Data Structure Project

Project #2

담당교수 : 이기훈 교수님

제출일 : 2018. 11. 11.

학과 : 컴퓨터정보공학부

학번 : 2017202029

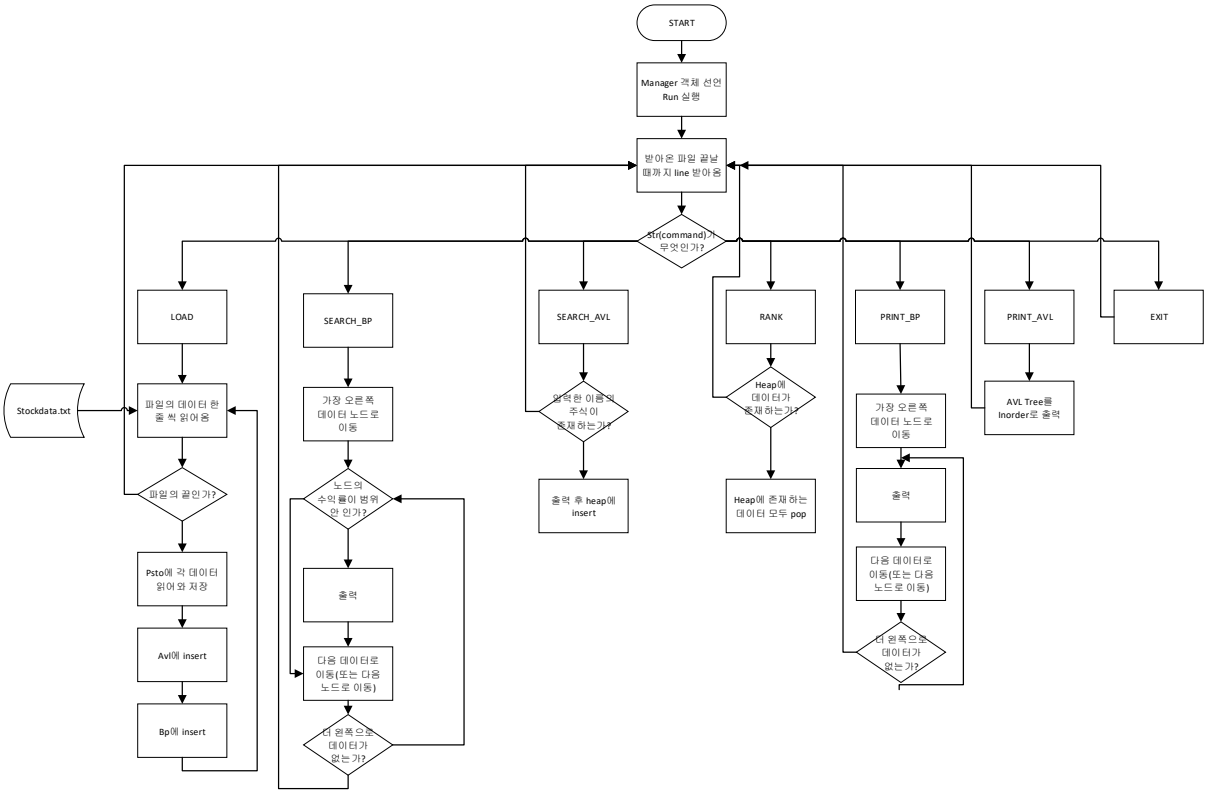
이름 : 전효희

1. Introduction

본 프로젝트는 avl tree와 b+ tree, heap을 사용하여 주식 정보에 대한 데이터를 저장 및 처리하는 프로그램을 제작하는 프로젝트이다. 프로그램을 활용하여 주식 정보를 입력 받아, 주식 전체 리스트의 이름 순, 수익률순으로 정렬된 데이터를 얻을 수 있다.

프로그램은 특정 이름 이나 특정 범위를 가진 수익률을 대상으로 하는 검색이 가능하며, 각각의 데이터가 존재하지 않거나 기능 수행에 문제가 있는 경우 해당 기능에 대한 오류 코드를 출력할 수 있는 예외처리를 진행한다.

