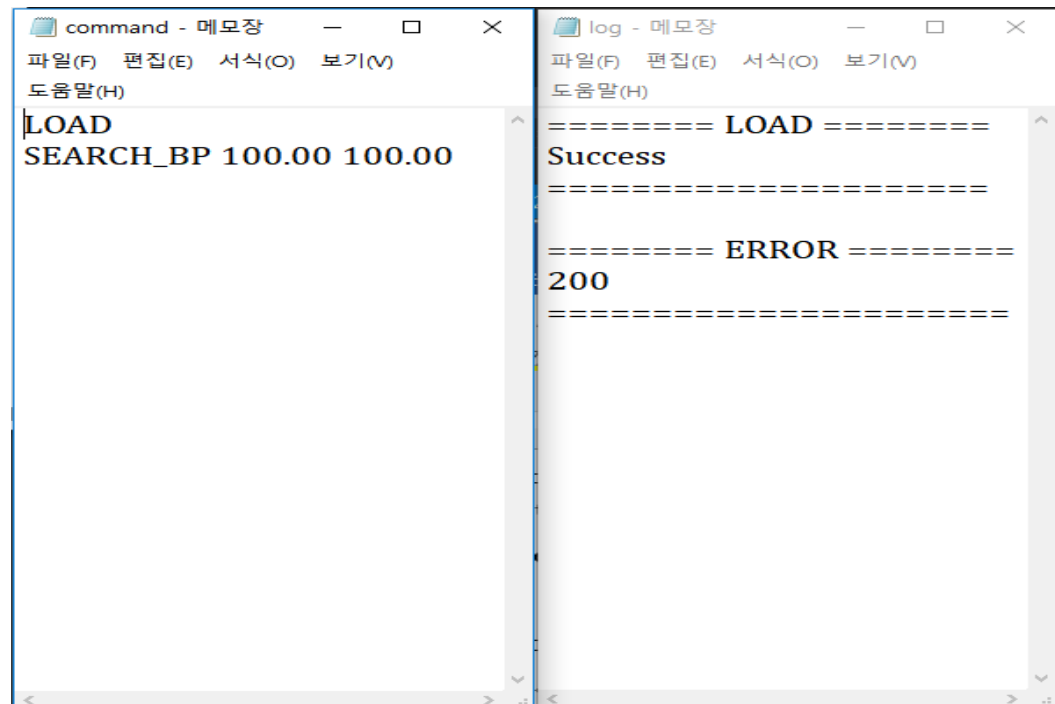
log - 메모장
파일(F) 편집(E) 서식(O) 보기(V)
도움말(H)
===== LOAD =====
Success
=====

===== SEARCH =====
4375 케이더블유 0.00
시가: 86
증가: 86
거래량: 3
수익률: 0.00
=====
```

다음은 SEARCH\_BP 숫자1 숫자2 형식의 명령어를 받게 되었을 때, 출력하는 format이다.

※Exception Handling※

(1) 입력한 조건의 종목이 존재하지 않는 경우



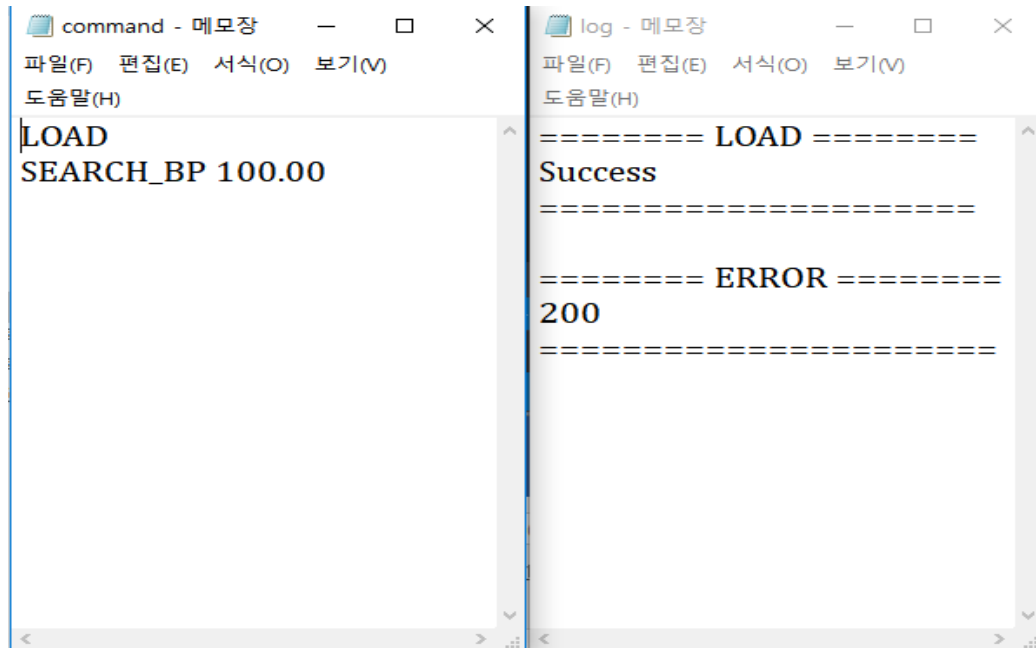
```
command - 메모장
파일(F) 편집(E) 서식(O) 보기(V)
도움말(H)
LOAD
SEARCH_BP 100.00 100.00

