데이터구조설계&실습 2차 Project

학과: 컴퓨터정보공학부

학번: 202220208

이름: 김재용

1. Introduction

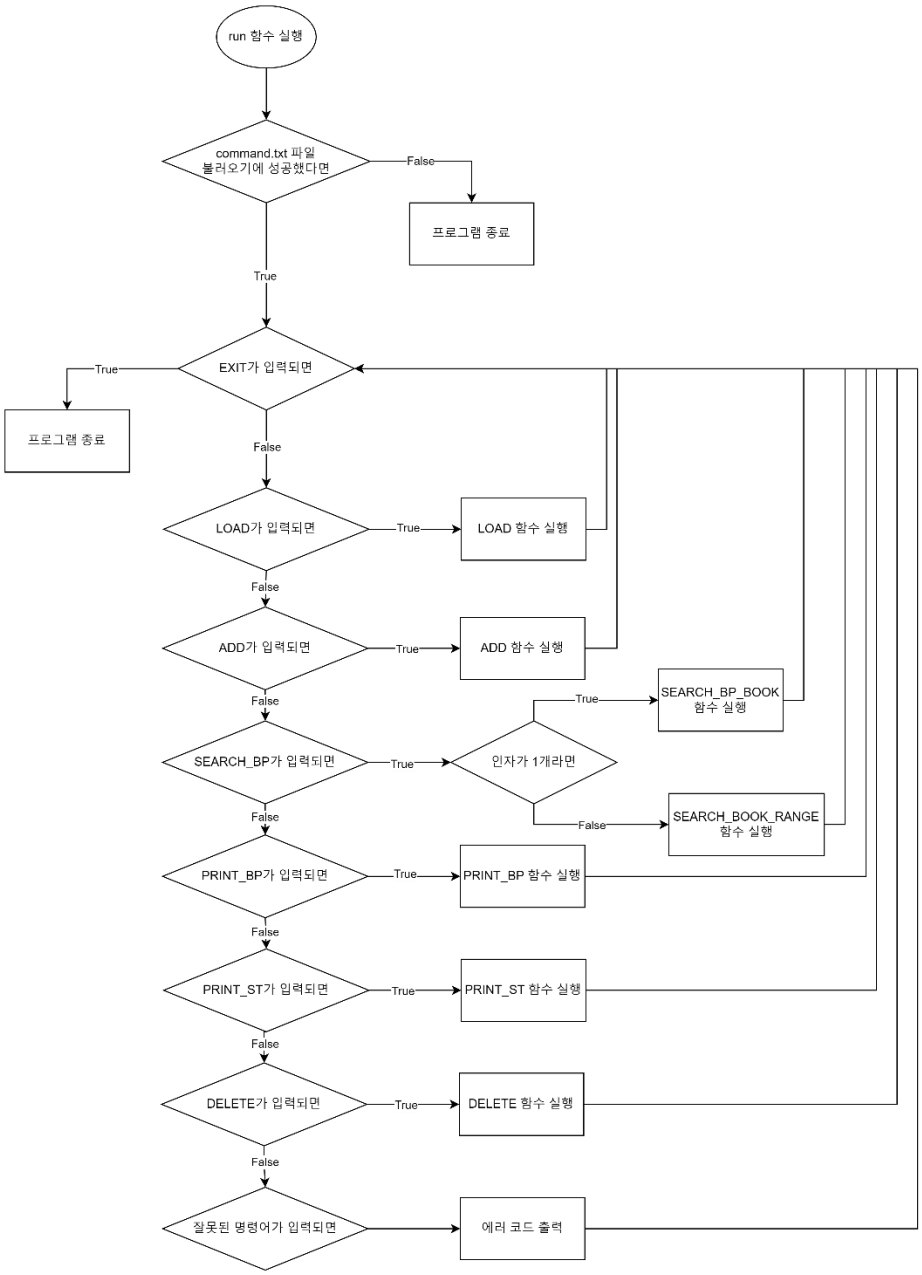
B+ Tree, Selection Tree, Heap을 이용하여 도서 대출 관리 프로그램을 구현한다. 대출 관리 프로그램은 도서명, 도서 분류 코드, 저자, 발행 연도, 대출 권수를 관리하고, 이를 이용해 대출 중인 도서와 대출 불가 도서에 대한 정보를 제공할 수 있다.

B+ Tree에서는 대출 가능한 도서를 관리한다. 처음에 Data를 LOAD할 때 모든 데이터가 B+ Tree 안으로 삽입되고, 이후의 명령어를 통해서 대출 권수가 일정 수를 넘어가면 B+ Tree애서 삭제한 후 Selection Tree로 삽입하게 된다.