2. Flow Chart



- 프로그램이 시작하면 manager 객체를 하나 생성하고 "command.txt"를 인자로 넣어준다. manager의 run을 시작한다.
- 받아온 파일이 끝날 때까지 한 줄 씩 읽어 str변수에 저장한다. str의 값에 따라 기능으로 이동한다.
- str가 LOAD일 경우 "stock_list.txt" 파일을 불러와 아직 데이터가 없을 경우 해당하는 자

료구조에 insert한다.

- str가 SEARCH_BP일 경우 범위를 입력 받아 가장 오른쪽의 데이터 노드로 이동한 후 범위 안에 있으면 출력 후 앞 데이터로, 아니면 앞 데이터로 이동한다.
- str가 SEARCH_AVL일 경우 주식 이름이 입력 받아 존재하는 주식정보라면 정보를 출력하고 heap에 인서트하고, 존재하지 않으면 에러를 출력한다.
- str가 RANK일 경우 힙에 데이터가 있으면 주식정보를 출력하고 없으면 에러를 출력한다.
- Str가 PRINT_BP 일 경우 가장 오른쪽의 데이터 노드로 이동한 후 앞 데이터가 없을 때까지 앞 데이터로 이동하며 출력한다.
- Str가 PRINT_AVL 일 경우 TREE의 inorder 순서로 출력한다.
- 각각의 기능에 실패하는 경우 해당하는 에러코드를 출력한다.

3. Algorithm

- LOAD

  * B+ tree insert

    루트부터 시작하여 데이터를 받아 수익률을 기준으로 수익률이 들어갈 수 있는 데이터 노드로 이동한다. 해당하는 곳에 데이터를 새로 insert 한다. 만약 insert 후 노드가 넘쳤으면 데이터 노드를 split 한다.

    Index로 올릴 데이터를 찾고, 동시에 올릴 데이터를 포함하여 올릴 노드 뒤에 존재하는 데이터로 새 노드를 만든다. 올릴 데이터를 index node에 insert 한다. 만약 insert 후 index 노드도 넘쳤으면 index node를 split한다.

    Index 노드를 split 하기 위해 올릴 데이터를 찾고, 올릴 데이터 뒤에 존재하는 데이터로 새 index 노드를 만든다. 현재 노드가 루트일 경우 새 index 노드를 만들어 위로 올리고, 루트가 아닐 경우 부모 index node에 insert한다. 만약 부모 index node에서도 데이터가 넘치면 다시 부모 index node를 split 한다.

    ex) 자신이 들어갈 위치 찾음 -> insert -> insert된 노드 exceed -> split data

    -> index node insert -> insert된 노드 exceed -> split index

  * AVL tree insert

    루트부터 들어온 데이터와 사전순으로 비교하여 사전 순으로 앞에 오면 left로, 뒤에 오면 right로 이동하여 insert할 자리를 찾고 insert 한다.

새 노드가 들어온 위치를 기준으로 하여 새 balance factor 를 설정한다. 만약 A노드의 변한 BP가 0이거나 -1, 1이면 rotate하지 않고, 그 외에 rotate하여 BP를 바꿔준다.

Ex) insert 된 후 A노드의 bp = 2 -> insert 된 노드의 위치가 왼쪽-> a의 자식 b 보다 왼쪽 -> rotate Left-Left

- SEARCH_BP

  루트에서 시작하여 가장 오른쪽 데이터 노드로 이동한다. 이후 prev 노드로 이동하면서 다음 prev 노드가 없을 때까지 이동한다. 그러면서 해당 노드의 가장 큰 데이터부터 작은 데이터까지 이동하면서 데이터의 수익률이 입력한 범위 내에 있으면 데이터를 출력하고, 없으면 다음으로 이동한다.

- SEARCH_AVL

  루트부터 시작하여 입력된 데이터가 현재 노드보다 작으면 왼쪽, 크면 오른쪽 자식으로 이동하면서 탐색한다. 만약 찾지 못할 경우, return 한다.

  데이터를 찾은 경우 주식정보를 출력하고 데이터를 heap에 insert 한다. Insert한 후 heap을 수익률 내림차순으로 정렬한다.