log - 메모장
파일(F) 편집(E) 서식(O) 보기(V)
도움말(H)
===== LOAD =====
Success
=====

===== ERROR =====
200
=====
```

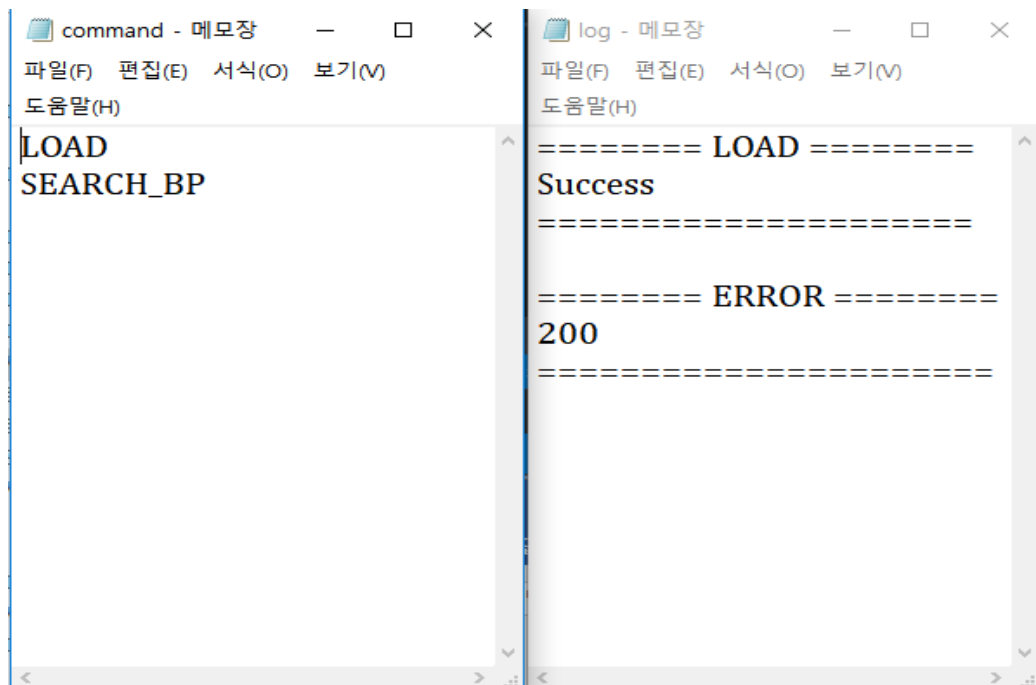
수익률이 100.00인 데이터는 없기 때문에 에러 코드를 출력한다.

(2)-1 시작 범위, 끝 범위 중 하나라도 입력하지 않은 경우



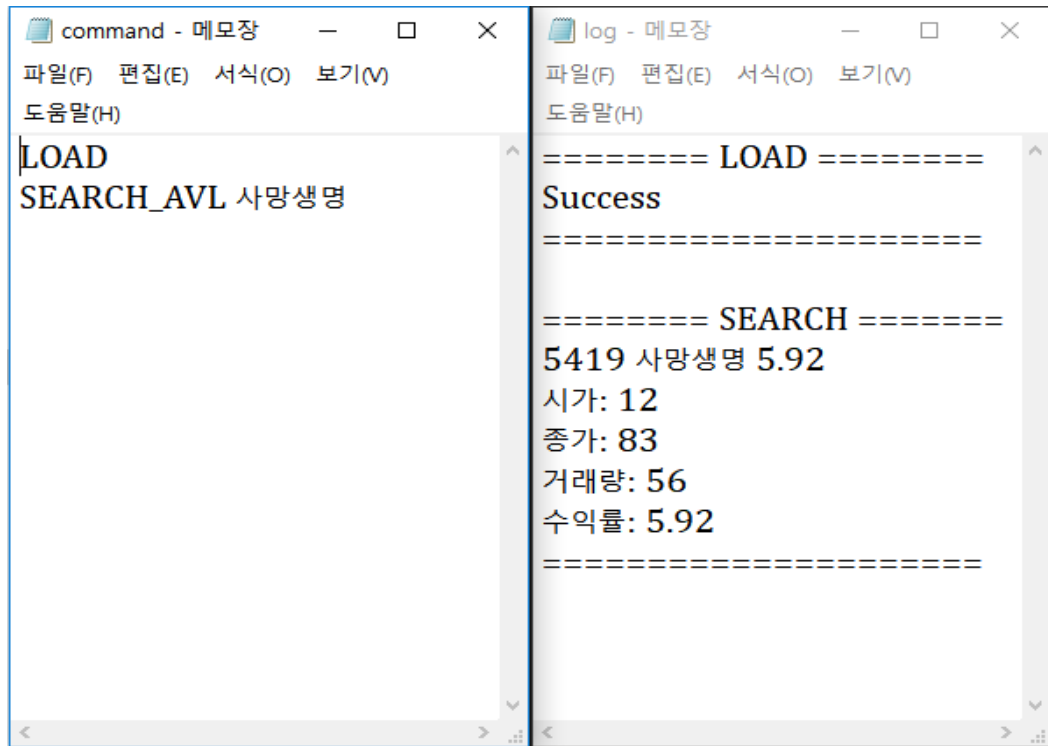
시작 범위나 끝 범위 중 하나라도 입력을 하지 않게 되면 다음과 같은 에러 코드를 출력하게 된다.

(2)-2 시작 범위와 끝 범위 둘다 입력하지 않은 경우



시작 범위와 끝 범위 둘 다 입력이 되지 않으면 다음과 같은 에러 코드를 출력하게 된다.

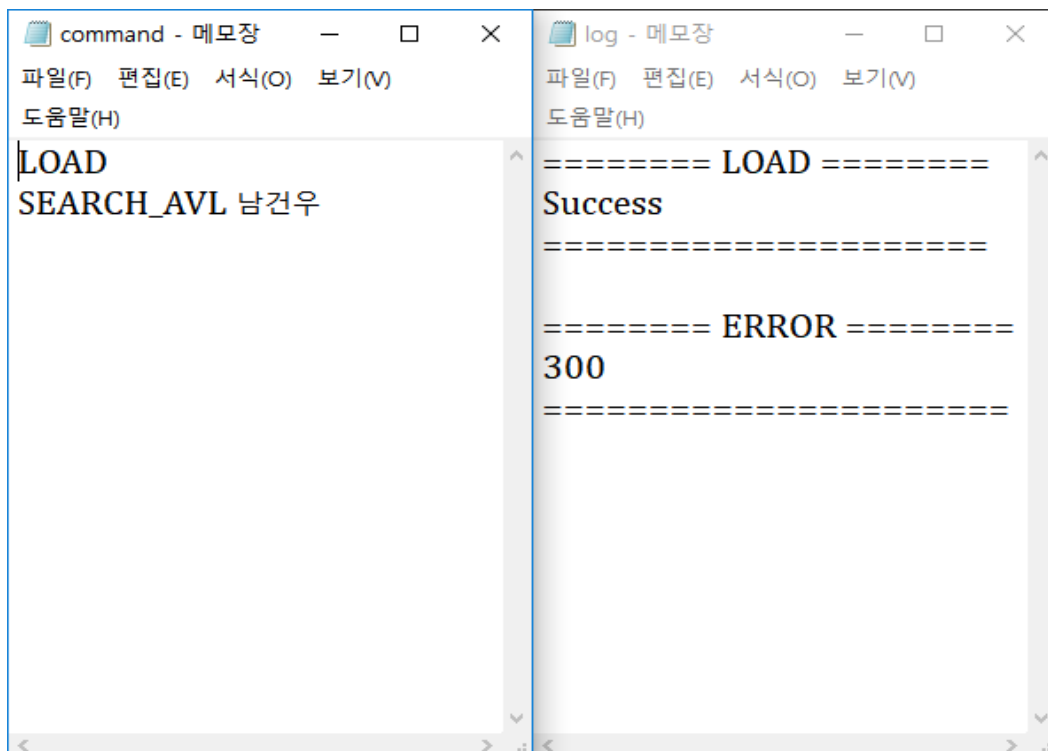
(3) SEARCH\_AVL



'SEARCH\_AVL 이름'의 format을 입력하게 되면 다음과 같은 출력이 일어난다.

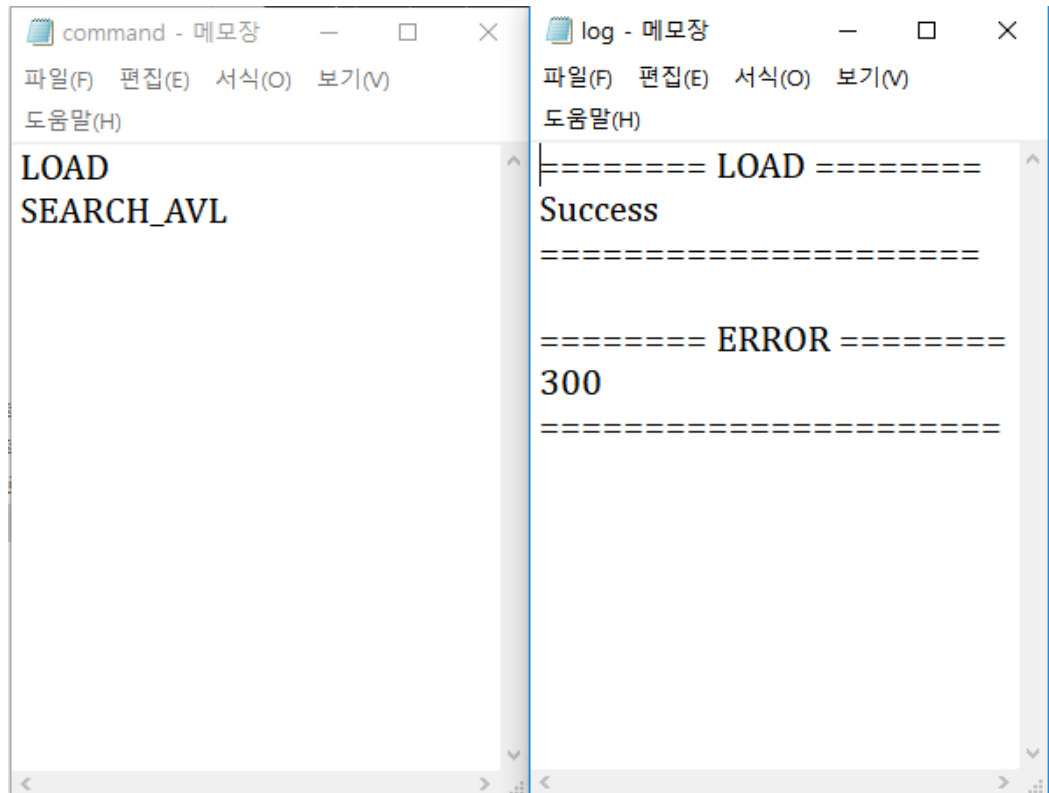
※Exception Handling※

(1) 입력한 이름의 주식이 존재하지 않는 경우

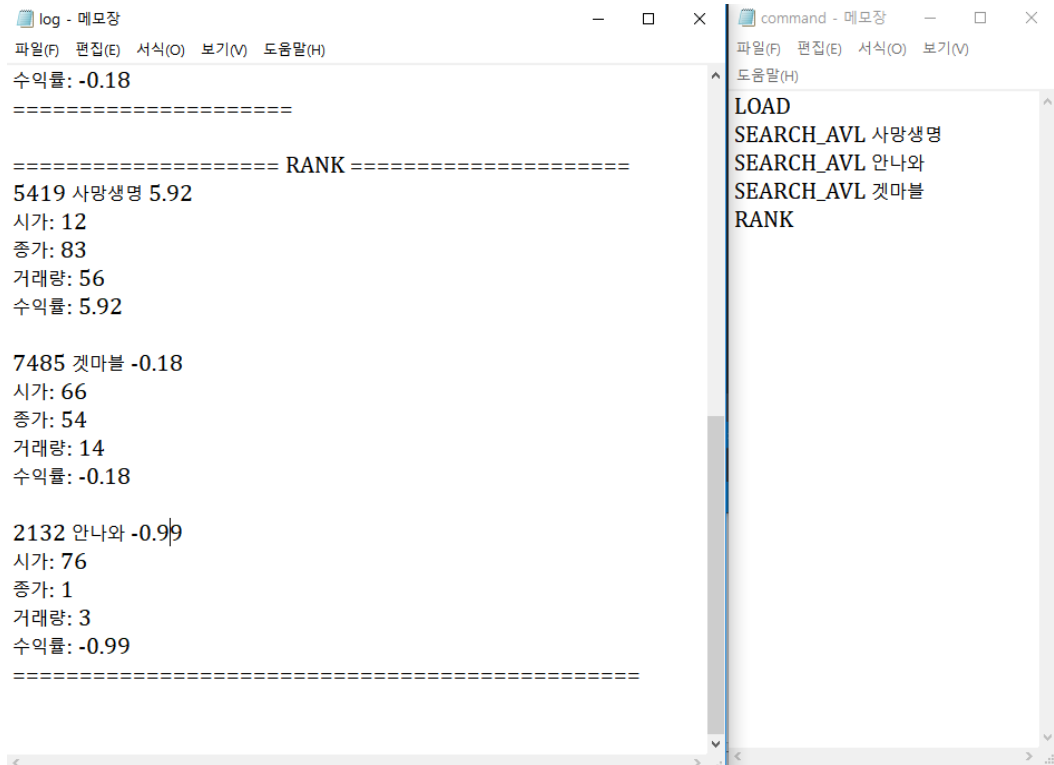




(2) 이름을 입력하지 않은 경우



#### (4) RANK



The screenshot shows two Notepad++ windows. The left window, titled 'log - 메모장', displays the output of the RANK command. It lists three items with their respective statistics: '5419 사망생명 5.92', '7485 갯마블 -0.18', and '2132 안나와 -0.99'. Each item is followed by its '시가' (price), '종가' (closing price), '거래량' (volume), and '수익률' (return). The right window, titled 'command - 메모장', shows the command input: 'LOAD', 'SEARCH\_AVL 사망생명', 'SEARCH\_AVL 안나와', 'SEARCH\_AVL 갯마블', and 'RANK'.