Selection Tree에서는 대출이 불가능한 도서를 관리한다. B+ Tree에서 대출 권수가 일정 권수 이상을 넘어간 도서들은 B+ Tree에서 삭제한 후 Selection Tree로 이동한다. Selection Tree의 각 Leaf Node는 Heap 자료 구조와 연결되어 있다. B+ Tree에서 삭제되어 Selection Tree로 삽입해야 하는 자료들은 각 도서 분류 코드에 따라서 Heap에 나누어 저장하게 된다. 이후 Heap이 재정렬이 되는 경우 Selection Tree를 재정렬하게 된다.

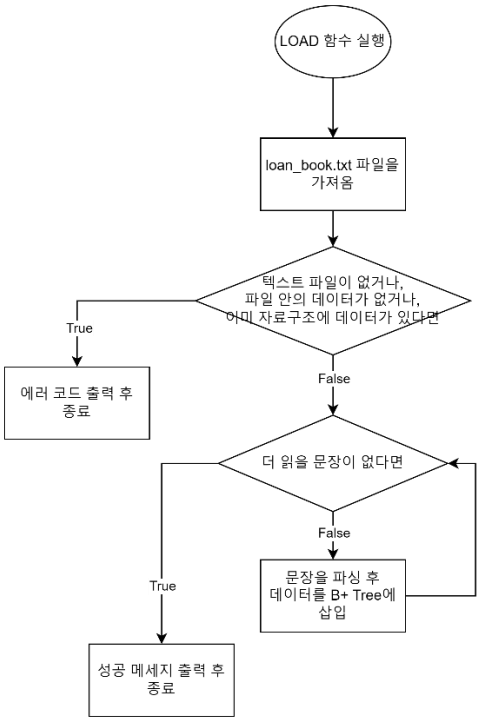
2. Flowchart

RUN



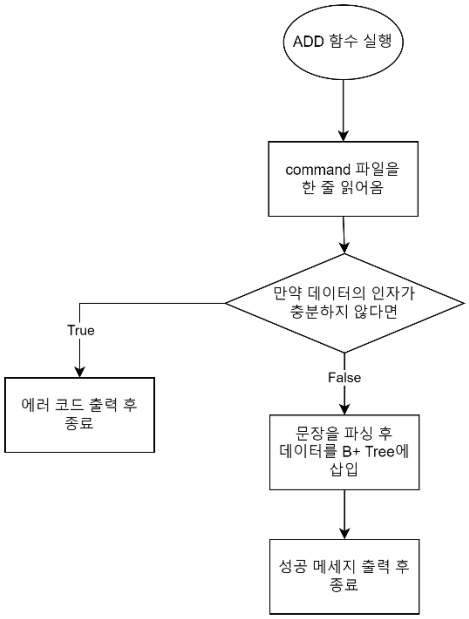
이는 전체적인 프로그램의 동작을 담당하는 run 함수의 동작 방식이다. 명령어를 입력받고 처리하는 모습을 보여준다.

LOAD



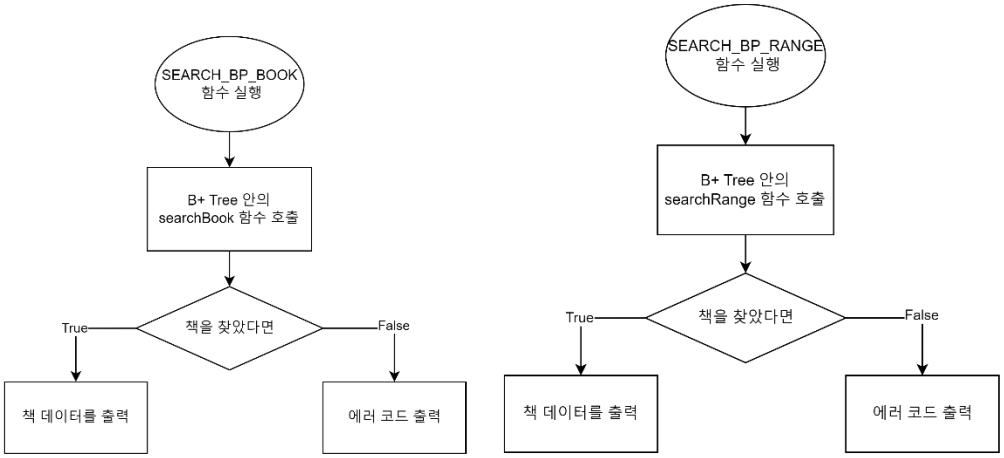
LOAD 함수의 플로우 차트이다. 만약 loan_book.txt 파일이 없거나, 파일 안의 데이터가 존재하지 않거나, 자료 구조에 데이터가 있는 경우 에러 코드를 출력하고, 그렇지 않은 경우엔 텍스트 파일의 끝까지 문장을 읽고 파싱해서 데이터를 삽입한다. 이 때, 잘못된 데이터가 들어온다는 가정은 하지 않는다.

