Data Structure Project

Project #2

담당교수 : 이기훈

제출일 : 2018. 11. 11.

학과 : 컴퓨터공학과

학번 : 2017202088

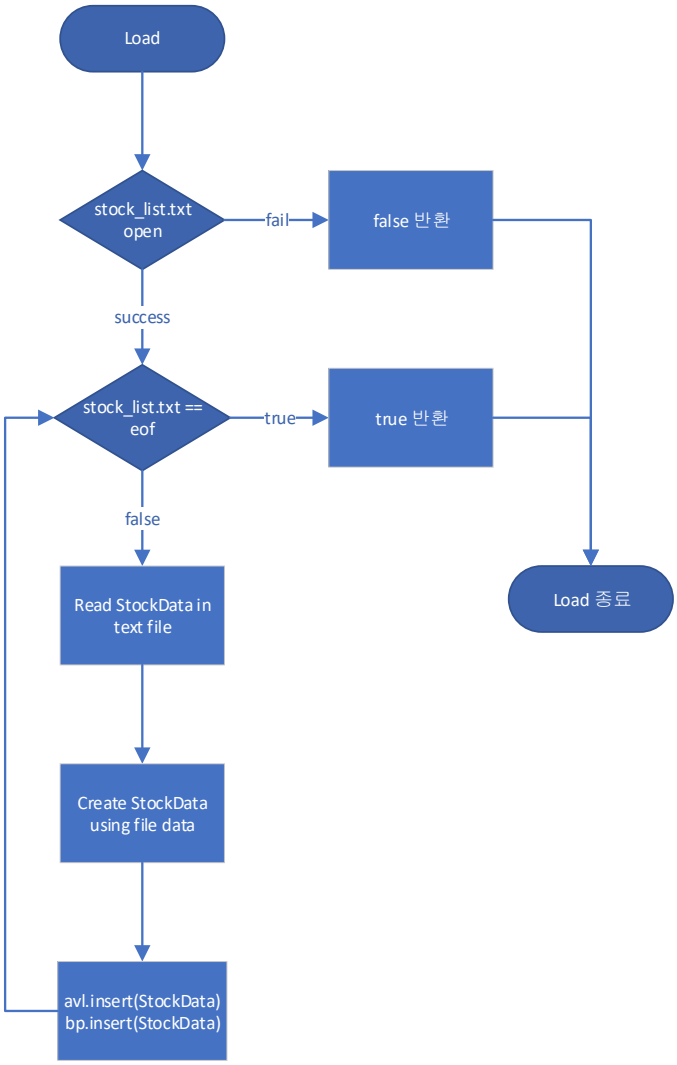
이름 : 신 해 담

1. Introduction

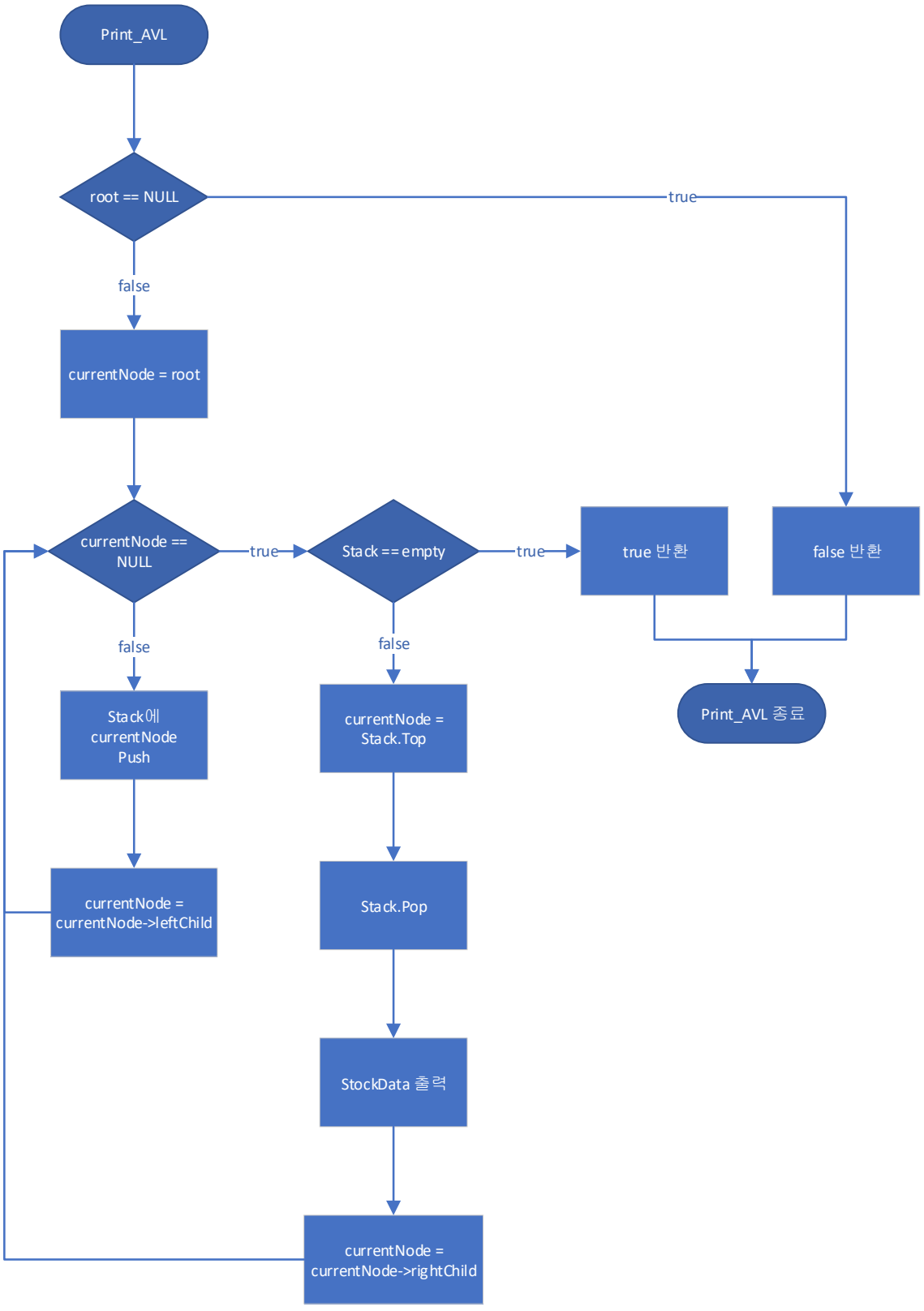
- 주식 정보를 저장하고, 다양한 기준으로 효과적인 검색 서비스를 제공하는 주식 정보 검색 시스템을 구현한다. B+ tree에는 주식들의 주가 데이터를 수익률을 기준으로 정렬하여 저장한다. 이 B+ tree는 저장된 데이터를 출력하는 기능과, 특정 종가 범위에 포함되는 종목들을 검색하는 기능을 가진다. AVL tree에는 주식들의 주가 데이터를 종목 이름을 기준으로 정렬하여 저장한다. 이 AVL tree는 저장된 데이터를 출력하는 기능과, 특정 종목의 수익률 정보를 검색하는 기능이 있다. 이 때, 검색한 종목의 데이터를 Heap에 넣어서 수익률의 순서대로 데이터를 출력하는 기능을 추가로 가진다. 데이터 삭제 또는 수정기능은 구현하지 않는다.

