Data Structure Project #2

학과: 컴퓨터정보공학부

학번: 2022202065

이름: 박나림

1. Introduction

도서 대출 관리 프로그램을 구현하는 프로젝트로, B+-Tree와 선택 트리(Selection Tree), Heap 자료구조를 이용한다. B+-Tree로 현재 대출 중인 도서를 저장하고, 일정 이상 수의 대출로 대출 불가 상태가 된 도서들은 B+-Tree에서 삭제된 후 선택 트리와 Heap에 저장된다. command.txt 파일에 저장되어 있는 명령어들을 차례대로 읽어오면서 그에 맞는 명령들을 수행하는 방식이다.

'LOAD'명령 시, 'loan_book.txt' 파일에 저장되어 있는 도서 정보들을 불러와서 B+-Tree에 데이터를 저장한다. 도서 정보들은 도서명(name), 도서분류코드(code), 저자(author), 발행연도(year), 대출 권수(loan count)로 탭으로 구분되어 있다. B+-Tree의 차수는 3으로 구성되며, 각 도서들은 LoanBookData Class로 선언된다. LOAD 명령의 성공 여부에 따라 결과를 출력한다.

'ADD'명령 시, 4개의 도서 정보 인자를 추가로 입력 받아서 B+-Tree에 직접 추가한다. 이때 중복된 도서가 들어올 시 새 노드를 생성하지 않고 대출 권수만 증가시키도록 한다. 대출 권수는 0부터 시작하여 각각의 대출 불가 도서는 분류 코드 중 000~200번이 3, 300~400번이 4, 500~700번이 2가 되면 B+-Tree에서 삭제되고 선택 트리와 Heap으로 보내진다. ADD 명령의 성공 여부에 따라 결과를 출력한다.

'SEARCH_BP'명령 시, 추가 인자로 1개가 들어오면 해당 도서 이름으로 검색한 결과를 출력한다. 2개가 들어오면 해당 알파벳으로 시작점과 끝점까지의 도서들을 모두 검색하여 출력한다. SEARCH_BP 명령의 성공 여부에 따라 결과를 출력한다.

'PRINT_BP'명령 시, B+-Tree에 저장되어 있던 데이터들을 도서명을 기준으로 오름차순으로 하여 차례대로 출력한다. PRINT_BP 명령의 성공 여부에 따라 결과를 출력한다.

'PRINT_ST'명령 시, 추가 인자로 도서 분류 코드가 들어오면 Selection Tree에서 해당하는 데이터 정보들을 오름차순으로 차례대로 출력한다. PRINT_ST 명령의 성공 여부에 따라 결과를 출력한다.

'DELETE'명령 시, Selection Tree에서 root node에 저장된 도서 정보를 제거한다. 해당 도서가 제거되어서 Heap에 저장된 도서들의 대소 관계가 변경된다면 Heap을 재정렬해야 한다. DELETE 명령의 성공 여부에 따라 결과를 출력한다.

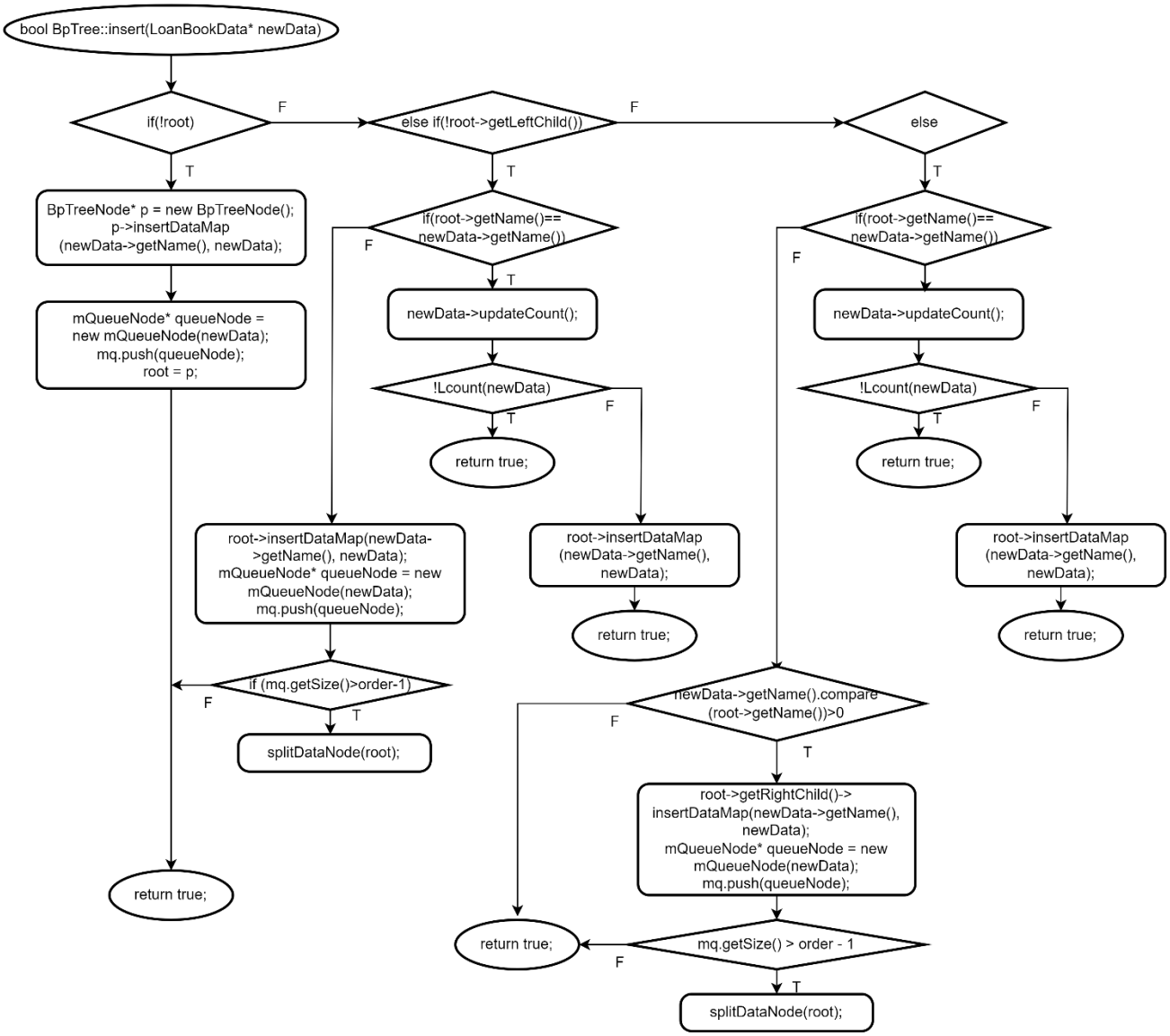
'EXIT'명령 시, 프로그램 상의 메모리들을 해제하고 프로그램을 종료시킨다. 이때도 마찬가지로 성공하면 결과를 출력한다.