ADD



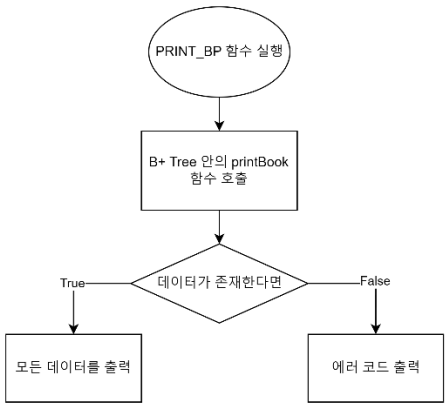
ADD 명령어를 실행한 모습이다. 문장을 파싱하는데, 파싱 중 데이터가 들어와야 하는데 널 문자가 들어온다면 인자가 부족하다는 뜻이므로 그 때는 에러 코드를 출력하고, 그렇지 않은 경우는 정상적으로 데이터를 만든 후 B+ Tree에 삽입한다.

SEARCH_BP



하나의 책을 출력하는 경우와, 범위를 출력하는 경우로 나눠지는데 우선 두 경우는 run 함수 내에서 따로 처리해준 후, bptree 안에 있는 각각의 함수를 호출한다. 이는 bool 형식으로 선언되어 있어, 반환값이 참이라면 책 데이터를 출력하고, 거짓이라면 에러 코드를 출력한다. 데이터를 찾는 과정은 Binary Search Tree와 거의 유사하므로 굳이 넣지 않았다.

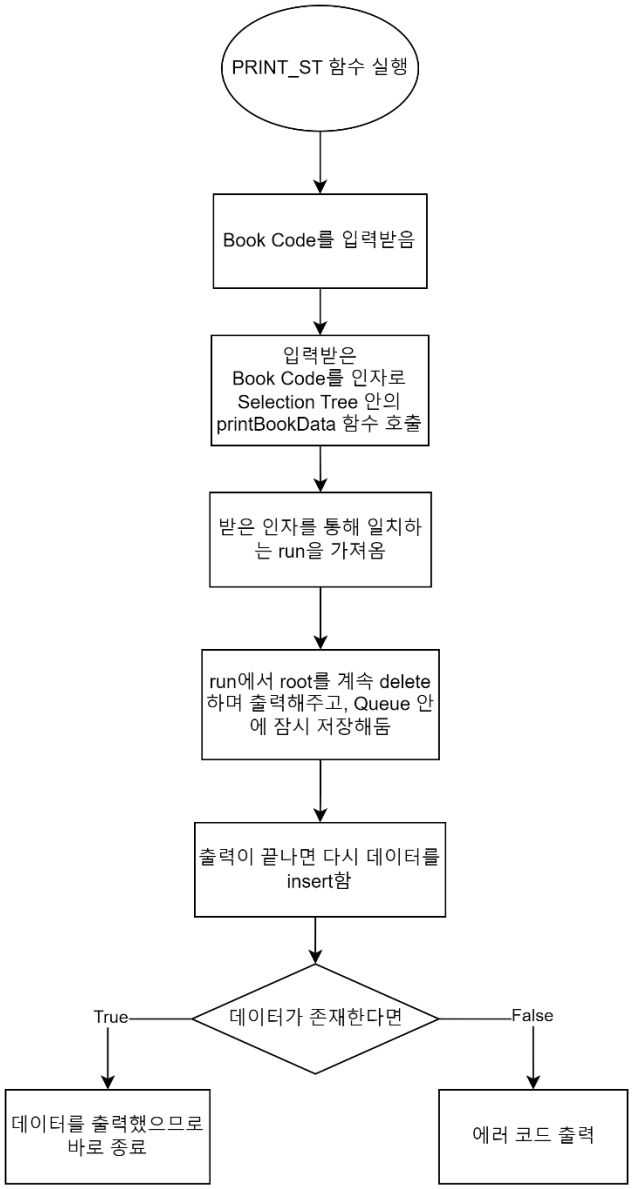
PRINT_BP



위의 SEARCH_BP와 크게 다르지 않은 모습이다. B+ Tree의 맨 앞 데이터를 가져온 후 이 데이터가 존재한다면 출력하고, 그렇지 않다면 에러 코드를 출력한다. 이 또한 함수가 bool 형식으로 되어 있어 반환형이 true이면 데이터 출력, false이면 에러 코드를 출력하