2. Flow Chart

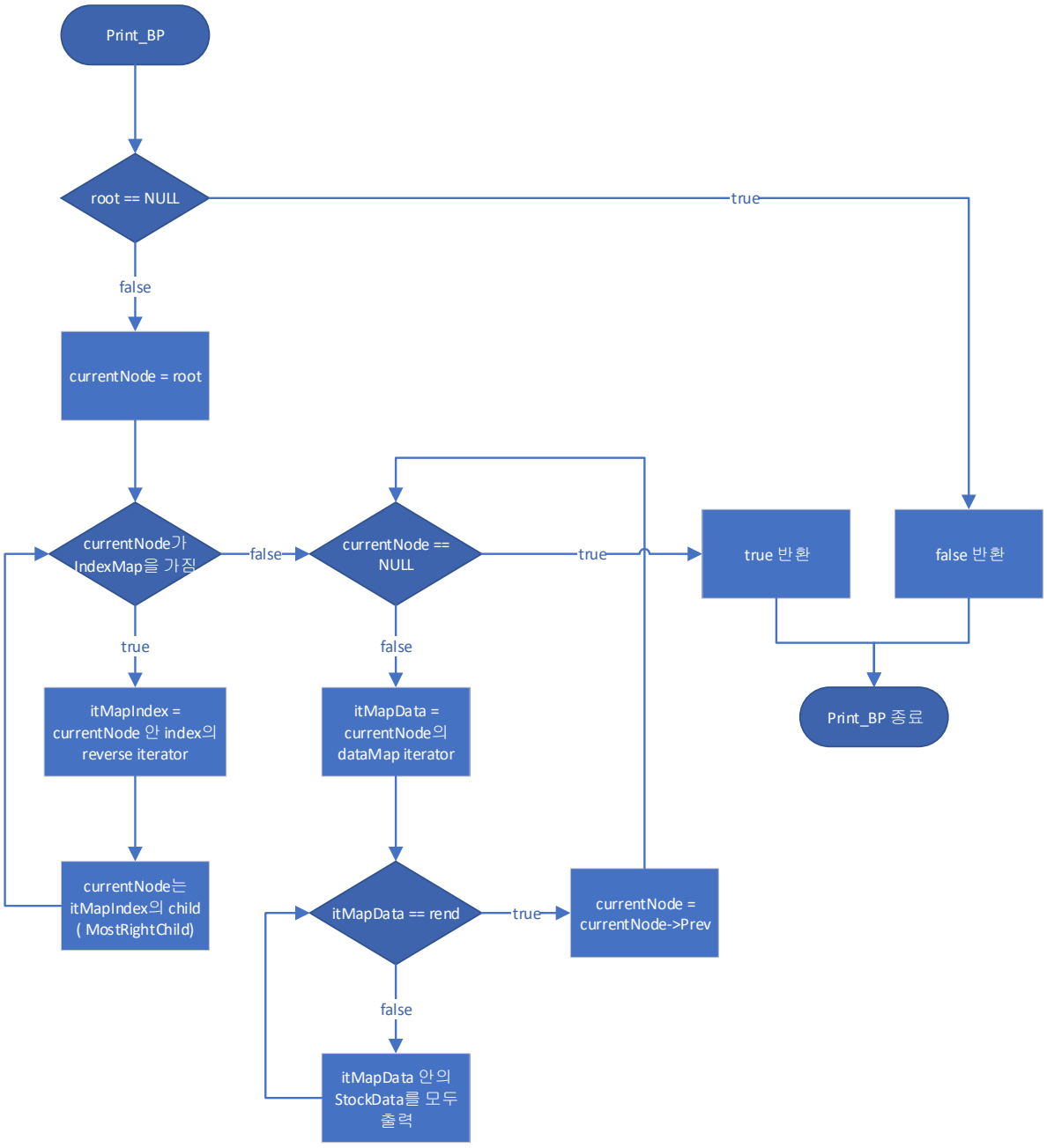
- LOAD의 flow chart. 파일을 열고 데이터를 읽어서 avl과 bp에 insert한다. 파일의 끝을 만나면 true를 반환한다.