```
log - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
수익률: -0.18
=====

===== RANK =====
5419 사망생명 5.92
시가: 12
종가: 83
거래량: 56
수익률: 5.92

7485 갯마블 -0.18
시가: 66
종가: 54
거래량: 14
수익률: -0.18

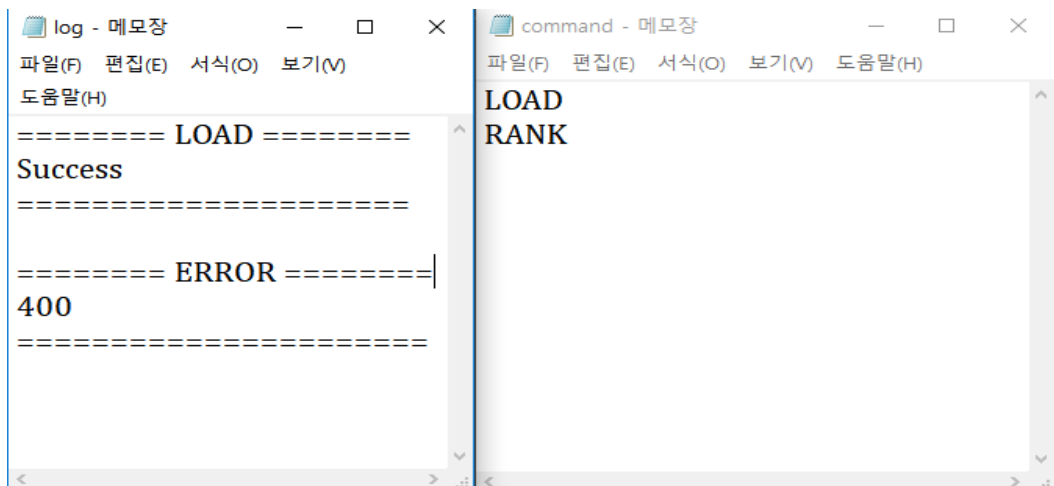
2132 안나와 -0.99
시가: 76
종가: 1
거래량: 3
수익률: -0.99
=====

