데이터 구조 설계 보고서

제목: DS_Project2

학　　과: 컴퓨터정보공학부

학　　번: 2022202064

성　　명: 최봉규

실습 분반: 금 5교시, 6교시

1. Introduction

본 프로젝트에서는 B+-Tree와 selection tree, heap을 이용하여 도서 대출 관리 프로그램을 구현한다. 대출 관리 프로그램은 도서명, 도서 분류 코드, 저자, 발행 연도, 대출 권수를 관리하며, 이를 이용해 대출 중인 도서와 대출 불가 도서에 대한 정보를 제공한다. B+-Tree를 이용하여 대출 중인 도서를 관리하며, 선택 트리를 이용하여 대출 불가 도서를 관리한다. 모든 명령어는 command.txt에 저장하여 순차적으로 읽고 처리한다. 모든 명령어를 입력할 때, 공백 문자가 필요한 경우 '₩t'을 사용한다. 예시는 아래와 같다.

- SEARCH_BP<₩t>aaaa<₩t>eee
- ADD<₩t>book<₩t>200<₩t>risa<₩t>1990

1) 도서 관련 정보 데이터

프로그램은 도서명(name), 도서 분류 코드(code), 저자(author), 발행 연도(year), 대출 권수(loan count) 정보가 저장된 파일 loan_book.txt를 LOAD 명령어를 통해 읽어 해당 정보를 B+-tree에 저장한다. loan_book.txt는 도서에 대한 정보가 '₩t'으로 구분되어 저장된다. 이때 도서명의 중복이 없다.

도서 분류 코드는 000 ~ 700 번대까지 있으며, 각 분류 코드에 따른 대출 가능 권수는 다음과 같다.

도서 분류 코드	대출 가능 권수
000	3
100	3
200	3
300	4
400	4
500	2
600	2
700	2

2) B+-Tree

B+-tree의 차수는 3이다.

주어진 loan_book.txt에 저장된 데이터를 읽은 후, 대출 중인 도서는 B+-tree에 저장한다. ADD 명령어로 추가되는 데이터를 읽은 후, B+-tree에 저장한

다. B+-tree에 없으면 node를 새로 추가하며, 존재하는 경우 대출 권수만 증가시킨다. 이후 대출 불가 도서(도서가 모두 대출된 경우)는 selection tree에 해당 도서를 전달하고, B+-tree에서 제거한다.

B+-tree에 저장되는 데이터는 LoanBookData class로, 멤버 변수로는 도서명, 도서 분류 코드, 저자, 발행, 연도, 대출 권수가 있다. B+-tree는 도서명을 기준으로하며, 모두 소문자와 숫자로 대문자는 고려하지 않는다.

B+-tree의 데이터 노드는 도서명, 도서 분류 코드, 저자, 발행 연도, 대출 권수를 저장한 LoanBookData를 map 컨테이너 형태로 가지고 있다.

3) Selection Tree

Selection tree는 도서명을 기준으로 Min Winner Tree를 구성하며, 이 역시 소문자와 숫자로 대문자는 고려하지 않는다. Selection Tree의 run의 개수는 도서 분류 코드 개수인 000 ~ 700으로 8개로 동일하다.

ADD 명령어로 B+- tree에 저장된 데이터의 대출 권수를 업데이트 후에, 대출 불가 도서는 도서 분류 코드에 따라 Min Heap으로 구현된 Selection Tree의 각 runs에 저장한다. 이때, 대출 불가한 도서는 B+-tree에서 삭제한다.

Selection Tree의 내부 node에 저장되는 데이터는 LoanBookData class로 선언되어 있으며, 멤버 변수로는 도서명, 도서 분류 코드 저자 발행 연도, 대출 권수를 갖고 있다. Selection Tree의 각 run은 LoanBookHeap class로 선언되며, Heap이 정렬되는 경우 Selection Tree로 같이 재정렬 된다.