각각의 명령어들은 인자 불일치, 다른 예외 처리, 저장된 데이터가 없는 등의 상황에서는 해당 명령어의 에러코드에 맞는 오류 결과문을 출력한다. 또한 이러한 출력 결과들은 log.txt에 저장되며, 이미 파일이 존재할 경우 가장 뒤에 이어서 저장하는 것으로 한다.

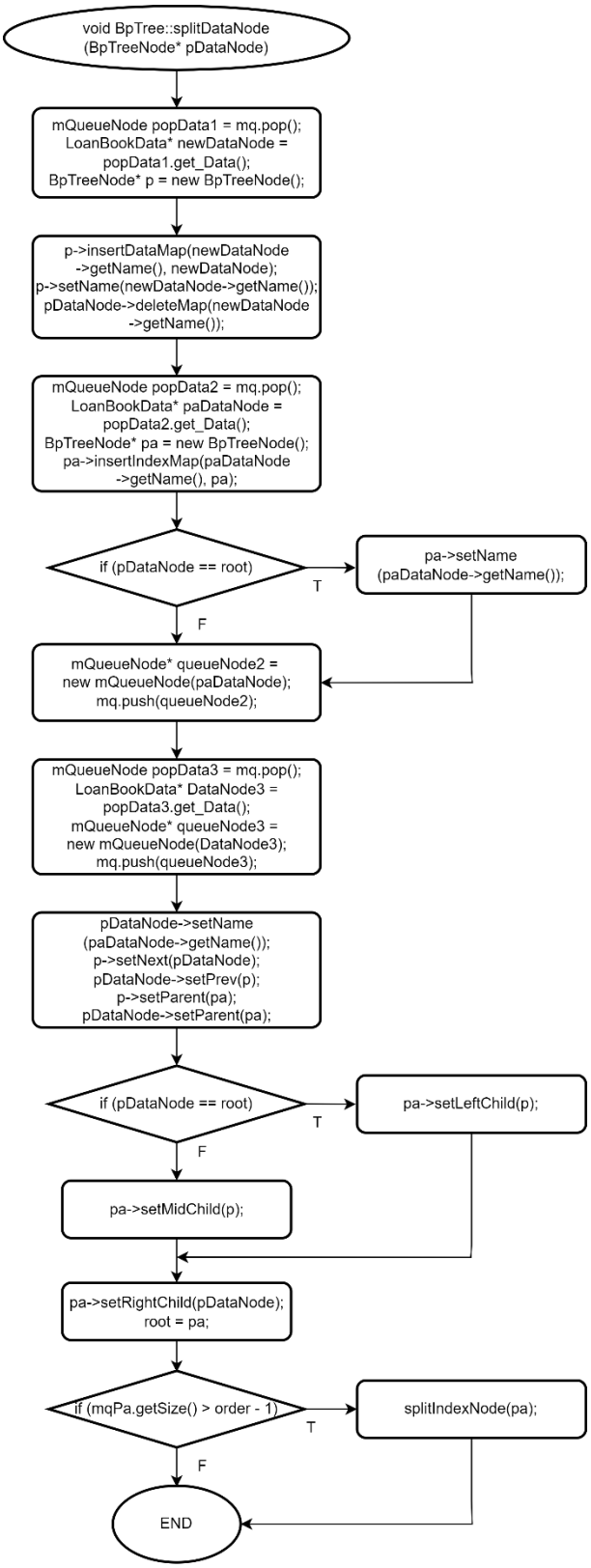
2. Flowchart

1) B+-Tree

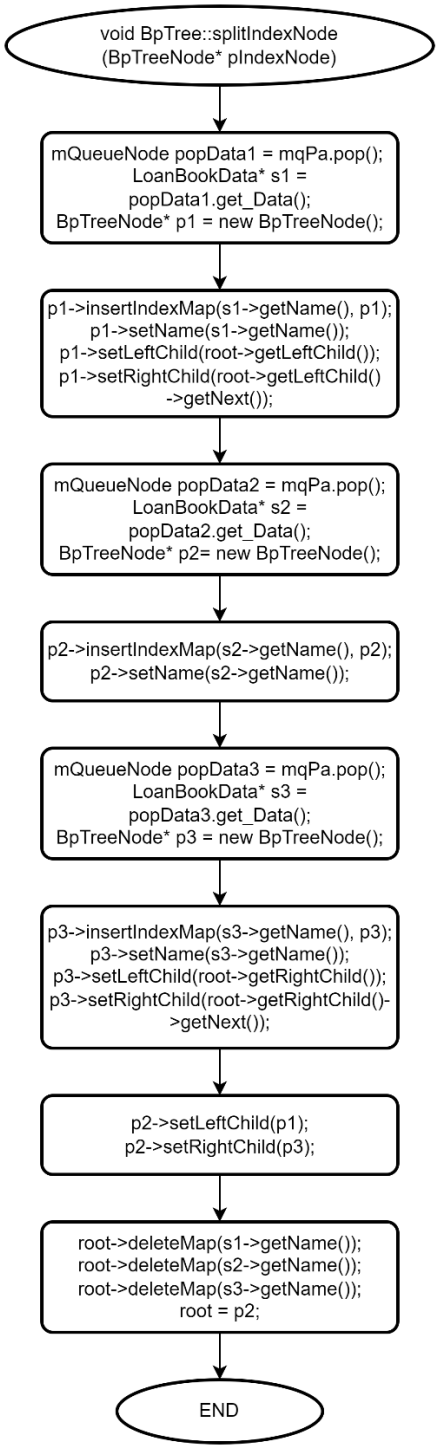
-insert



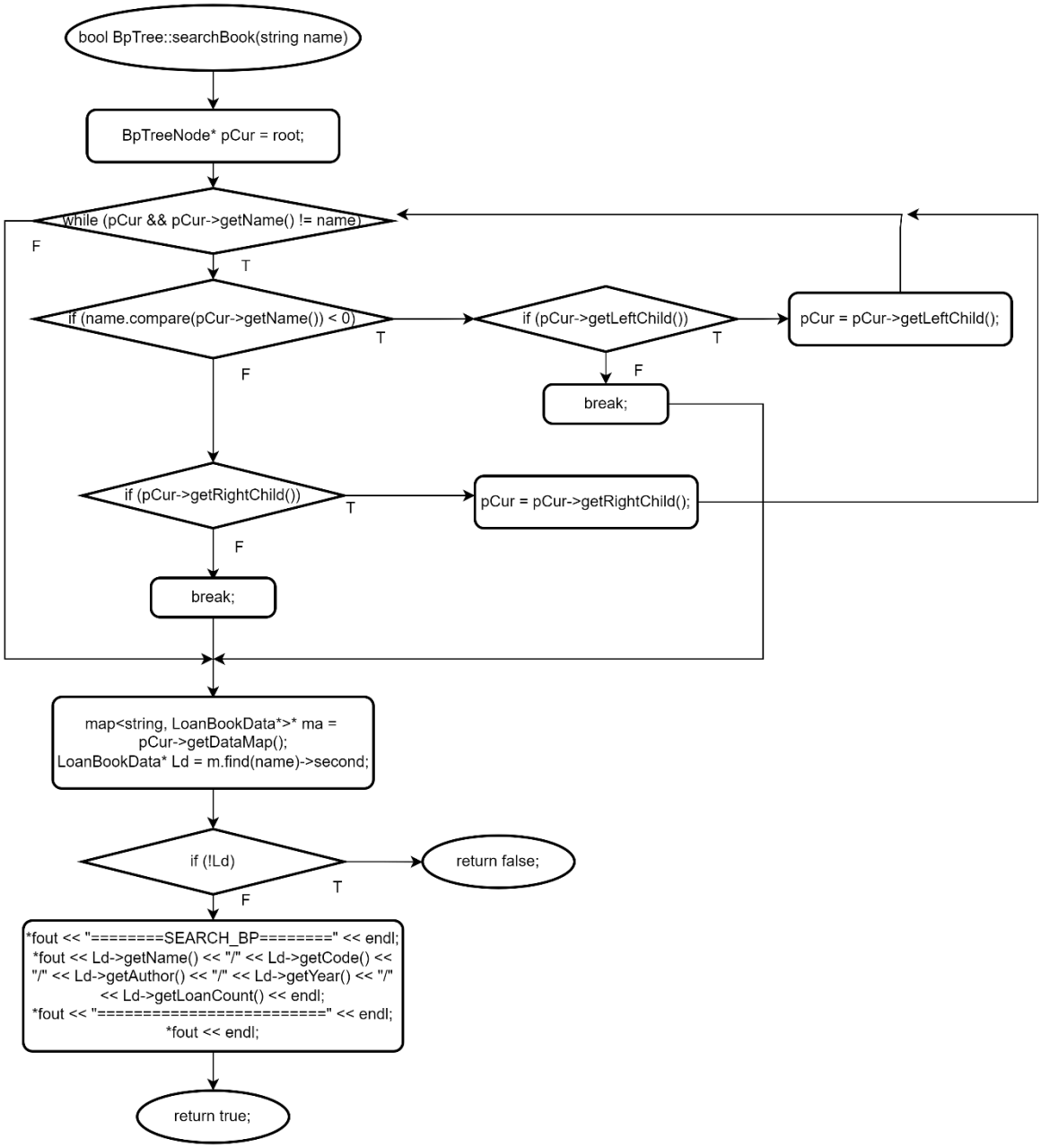
-splitDataNode



-splitIndexNode

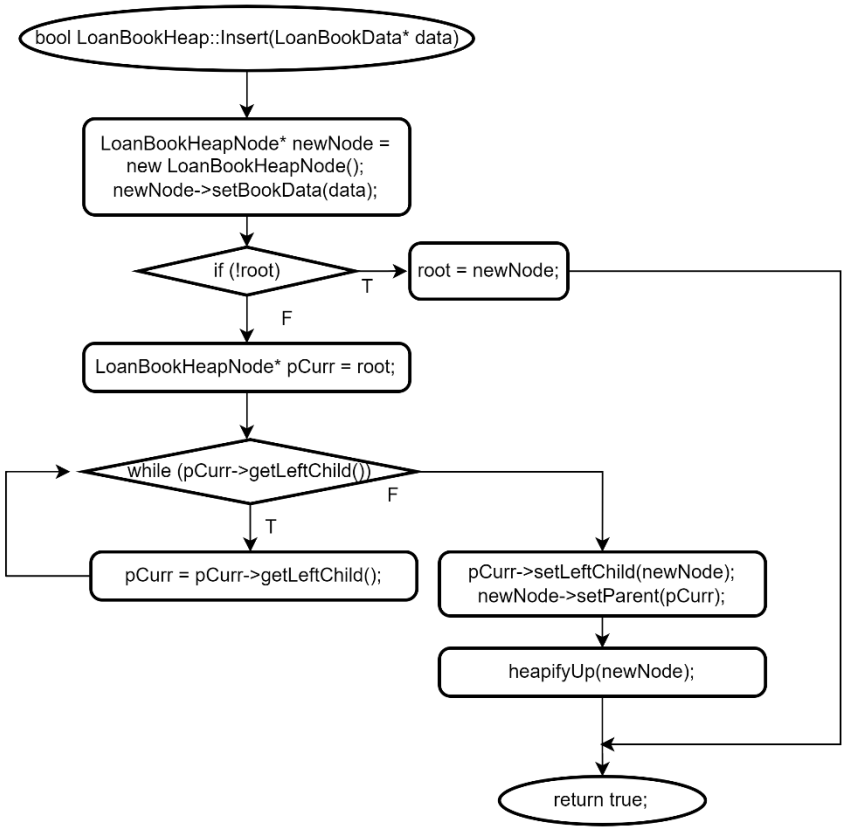