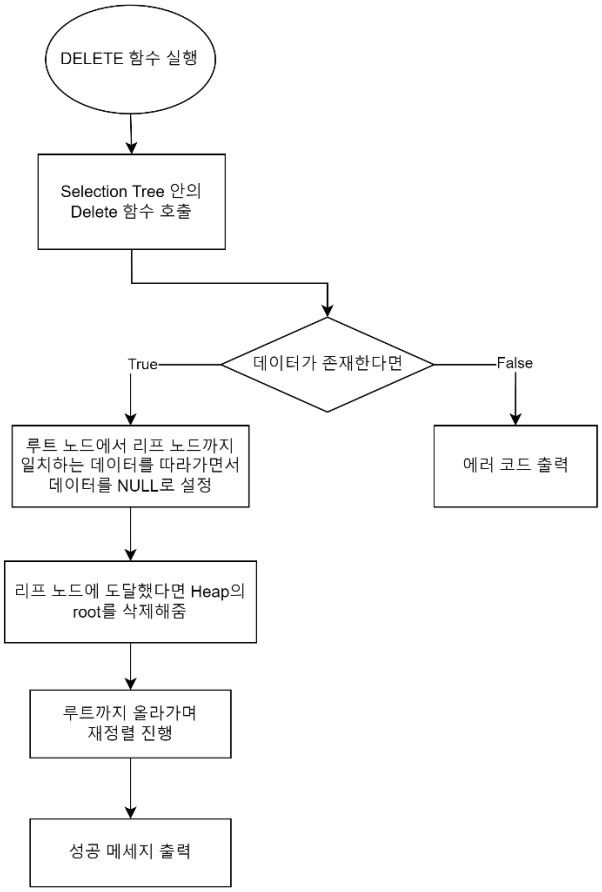
면 된다.

PRINT_ST



Selection Tree에서 Print를 하는 함수의 플로우차트이다. 이는 인자로 Book Code를 입력받기 때문에 이를 입력받고, 함수를 호출한다. 위 함수도 bool 형식으로 되어 있기 때문에 반환값이 true이면 데이터를 출력한 것이고, false이면 데이터를 출력하지 않았으니 에러 코드를 출력하면 된다. 각각의 run에서 root를 계속 delete하고 이를 잠시 Queue 안에 저장한 다음 출력한 후, 모두 출력이 끝나면 다시 Heap 안에 저장해준다.

DELETE



Selection Tree에서 루트 노드에 있는 데이터를 삭제하는 함수이다. 루트 노드에 데이터가 존재한다면, 이 데이터의 시작점이 되는 리프 노드까지 이동하며 데이터를 NULL로 설정해주고, 리프 노드의 Heap 안의 루트를 제거해준다. 이후 다시 Selection Tree의 루트까지 올라가며 재정렬을 진행한다. 재정렬은 변화가 생긴 부분에서 토너먼트를 한번 진행하면 완성된다.

3. Algorithm

- B+ Tree

B+ Tree는 Multiway Search Tree의 한 종류로 B Tree의 단점을 보완하기 위해서 만들어진 Tree이다. 먼저 B Tree에 대해서 설명을 하자면, AVL Tree에서 데이터의 수가 커지면 커질수록 아무리 Full Binary Tree라고 해도 Tree의 Height가 높아지는 것을 막지 못하기 때문에 Tree의 Height를 보다 더 줄일 수 있는 방법을 고안한 것이 B Tree이다. 이는 기본적으로 binary Search Tree와 같이 모든 데이터가 정렬되어 있고, 하나의 노드에 여러 개의 데이터가 들어갈 수 있는 것이 특징이다. 이 프로젝트에서 구현한 알고리즘은 B

Tree가 아닌 B+ Tree이므로 자세한 데이터 삽입, 삭제 연산은 설명하지 않고 B+ Tree로 넘어가겠다.

다음으로, B+ Tree는 위에서 설명한 B Tree에서 업그레이드된 버전인데, 이는 두 형식의 노드로 구성되어 있다. 가장 Leaf Node를 담당하는 Data Node와, 그 위의 노드를 담당하고 있는 Index Node이다. Data Node는 말 그대로 트리에서 데이터를 담당하는 노드이고, Index는 데이터는 저장하지 않고 그 데이터에 접근하기까지의 인덱스를 저장하는 노드이다. 각각의 Data Node끼리는 서로 Next Node와 Prev Node가 양방향으로 연결되어 있어 여러 데이터를 출력할 때는 Root부터 계속 탐색하는 것이 아니라 출력한 위치에서 다음 노드에 접근하면 되므로 여러 데이터에 접근하기가 더 쉬워진다.