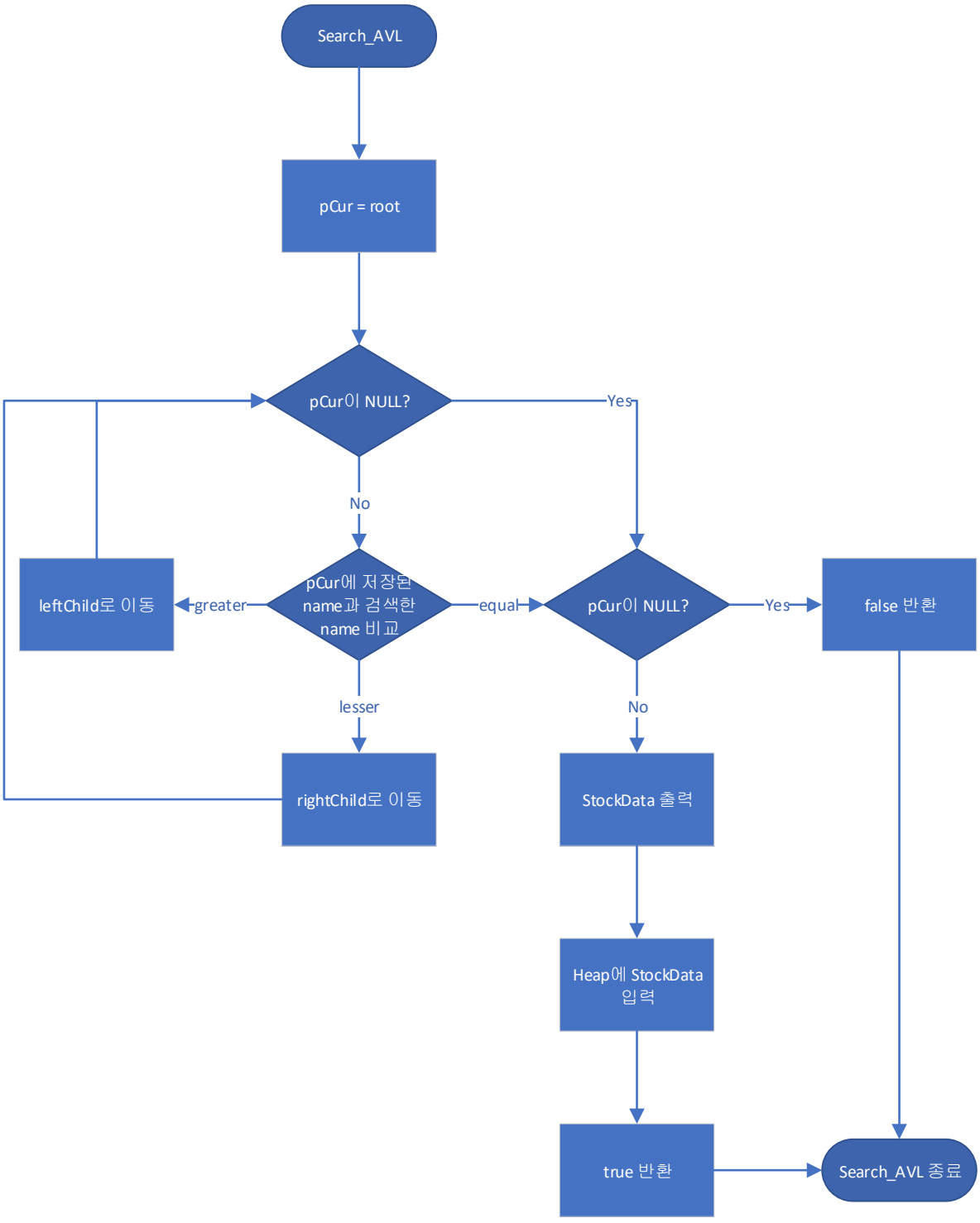
- PRINT_AVL의 flow chart. AVL tree가 empty이면 false를 반환한다. empty가 아니면 저장된 모든 데이터를 Inorder traversal 방식으로 출력하고 true를 반환한다.