  Ex) 루트부터 탐색한 데이터가 존재 -> heap에 삽입 -> heap을 정렬

- RANK

  Heap에 존재하는 데이터를 하나씩 꺼내 데이터가 더 존재하지 않을 때까지 출력한다.

  Ex) heap에 데이터가 존재함 -> heap의 가장 상위 데이터 출력 ->heap의 해당데이터 pop

- PRINT_AVL

  Avl tree를 inorder로 출력하여 사전순으로 출력한다.

  루트부터 시작하고, 루트를 stack에 넣고 왼쪽자식으로 이동하고 stack에 넣는다. 이후 오른쪽 자식을 모두 stack에 넣고 다시 돌면서 pop한다.

- PRINT_BP

  루트에서 시작하여 가장 오른쪽 데이터 노드로 이동한다. 이후 prev 노드로 이동하면서 다음 prev 노드가 없을 때까지 이동한다. 그러면서 해당 노드의 가장 큰 데이터부터 작은 데이터까지 이동하면서 데이터를 출력한다. 같은 벡터에 들어있는 경우 스택에 저장하고

pop하여 고유번호의 내림차순으로 출력한다.

4. Result Screen

- Command.txt

```
PRINT_AVL
PRINT_BP
LOAD
PRINT_AVL
PRINT_BP
SEARCH_BP 0.25 20.00
SEARCH_BP -0.5
SEARCH_BP -2 -3
SEARCH_AVL 광운대학교
SEARCH_AVL 단절통신
SEARCH_AVL 사망생명
SEARCH_AVL 모비스
LOAD
RANK
EXIT
```

-

```
======== ERROR ========
600
========================

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

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

구성되어 있는 AVL tree 없음 : 에러(600) 출력

구성되어 있는 BP tree 없음 : 에러(500) 출력

LOAD 성공 : Success 출력

- Print AVL (중략)

```
==================== PRINT ====================
7485 겟마블 -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
```

- 주식 이름 이 사전 순으로 출력된 모습

- PRINT_BP(중략)

```
==================== PRINT ====================
9922 성남화학 20.00
시가: 2
종가: 42
거래량: 64
수익률: 20.00

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

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

9821 석기전자 1.50
시가: 20
종가: 50
거래량: 90
수익률: 1.50
```

- 수익률의 내림차순으로 출력된 모습

- PRINT_BP

```
7485 겟마블 -0.18
시가: 66
종가: 54
거래량: 14
수익률: -0.18

1241 유투온 -0.18
시가: 57
종가: 47
거래량: 87
수익률: -0.18
```

- PRINT_BP에서 수익률이 같을 경우 고유번호의 내림차순으로 정렬

```
======= SEARCH =======
9922 성남화학 20.00
시가: 2
종가: 42
거래량: 64
수익률: 20.00

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

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

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

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

5984 미래콤 0.58
시가: 43
종가: 68
거래량: 65
수익률: 0.58

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

2354 단절통신 0.25
시가: 76
종가: 95
거래량: 89
수익률: 0.25

======================
```

- 0.25~20.00 범위의 모든 주식 정보가 출력된다.

  수익률을 기준으로 내림차순으로 출력된다.

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

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

SEARCH_BP -0.5 : 최대 범위가 없으므로 에러(200) 출력

SEARCH_BP -2 -3 : 범위 내에 존재하는 데이터가 없으므로 에러(200) 출력

SEARCH_AVL 광운대학교: 존재하지 않는 주식이름 이므로 에러(300) 출력

```
======= SEARCH =======
2354 단절통신 0.25
시가: 76
종가: 95
거래량: 89
수익률: 0.25

=====================

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

=====================

======= SEARCH =======
6542 모비스 -0.88
시가: 50
종가: 6
거래량: 31
수익률: -0.88

=====================
```

-

SEARCH_AVL 단절통신 : 존재하는 정보이므로 주식정보 출력

SEARCH_AVL 사망생명 : 존재하는 정보이므로 주식정보 출력

SEARCH_AVL 모비스 : 존재하는 정보이므로 주식정보 출력