command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
도움말(H)
LOAD
SEARCH_AVL 사망생명
SEARCH_AVL 안나와
SEARCH_AVL 갯마블
RANK
```

SEARCH\_AVL을 실행한 종목들이 수익률의 내림차순으로 출력한다. 만약 수익률이 같다면 고유번호의 오름차순으로 출력한다.

#### ※Exception Handling※

(1) Heap에 저장되어 있는 정보가 없을 경우



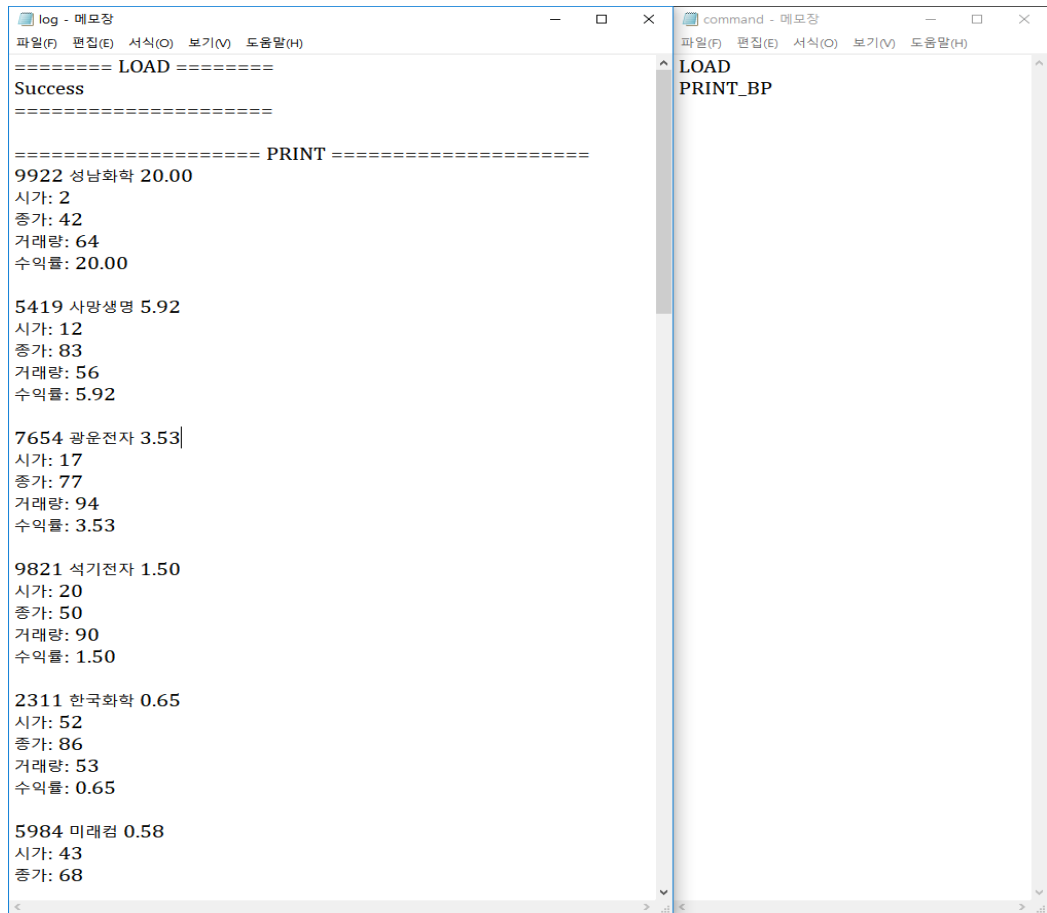
The screenshot shows two Notepad++ windows. The left window, titled 'log - 메모장', displays the output of the RANK command. It shows 'Success' followed by 'ERROR' and the code '400'. The right window, titled 'command - 메모장', shows the command input: 'LOAD' and 'RANK'.

```
log - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
도움말(H)
===== LOAD =====
Success
=====

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

command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
LOAD
RANK
```

## (5) PRINT\_BP



```
log - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
===== LOAD =====
Success
=====
===== PRINT =====
9922 성남화학 20.00
시가: 2
증가: 42
거래량: 64
수익률: 20.00

5419 사랑생명 5.92
시가: 12
증가: 83
거래량: 56
수익률: 5.92

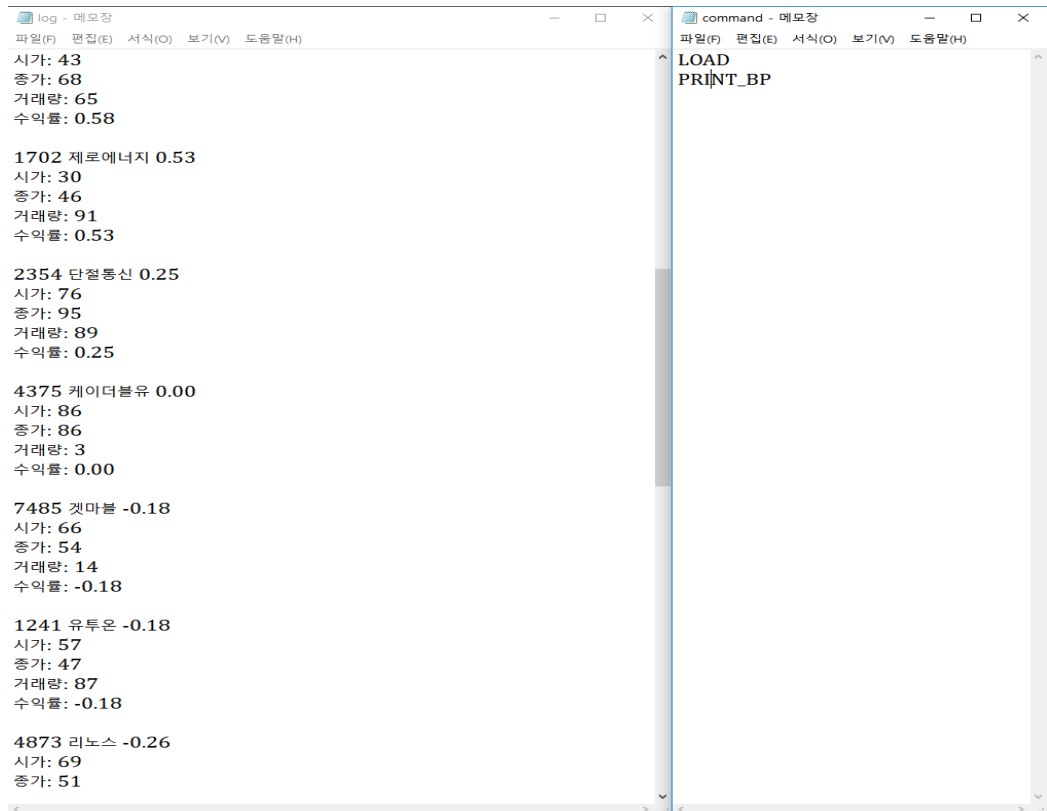
7654 광운전자 3.53
시가: 17
증가: 77
거래량: 94
수익률: 3.53