-search

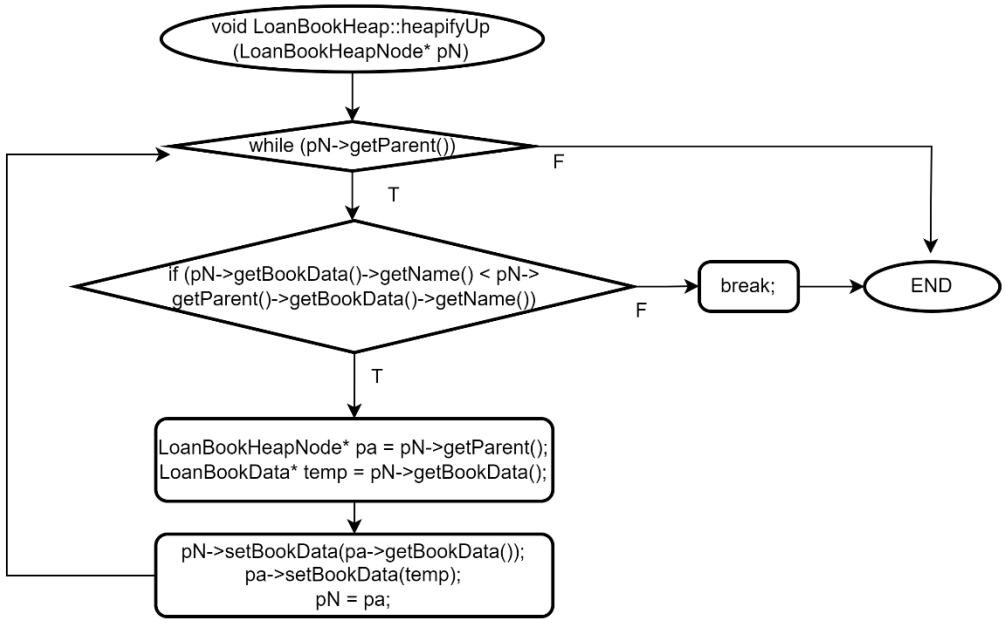

bool BpTree::searchBook(string name)
BpTreeNode* pCur = root;
while (pCur && pCur->getName() != name)
F
T
if (name.compare(pCur->getName()) < 0)
T
if (pCur->getLeftChild())
T
pCur = pCur->getLeftChild();
F
break;
F
if (pCur->getRightChild())
T
pCur = pCur->getRightChild();
F
break;
map<string, LoanBookData*>* ma =
pCur->getDataMap();
LoanBookData* Ld = m.find(name)->second;
if (!Ld)
T
return false;
F
*fout << "========SEARCH_BP========" << endl;
*fout << Ld->getName() << "/" << Ld->getCode() <<
"/" << Ld->getAuthor() << "/" << Ld->getYear() << "/"
<< Ld->getLoanCount() << endl;
*fout << "==========================" << endl;
*fout << endl;
return true;