B+ Tree에서 데이터를 삽입할 때에 경우의 수가 여러 가지 있다. 먼저 첫 번째로는 아무 데이터도 없는 경우이다. 이럴 때는 데이터 노드를 새로 생성해준 후 이를 루트로 지정한다. 다음으로는 굳이 데이터 노드의 Split이 필요 없는 경우이다. 이 경우 또한 이미 있는 데이터 노드에 추가적으로 데이터를 삽입한 후 종료한다. 그 다음으로는 데이터 노드에서 Split이 일어나는 경우이다. 이 경우 데이터 노드를 두개로 분할해준 후 이미 Parant Node가 있는 경우는 그 노드에 새로운 인덱스를 추가해주고, Parant Node가 없는 경우라면 새로운 Index Node를 생성해주고 Child와 Parant Node를 지정해준다. 만약 Parant Node에서 인덱스를 추가해주는 경우에는 인덱스 노드의 데이터의 개수를 확인해본 후 Split이 필요하다면 Split을 진행한다. Index Node에서 Split이 Data Node와 다른 점은, Data Node에서는 Split을 진행할 때 데이터를 삭제하지는 않지만, 인덱스 노드는 중앙에 있는 데이터를 Parant로 올리면서 그 노드에 있는 데이터는 삭제해야 한다.

B+ Tree에서 데이터를 삭제할 때에도 몇 가지의 경우의 수가 있다. 먼저 가장 간단한 경우인 삭제할 키가 인덱스 노드에 없고, 데이터 노드에서 가장 처음 키가 아닌 경우이다. 이 경우 그냥 Map에서 삭제한 후 종료하면 된다. 그러다 삭제할 키가 데이터 노드의 가장 처음 키인 경우는 여러 가지로 나뉠 수 있다. 삭제할 노드가 데이터의 가장 처음 키이고 왼쪽, 오른쪽 형제 노드 모두 최소 개수의 데이터를 가지고 있다면 먼저 부모 노드와 형제 노드를 병합한 후, 그 위의 루트 또한 다시 정리해준다. 이후 데이터를 삭제하고, 인덱스 노드의 키 값을 변경해주면 된다. 혹은 왼쪽 형제 노드가 데이터를 최소 개수 이상으로 가지고 있다면 데이터 값을 삭제해야 하는 노드로 가져온 후 삭제할 데이터를 삭제해주고, 인덱스 노드의 키값만 변경해줄 수 있다. 이는 오른쪽 형제 노드가 데이터를 최소 개수 이상으로 가지고 있어도 이 과정이 가능하며, 노드의 맨 첫번째 데이터가 삭제되는 경우라면 인덱스 노드의 값을 업데이트 해주는 과정이 필요하다.

- Selection Tree

Selection Tree는 Leaf Node의 데이터들 중에서 가장 작은 혹은 큰 데이터를 루트 노드로 옮겨오는 트리이다. 이는 토너먼트를 진행하는 것과 유사하게 진행된다. 각각의 Run에서 가장 작은 혹은 큰 데이터를 가져와서 루트까지 토너먼트를 진행해서 루트에 결국 가장 작은 혹은 큰 데이터가 들어오게 된다. 이 프로젝트에서는 Run이 8개로 고정되어 있었기 때문에 미리 트리를 구성해두었고, Min Selection Tree를 구현한다. 각각의 Run마다 Heap과 연결되어 있고, 이 Heap의 Root를 가져와서 토너먼트를 진행한다.

위 프로젝트에서 Insert를 진행할 때, 각각의 Run을 구성하는 Heap 안에 Insert를 진행하게 되고, 만약 Heap의 루트에 변화가 있다면 Selection Tree도 재정렬이 필요하므로 새로운 root로 Selection Tree의 부모 노드에 있는 데이터보다 더 작다면 재정렬을 Selection Tree의 루트까지 진행한다.

또, Delete를 진행하면, Root에 있는 데이터를 얕은 카피를 통해 가져오기 때문에 완전히 삭제하지는 않고 Data를 NULL로 지정해준 후 이 과정을 Leaf Node까지 반복해주고 Leaf Node에 도착한다면 Heap의 Delete 함수를 호출해준다. 이후 다시 Heap의 루트 노드를 가져와서 Selection Tree의 재정렬을 진행한다.