9821 석기전자 1.50
시가: 20
증가: 50
거래량: 90
수익률: 1.50

2311 한국화학 0.65
시가: 52
증가: 86
거래량: 53
수익률: 0.65

5984 미래컴 0.58
시가: 43
증가: 68

command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
LOAD
PRINT_BP
```



```
log - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
시가: 43
증가: 68
거래량: 65
수익률: 0.58

1702 제로에너지 0.53
시가: 30
증가: 46
거래량: 91
수익률: 0.53

2354 단철통신 0.25
시가: 76
증가: 95
거래량: 89
수익률: 0.25

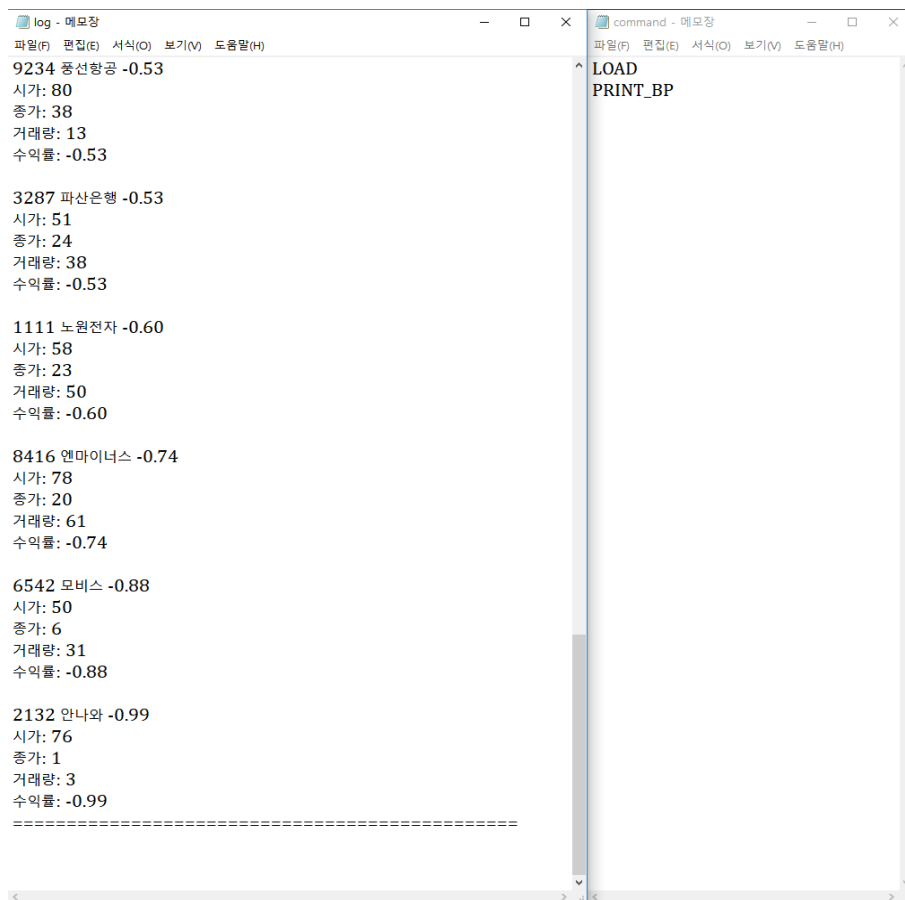
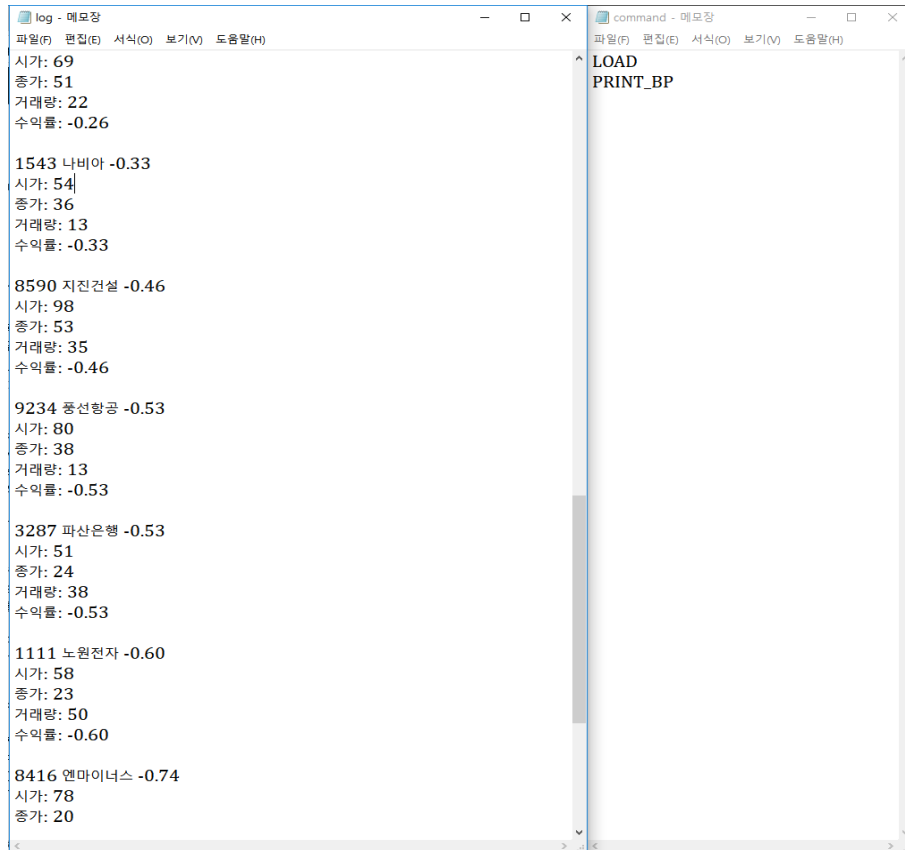
4375 케이더블유 0.00
시가: 86
증가: 86
거래량: 3
수익률: 0.00