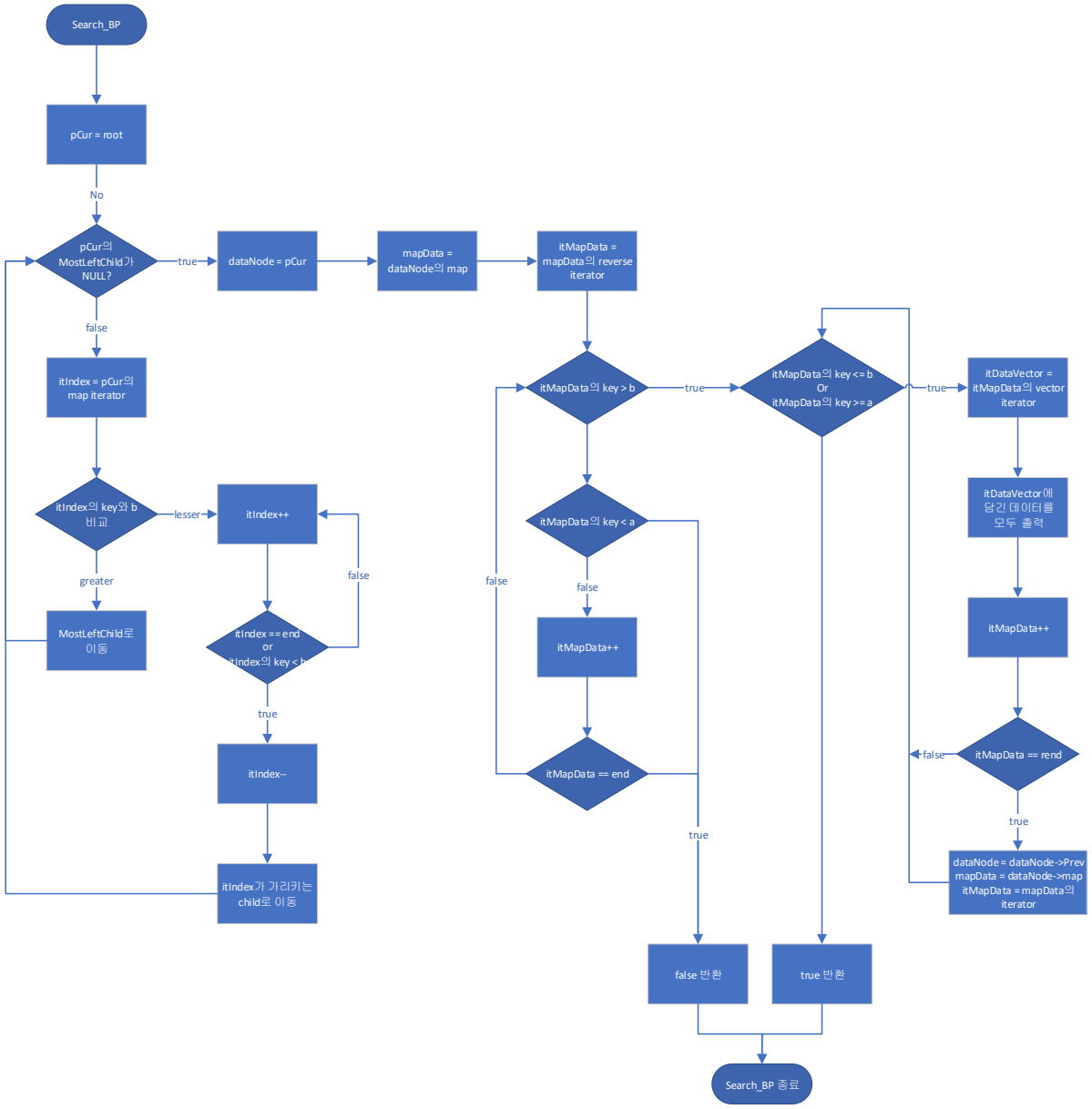
- PRINT_BP의 flow chart. B+ tree의 가장 우측에 있는 데이터 노드에서부터 좌측 데이터 노드로 이동하며 주식 정보를 출력한다. 출력이 끝나면 true를 반환한다.



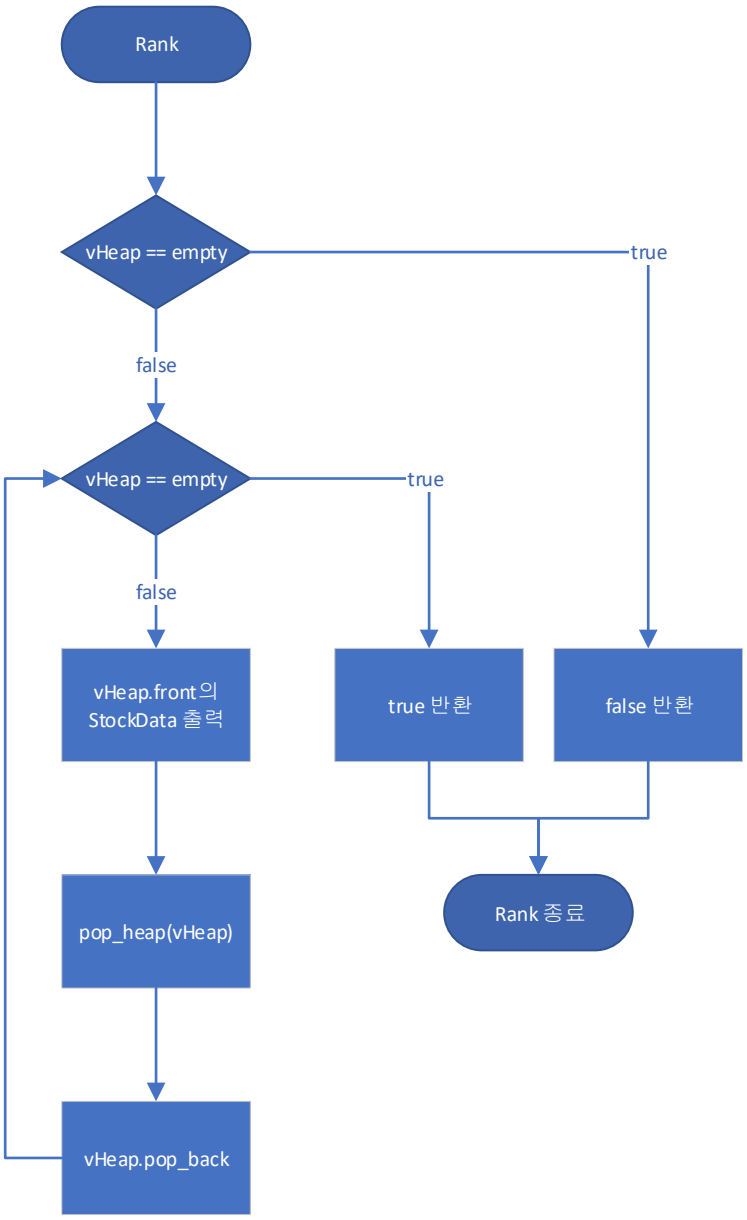
- SEARCH_AVL의 flow chart. 입력받은 name과 같은 name을 가진 노드를 찾아서 트리를 탐색한다. 노드를 찾을 수 없으면 false를 반환한다. 찾았다면 주식 정보를 출력한 뒤 heap에 push하고 true를 반환한다.