- Heap

Heap은 루트 노드에 항상 가장 크거나 작은 값이 존재하는 자료 구조이다. 위 프로젝트에서는 Min Heap을 구현하므로 루트 노드에 오름차순으로 정렬했을 때 가장 작은 값이 있어야 한다. 그 아래 노드는 제대로 정렬이 되어 있지는 않지만, 항상 부모 노드와 비교했을 때 자식 노드는 부모 노드보다 큰 값이 들어가 있도록 구성되어 있다.

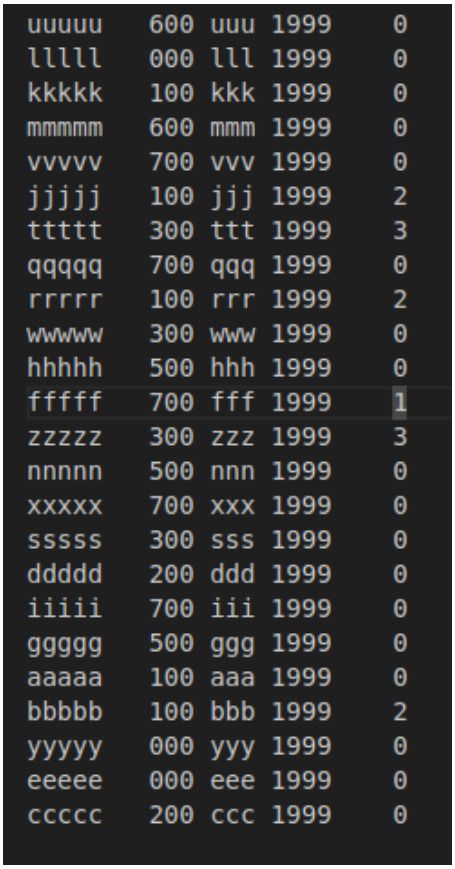
Heap에 데이터를 삽입하는 방식은 가장 마지막 인덱스에 데이터를 추가해준 후, Parant Node와의 값 비교를 통해 Parant Node보다 데이터가 작다면 값을 서로 변경해주는 식으로 Root까지 반복해주며 구현했다. 이와 유사하게 데이터를 삭제하는 방식은 먼저 Root 안의 데이터를 삭제한 후, 가장 마지막 인덱스에서의 데이터를 루트로 옮긴 후 자식 노드와 비교하며 자신보다 작은 노드가 없을 때까지 이를 반복한다.

링크드 리스트 방식에서 마지막 인덱스를 찾는 방법에서 고민이 많았는데, 인덱스를 이

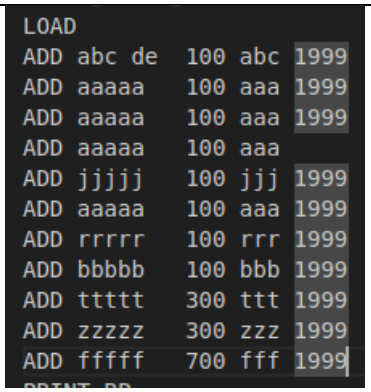
진수로 변화한 다음 규칙을 찾아보니 알 수 있었다. 맨 첫 번째 비트는 버리고 그 다음 비트부터 0이라면 왼쪽 자식 노드, 1이라면 오른쪽 자식 노드를 탐색하면 원하는 위치로 접근할 수 있었다.

4. Result Screen

먼저 LOAD 파일에서는 책 이름을 알아보기 쉽게 하기 위하여 같은 알파벳 문자 5개로만 구성하였다.