7485 겐마블 -0.18
시가: 66
증가: 54
거래량: 14
수익률: -0.18

1241 유투온 -0.18
시가: 57
증가: 47
거래량: 87
수익률: -0.18

4873 리노스 -0.26
시가: 69
증가: 51

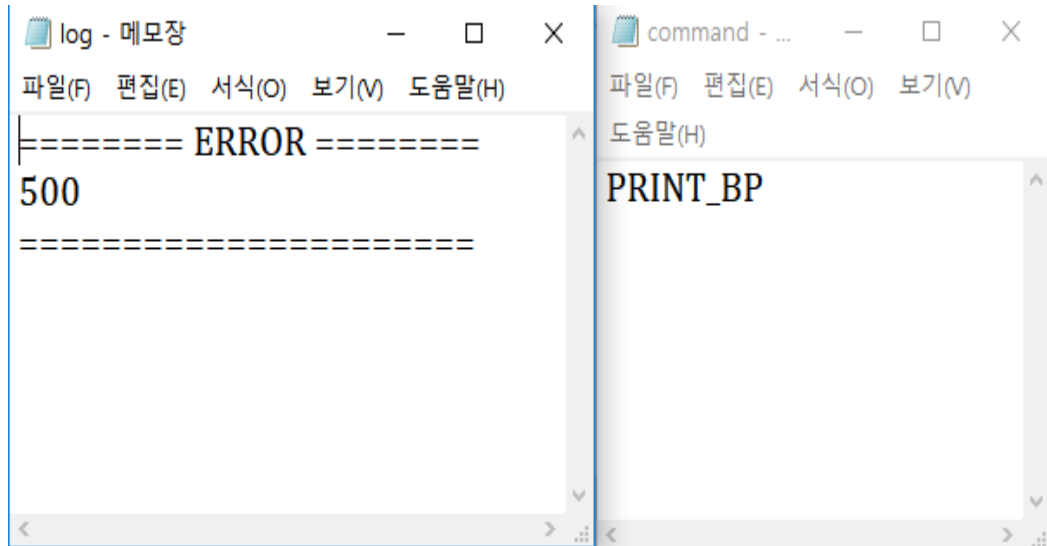
command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
LOAD
PRINT_BP
```



수익률을 기준 내림차순으로 출력 파일에 출력된다. 수익률이 같다면 고유번호 기준 내림차순으로 출력한다.

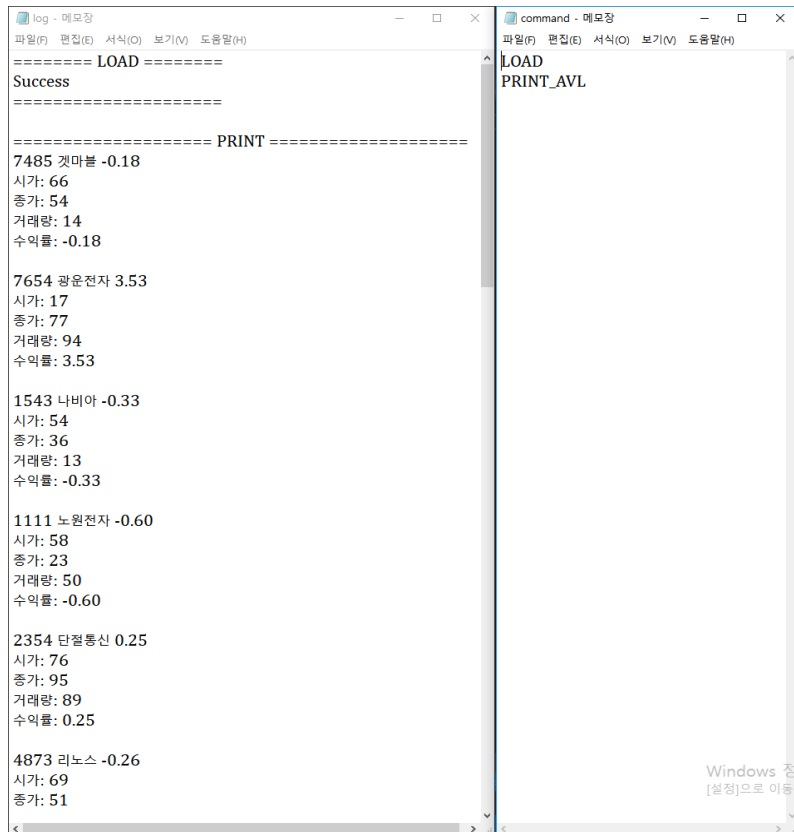
### ※Exception Handling※

(1) B+-tree에 저장되어 있는 정보가 없을 경우



LOAD를 수행하지 않았으므로 "stock\_list.txt"에 있는 데이터들이 자료구조 안에 저장되어 있지 않다. 그럼으로 B+-tree에 저장되어 있는 정보가 없다.

## (6) PRINT\_AVL



The screenshot shows two windows. The left window is titled 'log - 메모장' and contains the following text:

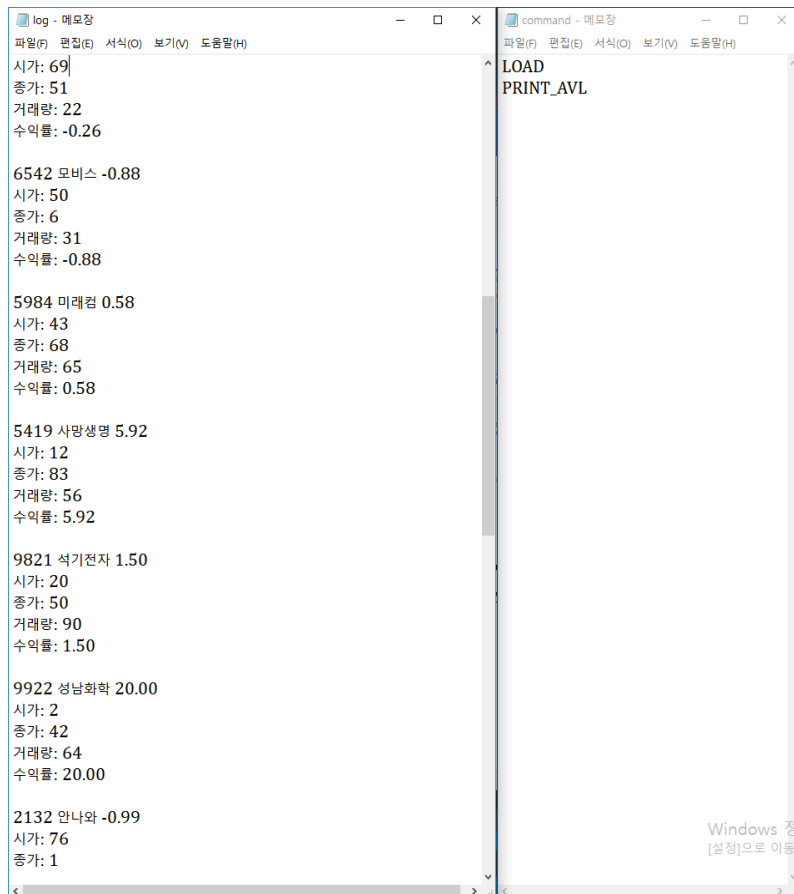
```
===== LOAD =====  
Success  
=====
```

The right window is titled 'command - 메모장' and contains the following text:

```
LOAD  
PRINT_AVL
```

The log file window also displays the output of the PRINT command, showing the following data:

```
===== PRINT =====  
7485 컷마블 -0.18  
시가: 66  
종가: 54  
거래량: 14  
수익률: -0.18  
  
7654 광운전자 3.53  
시가: 17  
종가: 77  
거래량: 94  
수익률: 3.53  
  
1543 나비아 -0.33  
시가: 54  
종가: 36  
거래량: 13  
수익률: -0.33  
  
1111 노원전자 -0.60  
시가: 58  
종가: 23  
거래량: 50  
수익률: -0.60  
  
2354 단절통신 0.25  
시가: 76  
종가: 95  
거래량: 89  
수익률: 0.25  
  
4873 리노스 -0.26  
시가: 69  
종가: 51
```

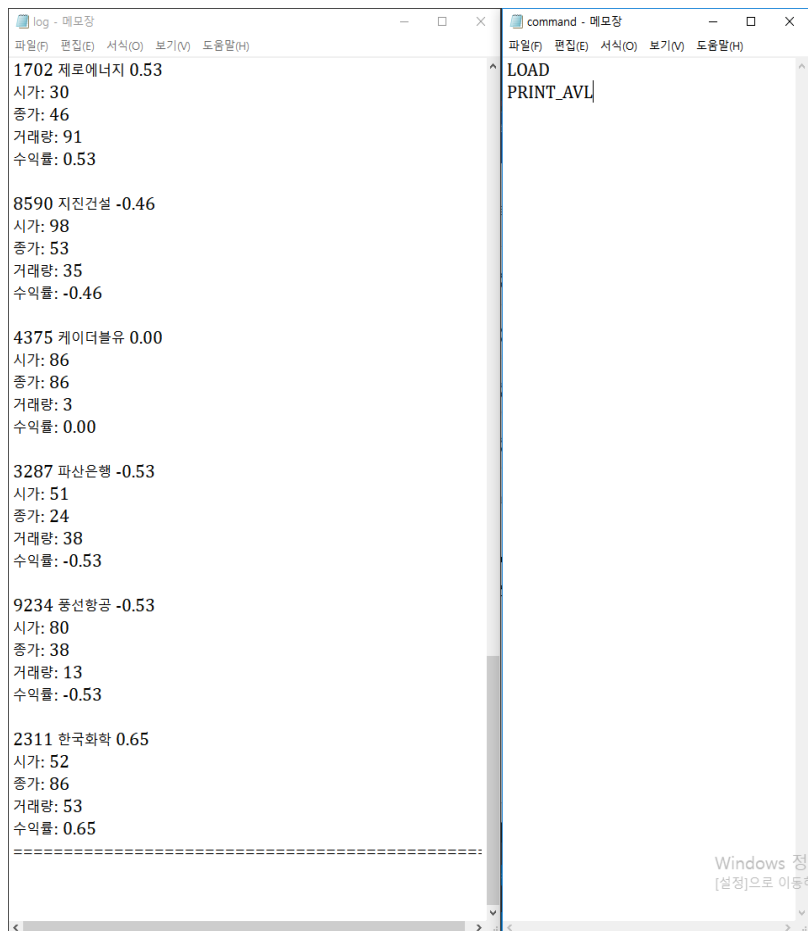
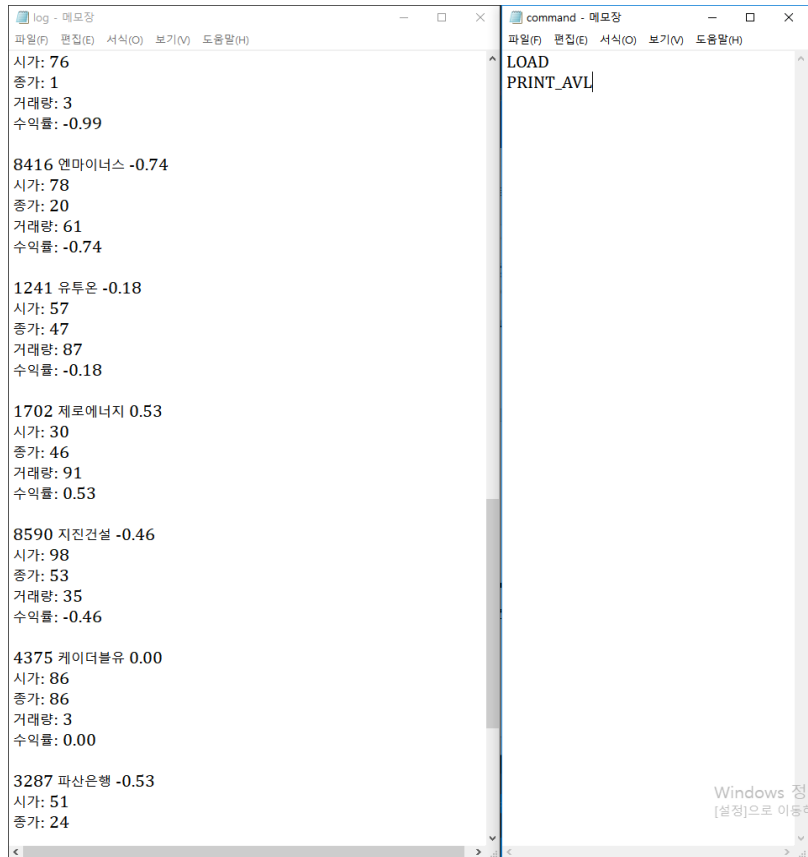


The screenshot shows two windows. The left window is titled 'log - 메모장' and contains the following text:

```
시가: 69  
종가: 51  
거래량: 22  
수익률: -0.26  
  
6542 모비스 -0.88  
시가: 50  
종가: 6  
거래량: 31  
수익률: -0.88  
  
5984 미래컴 0.58  
시가: 43  
종가: 68  
거래량: 65  
수익률: 0.58  
  
5419 사랑생명 5.92  
시가: 12  
종가: 83  
거래량: 56  
수익률: 5.92  
  
9821 석기전자 1.50  
시가: 20  
종가: 50  
거래량: 90  
수익률: 1.50  
  
9922 성남화학 20.00  
시가: 2  
종가: 42  
거래량: 64  
수익률: 20.00  
  
2132 안나와 -0.99  
시가: 76  
종가: 1
```

The right window is titled 'command - 메모장' and contains the following text:

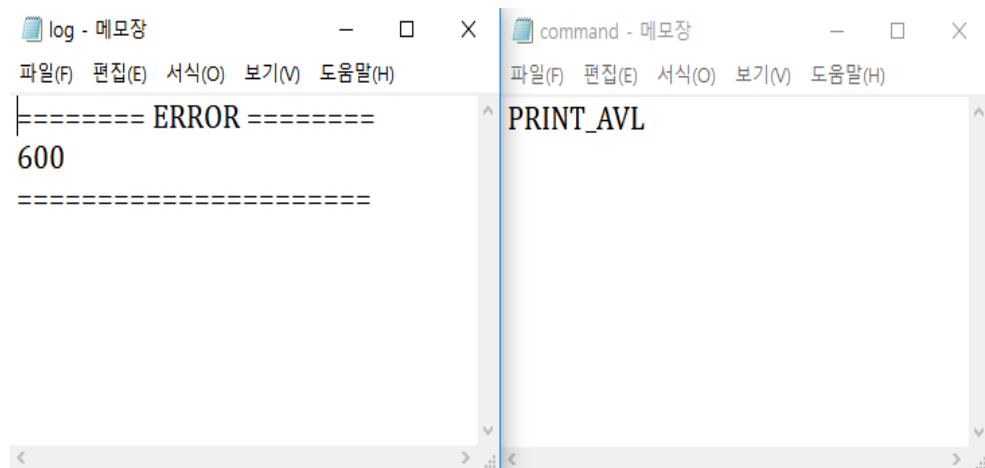
```
LOAD  
PRINT_AVL
```



종목 기준 오름차순으로 출력 파일에 출력된다.

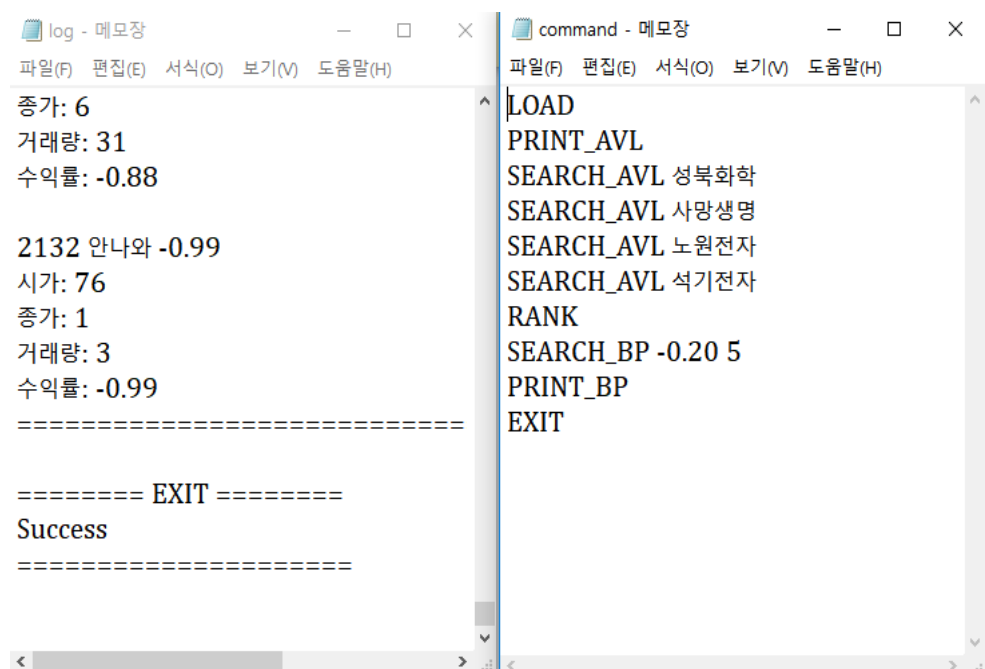
### ※Exception Handling※

(1) AVL tree에 저장되어 있는 정보가 없을 경우



LOAD를 수행하지 않았으므로 "stock\_list.txt"에 있는 데이터들이 자료구조 안에 저장되어 있지 않다. 그럼으로 AVL tree에 저장되어 있는 정보가 없다.

(7) EXIT



EXIT가 성공적으로 수행이 된다면 Success code가 나온다.



## 5. Consideration

프로젝트를 진행하면서 수 많은 역경과 고난이 존재하였다. 이번 프로젝트는 두 개의 자료 구조인 B+-tree와 AVL tree를 구현하여 데이터를 저장하는 것이었는데 이 과정 속에서는 수 많은 어려움이 있었다. AVL tree의 insert함수는 중간 고사를 공부하면서 이미 숙지했던 내용이고 코드의 흐름까지 외워 구현하였기 때문에 크게 어려운 점이 없었다. 하지만 heap과 vector의 사용으로 인해 여러가지 혼란이 있었다. vector라는 것의 개념과 사용 방법, 관련 함수를 아예 몰랐기 때문에 어떻게 구현해야 할지 막막했지만 skeleton code를 보면서 공부하고 블로그나 구글링을 통해 관련된 많은 코드를 보면서 어떻게 쓰이고 또 어떤 basic function이 있는지 또한 할 수 있었다. 이를 통해, skeleton code를 이해할 수 있었고 주석을 달을 수 있었다. 이러한 개념을 바탕으로 B+-tree를 구현하였는데 이 알고리즘을 통한 코드의 구현이 매우 어려웠다. 개념적으로 학습하였을 때, 데이터 노드인 경우 두 가지로 쪼개고 오른쪽에 있는 노드의 첫 번째 요소를 인덱스 노드로 올려서 오른쪽 노드와 연결시켜주면 되었다. 또한 인덱스 노드인 경우 3가지로 쪼개고 split이 일어나는 노드를 위로 올려 새로운 부모 인덱스 노드로 만들고 오른쪽 노드와 연결하는 것이라고 생각했다. 간단한 개념을 이해하는 것은 쉬웠지만 코드로 구현하는 것은 매우 어려웠다. 먼저 상속이라는 개념을 통해 부모와 가장 왼쪽에 있는 자식의 개념을 받아 노드를 만들고 upcasting을 쓰는 동시에 가상 함수를 구현함으로써 상속받은 노드의 함수를 실행시킬 수 있도록 만들어야 했다. 이런 개념들은 프로젝트를 하기 전에 모호하게 알고 있었기 때문에 이번 프로젝트를 통해 개념들을 정확히 잡을 수 있었다. 게다가 처음 접해보는 vector와 map의 사용을 통해 간단한 구조를 가진 B+-tree를 구현하는 것이 너무나 힘들었다. 개념은 알고 있지만 코드로의 구현이 매우 어려웠다. Map은 vector와 비슷하지만 처음 접해봤기 때문에 차이점을 알지 못했다. 하지만 구글링이나 인터넷 검색을 통해 자동적으로 sorting이 된다는 것을 알게 되었고 이를 통해 하나의 노드 안에 있는 map의 각 원소들이 자동적으로 sorting되는 것을 디버깅을 통해 확인할 수 있었다. 처음엔 sorting을 하는 방법을 코드로 구현해야 하는 줄 알았던 나에게 map은 너무나 신선하게 다가왔고 vector와는 다르게 너무나 구현이 편했다. Vector의 경우엔 sort라는 함수를 사용하여 parameter로 처음과 끝(end)를 넣어야 했기 때문에 다소 불편했다. 이렇게 생소했던 개념들을 배울 수 있었고 또한 1학년 때 배웠던 private과 public에 대한 개념을 짚고 넘어갈 수 있었다. private에 선언된 identifier들은 포인터 형을 통해 접근을 하지 못한다는 것을 직접적으로 경험할 수 있었다. AVL tree의 root에 접근을 하기 위해선 public으로 선언하면 되지만 root의 값을 실수나 의도치 않게 수정을 할 수 있기 때문에 위험하다는 것을 깨달았다. 이를 통해, 해당 헤더 파일을 통해 root를 반환하는 함수인 getRoot함수를 정의 했고 root의 반환형을 따라 만들었다. 이를 통해, root의 값을 전달받아 NULL 값이 될 때 예외 처리를 할 수 있었다. 또한 stack 사용을 통해 B+-tree의 print()함수를 구현함으로써 헤더파일 stack의 library function을 다시 한번 되새길 수 있었고 앞으로도 stack을 자주 사용할 수 있을 것 같았다. Stack은 queue와는 다르게 fifo가 아닌 lifo를 직접적으로 경험할 수 있었고 나중에 들어온 것을 먼저 출력할 때 사용할 때 유용하다는 것을 깨닫게 되었다.

이번 프로젝트를 통해 정말 많은 것을 배우고 느낀 것 같다. 과제가 어려웠던 것만큼 많이 힘들고 포기하고 싶었던 순간들도 있었지만 디버깅을 통해 다양한 실패를 경험하고 단 한번의 성공을 경험함으로써 뿌듯함 또한 느낄 수 있었다. 어렵고 힘든 만큼 많은 것을 배운 것 같아 많이 뿌듯했고 보람 있었다.