- SEARCH_BP의 flow chart. 우선 입력받은 범위의 최대값을 가지는 데이터 노드를 찾는다. 최대값보다 작은 값부터 데이터를 출력하면서 좌측의 데이터 노드로 이동한다. 최솟값보다 작은 데이터가 나오면 출력을 종료하고 true를 반환한다. B+ tree가 empty인 경우, 입력 범위가 잘못된 경우에는 false를 반환한다.



- RANK의 flow chart. Heap가 empty이면 false를 반환한다. empty가 아니면 heap을 pop하며 데이터를 출력한다. Heap가 empty가 되면 true를 반환한다.

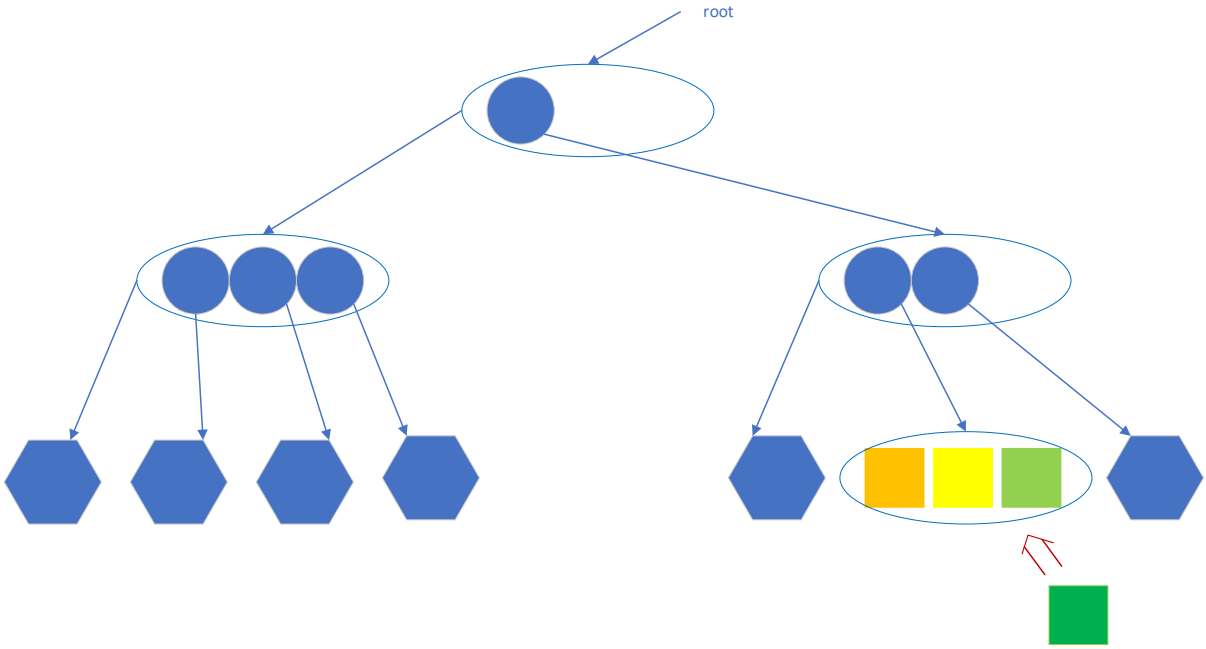


3. Algorithm

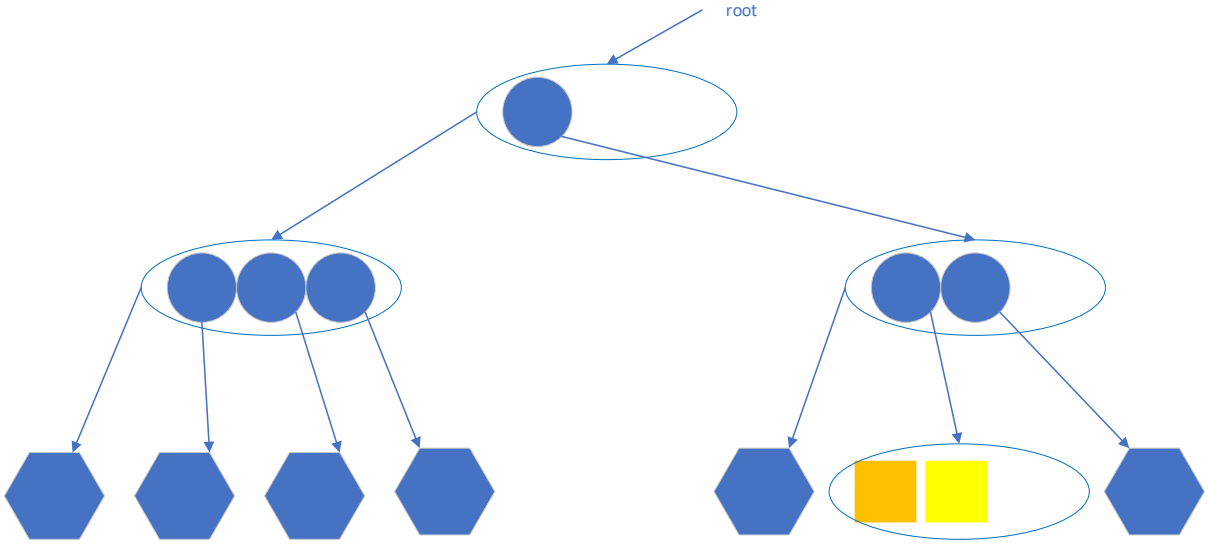
- B+ tree 삽입 알고리즘

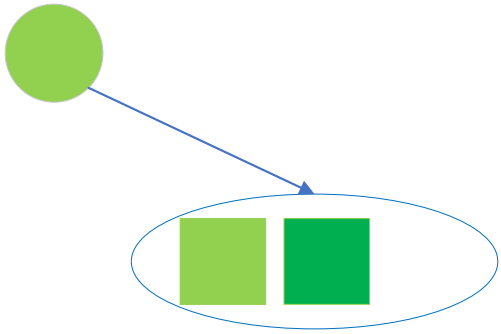