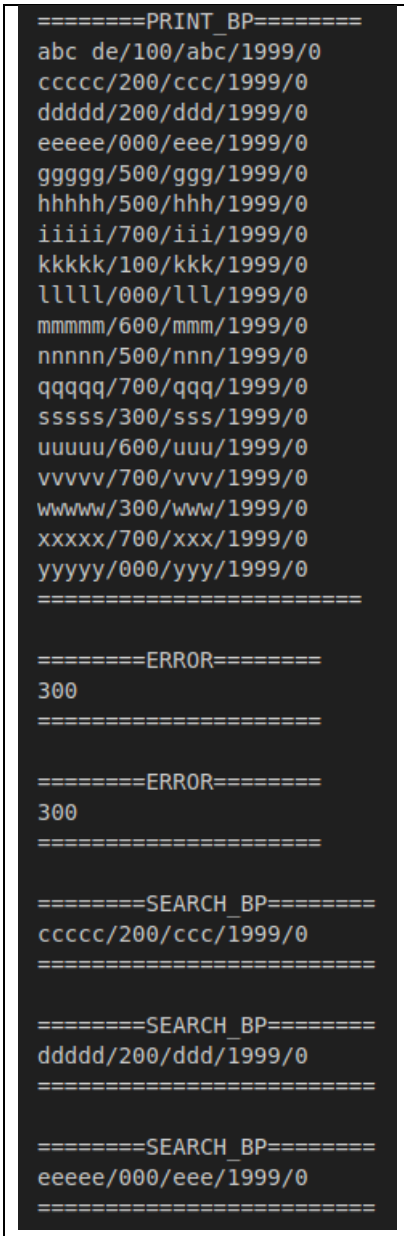


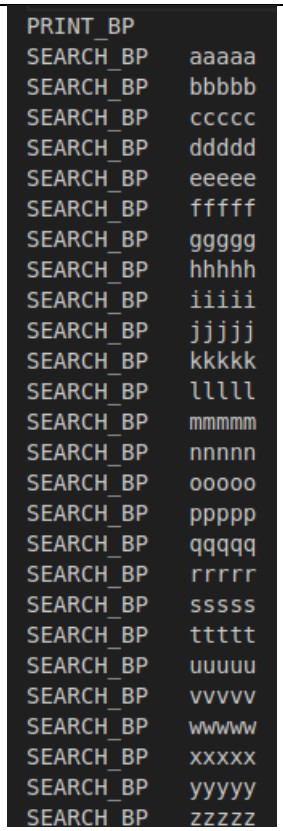
위 사진은 loan_book.txt의 모습이다. 이후 Selection Tree로 이동하는 모습을 보기 위해서 중간에 Count를 2, 3개로 넣어둔 모습도 볼 수 있다. 굳이 알파벳 순서가 아니라 임의의 순서로 넣은 이유는 B+ Tree에 정렬이 제대로 진행되며 삽입되는지 확인하기 위해서이다. 중간에 Search_BP_Range를 확인하기 위해서 ooooo와 ppppp가 빠져 있다. 다른 알파벳들은 모두 존재한다.



왼쪽의 사진은 위 명령어들을 실행했을 때의 사진이다. 만약 데이터의 인자가 적은 경우 에러 코드를 잘 출력하는 모습을 볼 수 있고, 그렇지 않은 경우는 B+ Tree 안에 데이터를 삽입한 후 잘 출력해주는 모습을 볼 수 있다.

띄어쓰기도 책 이름으로 잘 들어오는 것을 보이기 위해 abc de라는 인풋을 넣어보았다. 이 또한 제대로 인식하는 것을 볼 수 있다.





왼쪽의 사진은 위 명령어를 실행했을 때 log 파일의 사진이다. 중간중간 빠진 알파벳들이 보이는데, 이는 ADD 명령어를 통해 책의 일정 숫자를 넘겨 B+ Tree에서 삭제한 후 Selection Tree로 넘어갔기 때문이고, 나머지 문자들이 오름차순으로 잘 출력되는 모습을 볼 수 있다.

또, SEARCH_BP로 B+ Tree 안의 모든 데이터를 찾아보았고, 찾지 못했다면 300 Error Code를 출력하고, 찾았다면 제대로 출력하는 모습을 보여준다.