```
======== ERROR ========
100
=======================

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


2354 단절통신 0.25
시가: 76
종가: 95
거래량: 89
수익률: 0.25


6542 모비스 -0.88
시가: 50
종가: 6
거래량: 31
수익률: -0.88


==============================================


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

-

LOAD: 이미 트리가 만들어졌으므로 에러(100) 출력

RANK: SEARCH_AVL했던 내용을 수익률 내림차순으로 출력

EXIT : Success 출력

5. Consideration

처음 AVL tree를 구현함에 있어서 출력 결과, 3개의 데이터를 제외하고 더 이상 데이터가 insert 되지 않는 것을 볼 수 있었다. 그 이유를 탐색한 결과, rotate를 시작하면 tree의 연결이 끊어지는 것을 볼 수 있었다. Rotate를 시작하면 a노드 밑으로 노드를 움직이고 그에 따라 가장 부모에 있는 노드가 바뀌게 된다. 따라서 a노드의 부모와 rotate를 진행한 것 중 가장 부모 노드를 연결해야 트리가 연결된다. 이를 수정하고 실행 시키자 여러 개의 데이터 또한 트리에 바르게 insert되는 것을 확인 할 수 있었다.

SEARCH_BP를 구현함에 있어서 앞 뒤 범위 값을 나눠 각각이 해당하는 노드로 들어가 이동하여 두 노드가 같아지면 종료하는 방식을 처음 생각했다. 하지만 이 경우 그 구현이 복잡하지만 범위안에 데이터가 없는 경우를 처리해주기가 매우 곤란하다는 점이 있었다. 이런 점을 해결하는 방식은 매우 간단하게도, PRINT_BP 와 같은 방식으로 큰 데이터에서 작은 데이터로 이동하면서 범위안에 있는 경우에만 출력하는 방식으로 수정하여 데이터가 없는 경우를 처리할 수 있었다.

B+ tree의 insert에 있어서 split 과정이 매우 복잡했다. PRINT_BP등을 위해서 자식으로 이동하는 포인터와 부모로 이동하는 포인터가 모두 설정되어야 하는데, 몇몇 경우 이 설정이 없어 PRINT가 진행되지 않는 점이 있었다. 포인터를 모두 설정한 후에 이를 고칠 수 있었다.

올바른 방식으로 코딩하여 데이터 노드의 가장 오른쪽에서 가장 왼쪽으로 이동하면서 출력할 것을 예상했으나, prev로 이동하던 출력결과가 중간에서 출력을 멈추고 종료하는 것을 볼 수 있었다. 이러한 점을 해결하기 위해서 디버깅을 진행한 결과, 중간 노드에서 노드의 prev가 null로 설정되어 있는 것을 발견하였다. 문제는 index 노드 insert에 있었다. Insert를 진행하면서 만약 root index 노드가 반으로 split 되어 새로 root 가 생기는 경우, 위로 올라가는 데이터의 next를 나뉘기 전 노드의 next로 설정해주고 나뉘기 전 노드의 next의 prev를 올라갈 데이터의 데이터 노드로 설정하지 않았다는 것을 발견했다. 따라서 이 점을 해결하지 않아서 새 root가 생기며 중간 노드는 이전 prev를 update 하지 못하고 null로 남아있었던 것이다. 이 코드를 수정함으로써 문제를 해결할 수 있었다.

B+ tree의 경우 의도치 않게도 $O(N^3)$의 결과를 출력하게 되었다. 이러한 경우 데이터가 매우 커지면 작동하지 않는다는 점이 프로그램의 단점이었다. 함수로 나누어 일정 경우를 제거하는 방식으로 줄일 수 있었다.

그 밖에도 데이터를 리눅스를 옮기면서 글자가 깨지는 경우를 보게 되었다. 이 경우는 리눅스와 윈도우의 인코딩 방식이 다른 점에서 생기는 문제였다. 보통 윈도우는 iso방식을 사용하여 인코딩하는데, 리눅스에서는 한글 등의 유니코드를 utf-8 방식으로 인코딩하

기 때문에 윈도우 방식으로 된 파일은 리눅스에서 올바르게 출력되지 않는 것을 볼 수 있었다. 따라서 파일의 인코딩 방식을 utf-8로 바꿔주고 실행시켰을 때 한글이 깨지지 않고 출력되는 것을 볼 수 있었다.