  B+ tree에서 값을 삽입하면 각 노드에서 키값을 비교하여 알맞은 자식 노드로 보낸다. 최종적으로 맞는 데이터 노드를 찾아서 데이터를 삽입한다. 삽입한 데이터 노드의 데이터 수가 order-1의 값과 같으면 데이터 노드를 분리한다. 이 때 노드의 중심값 이상의 데이터를 분리하여 새 데이터 노드를 만들며, 중심값을 인덱스 노드의 키값으로 가지는 subtree를 만든다. 그리고 이 subtree를 기존 데이터 노드의 부모 노

드에 삽입한다. 만약 subtree의 삽입으로 부모 인덱스 노드의 인덱스의 수가 order-1의 값과 같다면 인덱스 노드를 분리한다. 이 때 노드의 중심값은 데이터 노드를 분리할 때와는 다르게 하위 인덱스 노드에 남기지 않는다. 이런 식으로 더 이상 노드의 분리가 일어나지 않거나, 부모 노드가 존재하지 않을 때까지 계속한다.
예를 들어 다음과 같은 B+ tree가 있다고 한다.

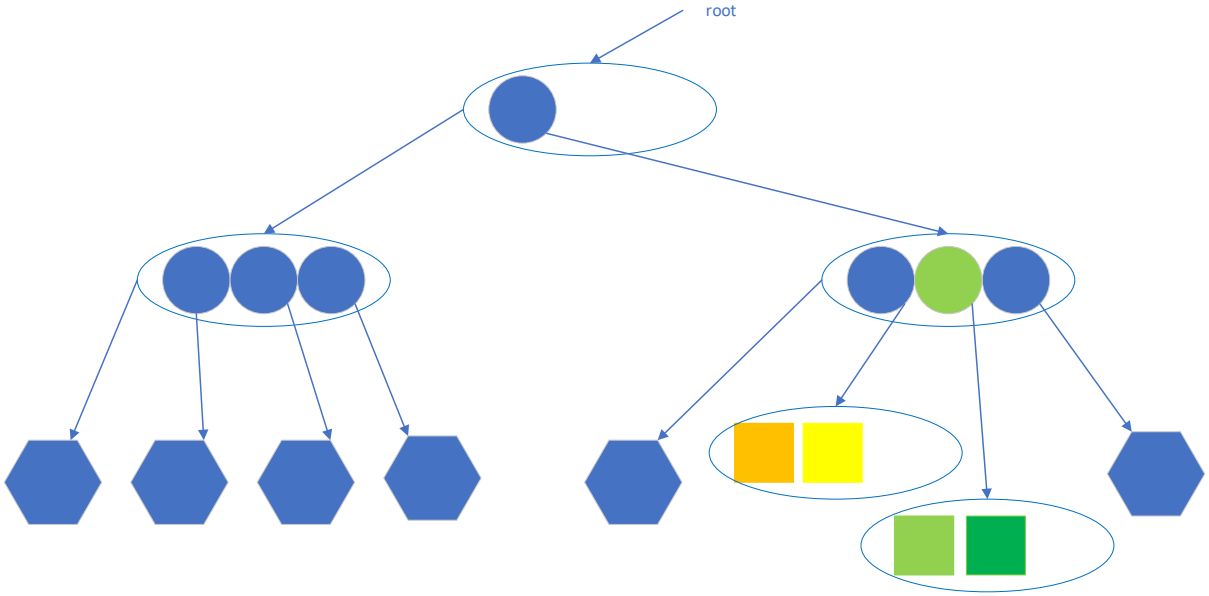


녹색 네모 사각형을 삽입하면 데이터 노드가 오버되어 스플릿이 발생한다.





스플릿으로 생겨난 subtree를 스플릿한 데이터 노드의 부모 노드에 삽입한다.



추가 스플릿을 할 필요가 없으므로 삽입을 마친다.