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

========SEARCH_BP========
ggggg/500/ggg/1999/0
=========================

========SEARCH_BP========
hhhhh/500/hhh/1999/0
=========================

========SEARCH_BP========
iiiii/700/iii/1999/0
=========================

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

========SEARCH_BP========
kkkkk/100/kkk/1999/0
=========================

========SEARCH_BP========
lllll/000/lll/1999/0
=========================

========SEARCH_BP========
mmmmm/600/mmm/1999/0
=========================

========SEARCH_BP========
nnnnn/500/nnn/1999/0
=========================

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

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

========SEARCH_BP========
qqqqq/700/qqq/1999/0
=========================

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

========SEARCH_BP========
sssss/300/sss/1999/0
=========================

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

========SEARCH_BP========
uuuuu/600/uuu/1999/0
=========================

========SEARCH_BP========
vvvvv/700/vvv/1999/0
=========================

========SEARCH_BP========
wwwww/300/www/1999/0
=========================

========SEARCH_BP========
xxxxx/700/xxx/1999/0
=========================

========SEARCH_BP========
yyyyy/000/yyy/1999/0
=========================

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

```
========SEARCH_BP========
abc de/100/abc/1999/0
ccccc/200/ccc/1999/0
=========================

========SEARCH_BP========
ccccc/200/ccc/1999/0
ddddd/200/ddd/1999/0
eeeee/000/eee/1999/0
=========================

========SEARCH_BP========
abc de/100/abc/1999/0
ccccc/200/ccc/1999/0
ddddd/200/ddd/1999/0
eeeee/000/eee/1999/0
ggggg/500/ggg/1999/0
hhhhh/500/hhh/1999/0
iiiii/700/iii/1999/0
kkkkk/100/kkk/1999/0
lllll/000/lll/1999/0
mmmmm/600/mmm/1999/0
nnnnn/500/nnn/1999/0
qqqqq/700/qqq/1999/0
sssss/300/sss/1999/0
uuuuu/600/uuu/1999/0
vvvvv/700/vvv/1999/0
wwwww/300/www/1999/0
xxxxx/700/xxx/1999/0
yyyyy/000/yyy/1999/0
=========================

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

```
SEARCH_BP   a   c
SEARCH_BP   b   e
SEARCH_BP   a   y
SEARCH_BP   o   p
```

다음은 SEARCH_BP 명령어 중에서 범위를 통해 탐색하는 명령어이다.

o부터 p까지는 아무 데이터도 존재하지 않기 때문에 에러 코드를 출력하는 모습을 볼 수 있고, 나머지 경우는 모두 정상적으로 출력이 되는 모습을 확인할 수 있다.

```
========ERROR========
500
====================

========PRINT_ST========
aaaaa/100/aaa/1999/3
bbbbb/100/bbb/1999/3
jjjjj/100/jjj/1999/3
rrrrr/100/rrr/1999/3
========================

========PRINT_ST========
aaaaa/100/aaa/1999/3
bbbbb/100/bbb/1999/3
jjjjj/100/jjj/1999/3
rrrrr/100/rrr/1999/3
========================

========DELETE========
Success
====================

========DELETE========
Success
====================

========PRINT_ST========
fffff/700/fff/1999/2
========================

========DELETE========
Success
====================

========ERROR========
500
====================
========PRINT_ST========
jjjjj/100/jjj/1999/3
rrrrr/100/rrr/1999/3
========================

========DELETE========
Success
====================

========DELETE========
Success
====================

========DELETE========
Success
====================

========DELETE========
Success
====================

========ERROR========
600
====================

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

```
PRINT_ST    200
PRINT_ST    100
PRINT_ST    100
DELETE
DELETE
PRINT_ST    700
DELETE
PRINT_ST    700
PRINT_ST    100
DELETE
DELETE
DELETE
DELETE
DELETE
EXIT
```

다음은 PRINT_ST와 DELETE 명령어의 테스트이다. 처음에 도서 코드가 200인 경우는 없으므로 오류 코드를 출력했고, DELETE를 진행한 후 모든 Run에서 정상적인 값이 삭제되는 모습을 볼 수 있다.

또, Selection Tree에 값이 존재하지 않는 경우에는 오류 코드 600을 정상적으로 출력하는 모습을 볼 수 있다.

마지막으로 EXIT 명령어를 입력하면 프로그램이 정상적으로 종료된다.

5. Consideration

B+ Tree 데이터 삽입 연산을 진행할 때 여러가지 문제가 생겼다. 삽입 연산 자체는 잘 작동했지만 Split을 진행할 경우, 특히 Index Node Split을 진행할 경우에 각각의 Child와 Parant가 잘못 Setting되는 경우가 많이 생겼다. 이는 Split을 해서 새로운 노드를 만드는 과정에서 원래 노드에서 이동되어 온 자식 노드의 부모 노드를 다시 세팅하지 않아서 생기는 문제였다. Split을 하는 과정을 너무 간단하게 생각하고 꼼꼼히 체크하지 않아서 생긴 문제라고 판단했다.

Selection Tree와 Heap을 구현할 때 Linked List를 사용하여 구현했는데, 특히 Heap을 구현할 때 시간 복잡도 면에서 고민이 있었다. Heap의 마지막 노드의 위치를 어떻게 구해야 할지 고민이 있었는데, 모든 노드를 탐색해버리면, 이는 원래 Heap의 데이터 삽입 시간 복잡도인 O(log n)이 나오지 않을 것이라고 생각했다. 이를 해결하기 위해 노드의 Index를 2진수로 나타내어 보았더니 일정한 규칙을 찾을 수 있었다. 2진수로 나타낸 길이는 탐색 깊이를 나타내고, 맨 앞 비트를 제외한 다음 비트부터, 0이라면 LeftChild를, 1이라면 RightChild를 가져온다면 원하는 위치를 찾을 수 있다. 이를 통해 Heap의 원래 시간복잡도대로 구현할 수 있었다.