2) Heap

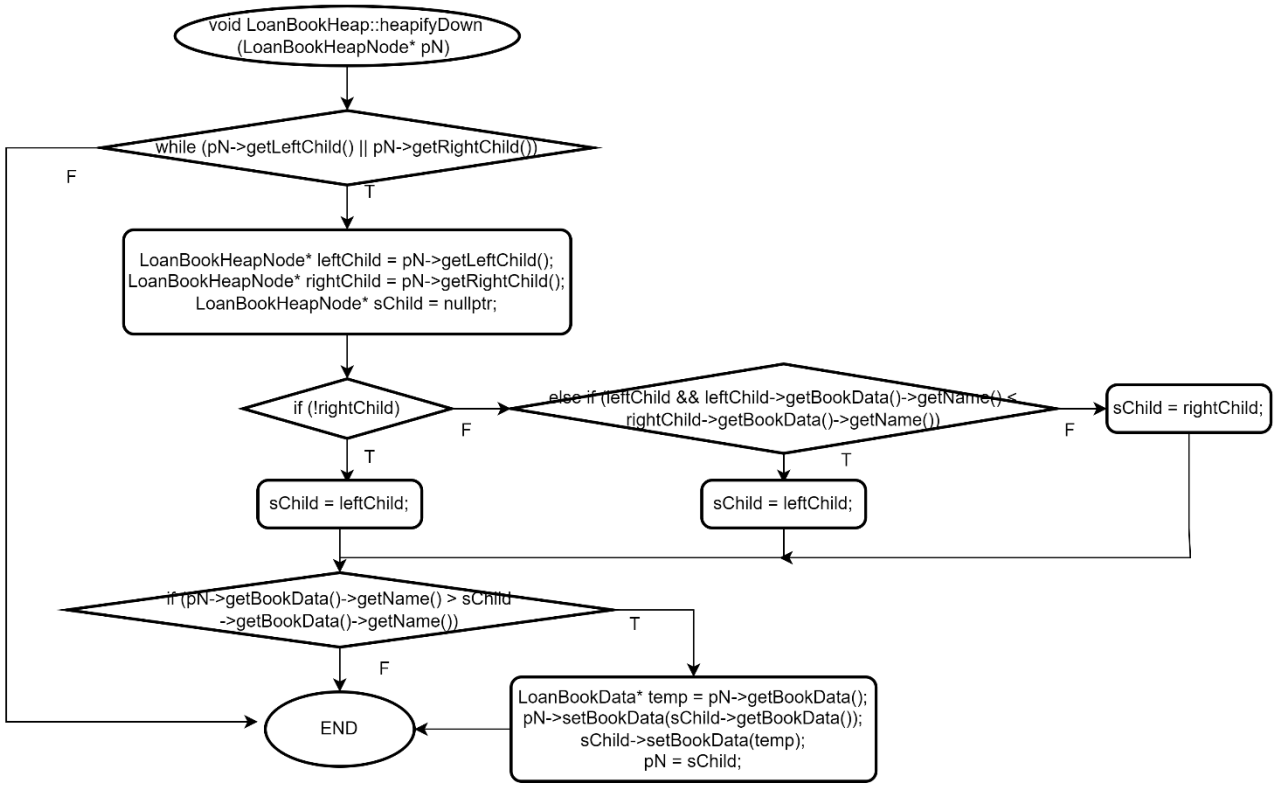
-insert



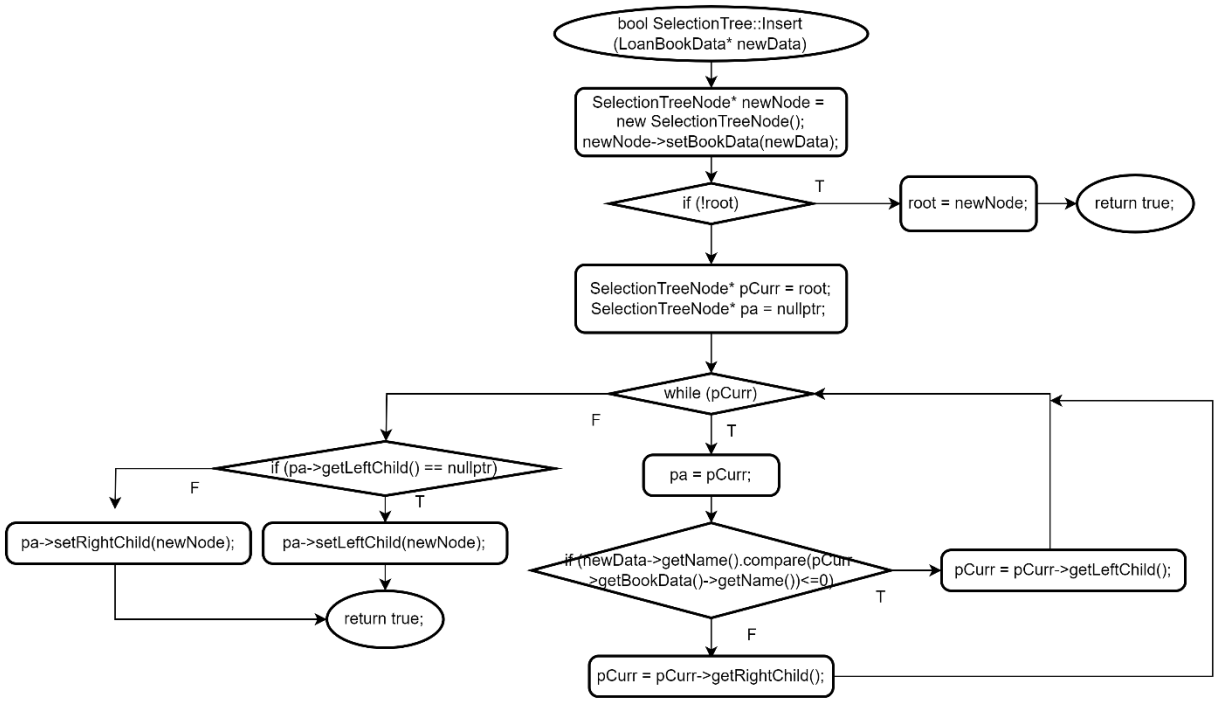
-heapifyUp



-heapifyDown



3) SelectionTree



3. Algorithm

1) B+-Tree

-Insert

BpTree의 insert 알고리즘으로, 먼저 root==nullptr인 첫번째 노드인 경우 BpTreeNode* 타입으로 newNode를 생성한다. 그리고 DataMap으로 인자로 받아온 LookBookData의 이름과 데이터 정보들로 키와 요소들을 pair로 저장한다. 또한 각 노드의 현재 map의 개수에 따라 분할을 파악하기위해 queue 자료구조를 사용하여 저장하였다. 이러한 노드를 root로 설정한 뒤 true를 반환한다.

두번째와 세번째로 들어오는 데이터의 경우, root의 이름과 newData의 이름을 비교하여 만약 같은 도서일 시 해당 노드의 LoanDataCount만 업데이트시키고 기존에 map에 저장되어있던 데이터와 교체하는 식으로 진행한다. 또한 Lcount함수로 해당 도서 코드에 따라 대출 불가 도서인 경우를 switch문으로 확인한다. 대출 불가 도서인 경우 각 코드마다 생성한 heap 객체의 insert 함수로 해당 도서를 전달시키고 selectionTree에도 추가한 뒤 기존의 데이터는 map에서 삭제시키는 것으로 한다.

다른 이름인 경우, root 노드에 이어서 DataMap으로 newData를 추가시킨다. 이때 Map은 자료구조에 따라 자동 정렬되므로, 따로 사전순으로 정렬할 필요없이 바로 추가하는 작업만 진행한다. 또한 queue에도 push를 하여 getSize 함수를 통해 그때마다 각 노드의 DataMap 개수를 확인한다. 이번 프로젝트에서는 3차 B+-Tree로 진행하므로, order를 3으로 받아서 3이상인 경우에 splitDataNode 함수를 호출하게 된다.