- AVL tree 삽입 알고리즘

  AVL tree의 각 노드는 balance factor(이하 BF)를 가지고 있어 삽입과 동시에 업데이트 해 주어야 한다. A 노드는 BF를 업데이트하면 -2 또는 +2가 되어 균형이 무너지는 노드이다. 이 노드와 새 노드 삽입 지점 사이의 노드는 모두 BF가 0이다. 왜냐하면 0이 아닌 노드가 나타나면 그 노드가 A 노드로 업데이트되기 때문이다. A 노드의 밸런스가 무너지지 않도록 A 노드를 root로 하는 subtree를 수정해야 한다. 이 때 A 노드의 왼쪽 자식의 왼쪽 subtree에 삽입이 발생하는 경우 (LL), 오른쪽 subtree에 삽입되는 경우 (LR), 오른쪽 자식의 왼쪽 subtree에 삽입이 발생하는 경우 (RL), 오른쪽 subtree에 삽입되는 경우 (RR)의 4가지 케이스가 존재한다.

  LL 케이스에서 A는 BF가 +1로 좌측 subtree의 높이가 더 높다. 이 때 왼쪽 자식의 왼쪽 subtree에 삽입이 발생하면 A 노드의 오른쪽 subtree(AR)의 높이가 h라 할 때, 왼쪽 노드의 왼쪽 subtree(ALL)는 h+1, 오른쪽 subtree(ALR)는 h이다. 이들의 균형을 맞추기 위해 AL을 A의 위치로 올린다. 그리고 A를 AL의 오른쪽 노드로 삽입하고, A

의 왼쪽 노드는 ALR(h), 오른쪽 노드는 AR(h)로 설정하면 밸런스가 잡힌다.

A AL

AL AR(h) => ALL(h+1) A

ALL(h+1) ALR(h) ALL(h) ALR(h)

LR 케이스에서도 BF가 +1로 좌측 sub tree의 높이가 더 높다. 이 때 왼쪽 자식의 오른쪽 sub tree에 삽입이 발생하면 A노드의 오른쪽 subtree(AR)의 높이가 h라 할 때, 왼쪽 노드의 왼쪽 subtree(ALL)는 h, 오른쪽 subtree(ALR)는 h+1이다. 이 때 ALR의 왼쪽 subtree(ALRL)에 추가되는지 오른쪽 subtree(ALRR)에 추가되는지로 한번 더 케이스가 나뉜다. ALRL에 추가되면 ALRL의 높이는 h이고, ALRR은 높이가 h-1, ALRR에 추가되면 ALRL의 높이는 h-1이고, ALRR은 높이가 h이다. 밸런스를 맞추기 위해 ALRL(h/h-1)과 ALL(h)을 같은 부모 노드 AL의 자식으로 설정하고, ALRR(h-1/h)과 AR(h)을 같은 부모 노드 A의 자식으로 설정한다. 그리고 ALR을 기존의 A의 위치로 설정하여 왼쪽 자식으로 AL(< ALR), 오른쪽 자식으로 A(> ALR)로 설정하면 밸런스가 맞는다. ALRL과 ALRR 중 어디에 추가되었는지에 따라 AL과 A의 BF는 0/-1 또는 +1/0이 된다.

A ALR

AL AR(h) => AL(0) A(-1)

ALL(h) ALR ALL(h) ALRL(h) ALRR(h-1) AR(h)

ALRL(h) ALRR(h-1)

A ALR

AL AR(h) => AL(+1) A(0)

ALL(h) ALR ALL(h) ALRL(h-1) ALRR(h) AR(h)

ALRL(h-1) ALRR(h)

RR과 RL은 LL과 LR의 알고리즘을 좌우반전하여 얻을 수 있다.

- Heap push 알고리즘

  Heap에 값을 push하면 heap array의 맨 끝에 값이 들어간다. 그리고 그 array 주소의 절반 위치에 있는 값, 즉 삽입한 노드의 부모가 지닌 값과 값을 비교한다. 부모 노드를 heap의 종류에 따라서 더 큰 값 또는 더 작은 값을 가지면 부모와 값을 바꾼다. 만약 부모와 바꿀 조건을 만족하지 않거나 root에 도달하면 종료한다.

  예를 들어 max heap array에 (x 9 6 4 2 1) 순서로 값이 들어있다고 한다. 10을 이 heap에 push하면 먼저 배열의 맨 뒤에 입력된다.

  (x 9 6 4 2 1 10)

10이 들어간 노드의 부모값 4와 비교하여, 4보다 10이 더 크므로 4와 10을 바꾼다.
(x 9 6 10 2 1 4)
다시 10이 들어간 노드의 부모값 9와 비교하여, 9보다 10이 더 크므로 바꾼다.
(x 10 6 9 2 1 4) <= 최종 heap array
10의 위치가 root이므로 종료한다.

- Heap pop 알고리즘
  Heap의 pop은 root에서 발생한다. Heap의 종류에 따라 root는 heap의 최대 또는 최소값을 가진다. 이 값이 pop되어 제거되면 heap의 빈 공간을 새 값으로 채워야 한다. 따라서 배열의 가장 마지막 값을 root로 올리고 heap을 정렬한다. 우선 새로운 root의 값과 자식 노드의 값을 비교한다. 그리고 더 큰/작은 값을 root로 올린다. 만약 root에 존재하는 값이 가장 크면 pop을 종료한다. root로 값을 올리면서 값이 빈 노드는 그 자식 노드의 값을 비교하여 큰/작은 값을 올린다. 이미 큰/작은 값이 들어가 있으면 pop을 종료한다. 이 과정을 반복하면 heap은 다시 complete tree가 된다.
  예를 들어 (x 10 6 9 2 1 4)의 max heap array를 pop하면 root인 10이 제거된다.
  (x O 6 9 2 1 4)
  그리고 나서 배열의 가장 마지막 값을 root로 올린다.
  (x 4 6 9 2 1)
  그 다음에 root부터 시작하여 값을 정렬해야 한다. root의 자식들은 4보다 크고, 그중 9가 가장 크므로 9의 값을 root에 넣는다.
  (x 9 6 4 2 1) <= 최종 heap array
  그 다음 4의 값을 가진 노드에서 자식 노드와 값을 비교해야 하지만 자식 노드가 존재하지 않으므로 종료한다.

4. Result Screen

- LOAD의 result screen. LOAD가 성공하여 Success가 출력되었다.

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

- PRINT_AVL의 result screen. AVL tree에 저장된 주식 정보를 이름의 오름차순으로 출력한다.

```
======== PRINT ========
7485 겟마블 0.31
시가: 66
종가: 54
거래량: 14
수익률: 0.31
# 중간 생략 #
2311 한국화학 1.15
시가: 52
종가: 86
거래량: 53
수익률: 1.15

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

- PRINT_BP의 result screen. B+ tree에 저장된 주식 정보를 수익률의 내림차순으로 출력한다.

```
======== PRINT ========
9922 성남화학 20.50
시가: 2
종가: 42
거래량: 64
수익률: 20.50
# 중간 생략 #
2132 안나와 -0.49
시가: 76
종가: 1
거래량: 3
수익률: -0.49

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

- SEARCH_AVL의 result screen. 입력받은 종목 이름과 일치하는 주식 정보를 출력한다.

```
========= SEARCH =========
9922 성남화학 20.50
시가: 2
종가: 42
거래량: 64
수익률: 20.50

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

- SEARCH_BP의 result screen. 입력받은 범위 안에 존재하는 주식 정보를 수익률이 높은 순서대로 출력한다. 현재의 결과화면은 0.00~3.00의 데이터를 검색하자 범위 내의 데이터를 출력하였다.

```
========= SEARCH =========
9821 석기전자 2.00
시가: 20
종가: 50
거래량: 90
수익률: 2.00

7485 겟마블 0.31
시가: 66
종가: 54
거래량: 14
수익률: 0.31

4873 리노스 0.23
시가: 69
종가: 51
거래량: 22
수익률: 0.23

1543 나비아 0.16
시가: 54
종가: 36
거래량: 13
수익률: 0.16

8590 지진건설 0.04
시가: 98
종가: 53
거래량: 35
수익률: 0.04

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

- RANK의 result screen. SEARCH_AVL로 검색했던 주식 정보를 수익률이 높은 순서대로 출력한다. 이 경우 성남화학과 단절통신을 SEARCH_AVL로 검색한 다음 RANK를 수행하여 나온 결과이다.

```
======== RANK ========
9922 성남화학 20.50
시가: 2
종가: 42
거래량: 64
수익률: 20.50

2354 단절통신 0.75
시가: 76
종가: 95
거래량: 89
수익률: 0.75

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

- EXIT의 result screen. 프로그램을 종료하고 Success를 출력한다.

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

5. Consideration

Project 코드를 본격적으로 작성하기에 앞서, 스켈레톤 코드가 어떤 식으로 작성되어 있고, 어떤 역할을 하는 변수, 함수인지 이해한 다음 시작해야 한다.

BpTree에 데이터를 삽입할 때 MostLeftChild를 설정하지 않아 문제가 발생했다. MostLeftChild 설정은 tree를 split할 때 설정해주어 해결했다.

Visual studio에서는 strtok 함수를 사용하면 컴파일 에러가 발생한다. Visual studio에서 원활한 디버깅을 수행하기 위해서 strtok 대신 istringstream을 사용하여 파일을 읽는 데 사용했다.

Map과 vector에 저장된 데이터에 쉽게 접근하게 해주는 iterator로 SEARCH_BP를 구현하면서, end로 시작하여 operator--로 접근하면 빈 iterator를 읽는 오류가 발생했다. 따라서 데이터에 역으로 접근하는 방법이 필요하여 찾아보니 역순의 iterator를 만들어주는 reverse_iterator와 rbegin, rend 를 발견할 수 있었다. 이를 사용하여 구현함으로써 데이터 노드에 저장된 map에 역순으로 접근하여 SEARCH_BP를 구현할 수 있었다.

SEARCH_BP에서 while loop condition으로 iterator가 end인지와 key값이 b보다 큰지를 동시에 체크했더니 에러가 발생했다. Iterator가 end일 때 key에 접근해서 발생한 오류였다. end 체크를 loop 안에서 함으로써 해결했다.

BpTree의 스플릿 함수에서 map의 erase함수를 수행할 때 에러가 발생했다. 디버깅하여 원인을 찾아보니 erase를 시도하면 동시에 iterator도 지워져 end상태의 iterator를 읽어 발생하는 에러였다. 이를 해결하기 위해 vector를 선언하여 지울 iterator의 key를 push해서 넣어놓았다. iterator를 다 쓰고 난 후에 vector에 저장된 key의 데이터를 erase하고 pop함으로써 에러를 해결할 수 있었다.

BpTree를 print해서 확인한 결과 BpTree를 생성하면서 오류가 발생하는 것을 발견했다. 2.00보다 수익률이 큰 데이터는 성남화학 뿐이었다. 문제 발생 위치는 트리를 스플릿하는 함수에서라고 생각하여 검토하였으나 문제점을 찾지 못했다. 정확한 문제 발생 위치를 특정하기 위해 출력되지 않은 데이터와 트리에 삽입하면서 스플릿이 발생하는 데이터를 비교하여 보았지만 특정한 관계는 존재하지 않았다. 트리의 MostLeftChild에서 데이터를 출력하자 모든 데이터가 출력되었지만 11번째 이상의 데이터는 정렬이 되어 있지 않았다. 결국 이를 해결하지 못하였다.