이러한 B+-Tree의 높이가 1이상인 경우, 즉 root의 leftChild 또는 rightChild가 존재하는 경우에도 먼저 해당 노드의 이름이 newData의 이름과 일치하는지 확인한다. 같다면 이전과 같은 방식으로 Count만 증가시키고 Lcount함수를 통해 대출 불가 도서 여부를 확인한다. 다른 이름인 경우에는 이름을 비교하여 사전 순으로 leftChild 또는 rightChild쪽으로 끝까지 이동하여 맞는 DataNode에 map으로 추가하고 queue에도 push한다. 마찬가지로 queue의 크기를 확인하여 3이상이 되면 splitData 함수를 호출한다.

-Split

queue에 저장되어 있던 데이터들을 차례대로 하나씩 pop하여 각각 popData1, popData2, popData3로 저장을 한 뒤 분할을 시작한다. BpTreeNode* p를 새로 만들어서 popData1의 데이터들을 mapData로 저장한다. 이때 queue는 선입선출 구조이므로, 3개의 데이터 중 첫번째 데이터가 p에 저장되게 된다. 인자로 받아서 분할을 하게 되는 pDataNode는 나머지 2, 3번째 데이터만 저장될 수 있도록 첫번째 데이터의 이름을 이용하여

deleteMap으로 삭제시킨다. 또한 popData2는 새로 BpTreeNode* pa를 만들어서 parent node로 저장한다. popData3이 필요한 이유는, queue가 선입선출 구조이다 보니 두번째 데이터가 부모 노드로 복사하기 위해 pop했다가 다시 push를 하면 두번째와 세번째의 데이터 순서가 바뀐다. 따라서 다시 순서에 맞게 정렬해주기 위하여 세번째 데이터까지 pop한 뒤 다시 두번째 데이터부터 push해서 저장해주는 것이다.

queue 정렬까지 끝나면 마지막으로 p와 pa를 연결리스트와 B+-Tree로 연결시킨다. p의 setNext는 pDataNode로 하고, setParent는 pa로 한다. 또한 pDataNode가 root인지 아닌지에 따라 p는 pa의 leftChild 또는 MidChild가 되는 것으로 한다. 이때 부모 노드가 되는 pa의 data수도 mqPa queue로 따로 저장하여서, 이 역시 개수가 3이상이 되면 indexNode를 분할한다. indexNode는 DataNode의 분할과 비슷한 알고리즘으로 구성되지만, 중간 노드가 중복되지 않고 바로 부모 노드로 올라간다는 것에 차이점이 있다.

-Search

현재 노드인 pCur을 생성하여 root부터 시작해 이름 비교를 하여 leftChild 또는 rightChild로 내려가면서 검사를 진행한다. 끝까지 내려가서 해당 노드를 찾으면, 노드에 저장되어 있는 DataMap의 데이터 중에서 찾기 위해 map의 find 함수를 사용한다. 이때 데이터가 없다면 false를 return하고, 있다면 btree에서 인자로 받았던 fout 객체를 이용하여 log.txt에 탐색한 도서 정보를 출력한다.

2) Heap

ADD 명령을 통해 입력된 도서들 중 중복된 도서들이 대출 불가 도서 상태가 되면 각각 도서 분류 코드에 따른 Heap 객체들로 insert하게 된다. 이때 첫번째 노드는 root로 설정하며, 두번째 부터는 pCurr을 생성하여 왼쪽 자식 끝까지 내려가도록 한다. 마지막 위치에 전달받은 newNode를 삽입하며, 마지막 위치부터 시작하여 사전 순에 따라 노드를 올리는 방식으로 heap을 정렬하게 된다. Min Heap이기 때문에 부모 노드보다 이름이 더 작은 경우에 부모 노드와 위치를 교환하는 식으로 한다. 필요에 따라 아래로 내리는 경우에는 leftChild와 rightChild중 더 작은 노드를 sNode로 선택하여 그와 교환하는 식으로 한다.

3) Selection Tree

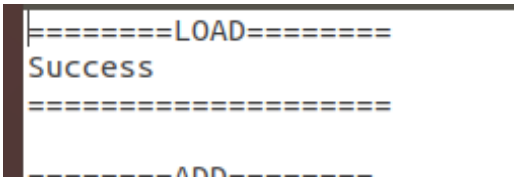
각 run들을 도서 분류 코드 100부터 700까지의 heap으로 구성한다. 이들의 root부터 시작하여 도서 이름을 사전순으로 비교하여 정렬하는 방식으로 한다. Min Winner Tree이기

때문에 부모 노드보다 더 작은 이름일 경우에 위로 올라가는 것으로 하며, 처음에 root는 newNode로 설정된다. 두번째 노드부터 이름을 비교하여 leftChild와 rightChild를 설정해서 Tree를 구성한다.

4. Result Screen

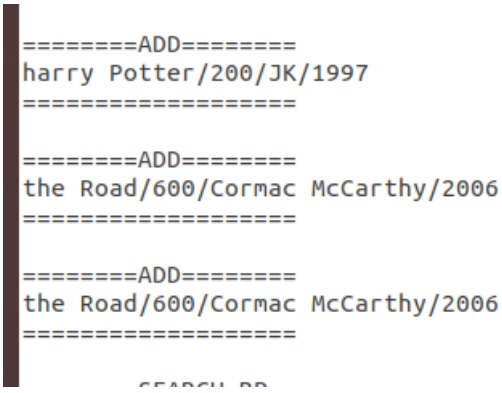
1) LOAD

loan_book.txt에 있는 데이터의 정보들을 불러와서 차례대로 B+-Tree에 저장한 뒤 결과를 출력한 모습이다. tap으로 구분되어 있는 도서 정보들을 각각 name, code, author, year, loan_count로 LoanBookData 노드에 저장하여 bptree insert로 전달하여 추가한 것이다. 이때 각 도서들은 중복이 없으므로 모든 도서들의 loanCount는 0이 된다. 따라서 이때까지는 heap과 selectionTree에는 도서들이 없고 B+-Tree에만 도서들이 저장된다.



2) ADD

B+-Tree에 직접 도서를 추가하기 위하여 command.txt에서 tap으로 구분되어 있는 도서 정보들을 각각 name, code, author, year로 LoanBookData 노드에 저장하여 bptree insert로 전달하여 추가한 것이다. 이때부터는 중복되는 도서들이 있을 시 loanCount를 증가시켜서 각 도서 분류코드에 맞는 대출 불가 도서가 발생할 시 heap과 selectionTree에 전달되게 된다. 아래는 ADD 명령을 통해 추가된 도서들의 결과를 출력한 모습이다.



3) SEARCH_BP

1개의 인자로 도서 이름을 입력했을 때, 해당 도서를 B+-Tree에서 탐색하여 도서가 존재할 시 해당 도서의 정보를 출력한다. 아래는 검색한 도서의 정보를 출력한 모습이다.

```
========SEARCH_BP========
dune/100/Frank Herbert/1965/0
=========================
```

범위로 검색하는 경우는 구현하지 못하여 ERROR로 뜬다.

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

4) PRINT_BP / PRINT_ST / DELETE

마찬가지로 나머지 명령어들은 구현하지 못한 미완성 상태로, 각각 명령어 종류에 따라 에러 코드가 출력된다.

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

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

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

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

5) EXIT

마지막으로 EXIT 명령 시 프로그램의 메모리를 해제하며 종료하는 것으로 결과가 출력된 모습이다.

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

5. Consideration

이번 프로젝트를 진행할 때 가장 시행착오를 겪었던 문제는 map 자료구조를 사용하는 것이었다. map의 데이터들을 저장하고 다시 그 데이터에 접근하는 과정에서 여러 오류들이 발생하였었다. 대체로 this가 nullptr이라는 메모리 접근 문제, 잘못된 주소 등의 문제였다. 이에 각각 map들의 데이터들을 출력해보기도 하였으나 전부 0으로 출력이 되었다.

그래서 다른 방법들을 시도해보다가, queue 자료구조를 사용하는 방법을 생각해보게 되었다. queue에 각각 데이터들을 저장했다가 필요에 따라 pop하여 데이터를 조회하는 방법이었다. 이에 map 컨테이너를 구성하기 위해 map에도 저장하고 따로 queue에도 저장하여 코드를 설계하는 방식을 사용하였다. 하지만 이런 방법으로 하니 B+-Tree의 insert, split 부분이 상당히 복잡해졌다. 이 부분은 map 자료구조에 대해 더 공부를 해야 간결하게 짤 수 있을 것 같다고 느꼈다.

B+-Tree insert 알고리즘 자체에도 여러 고민을 하였다. 일반적인 B Tree같은 경우도 배열로 구성하여 해당 index에 삽입하는 방식으로 공부했었는데, 이번에는 포인터를 통해 바로 tree형태로 구현하는 방식이어서 어려웠던 것 같다. 또한 3차 B+-Tree이다 보니 도서 이름에 따라 삽입될 때 왼쪽, 오른쪽 자식 이외에 중간 자식으로 삽입되는 경우도 있어서 고민을 했었다.

제일 생각이 많이 필요했던 건 B+-Tree의 삽입 방식이 아래에서부터 진행하여 tree를 반대로 구성한다는 점이었다. 그래서 이번에 프로젝트를 진행하면서 코드를 간결하게 구성하지 못하고 경우에 따라 나누어서 각각 구성한 점이 아쉬웠다. 그래서 search를 할 때도 해당 도서만 검색하는 건 진행하였으나 범위 탐색부터 다른 print나 delete명령 같은 경우는 구현을 하지 못해 더 아쉬웠던 것 같다. 이 점도 B+-Tree의 알고리즘을 더 공부해봐야 할 것 같다.