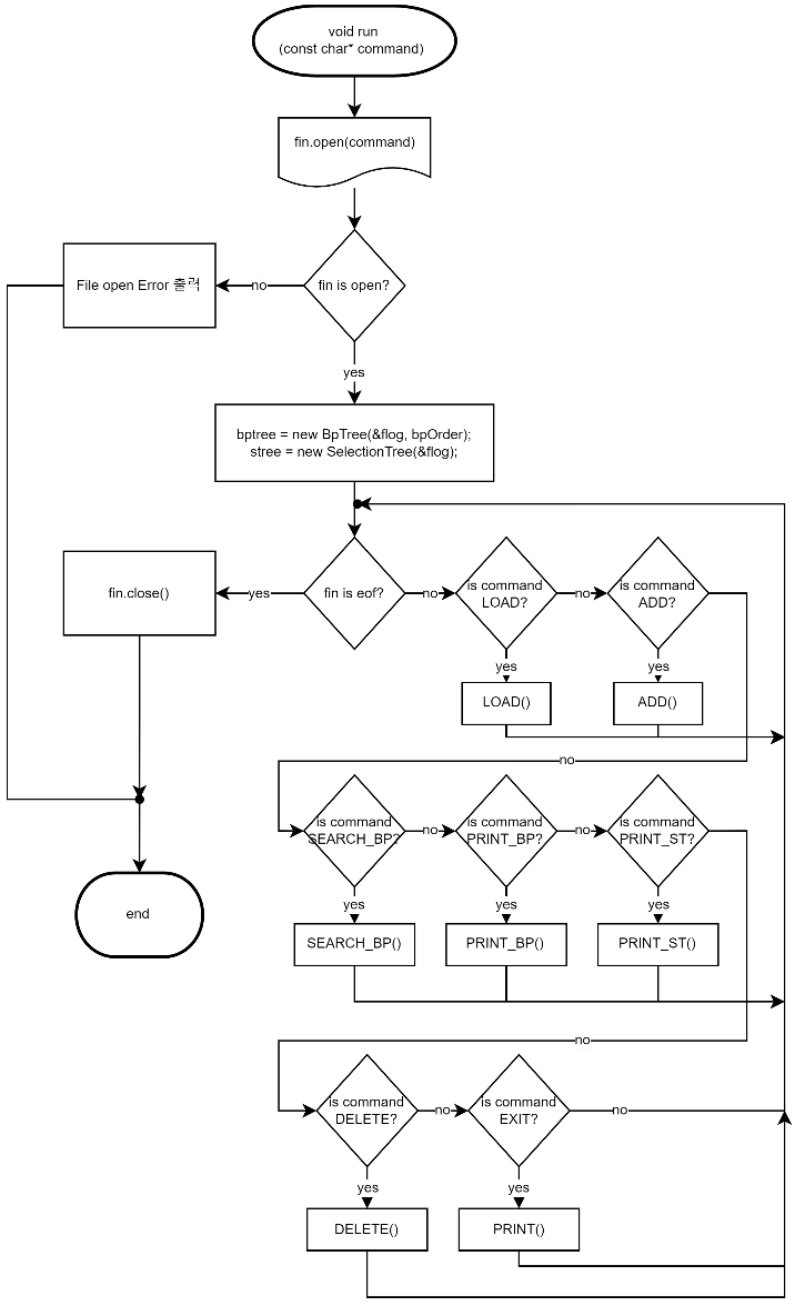
4) LoanBookHeap

LoanBookHeap은 Min Heap으로 이 역시 도서명을 기준으로 정렬된다.
ADD명령어를 통해 B+-tree에 저장된 데이터 중 대출 불가 도서를 받아와 Heap을 구축하며, 기존에 Heap이 존재하지 않는 경우 새로 구축한다. Heap에 새로운 데이터가 추가될 때는 왼쪽 자식 노드부터 생성한다. (이는 Heap을 단순히 array로 구현하는 것이 아닌 linked list로 구현함에 있어 생성된 조건이다.)

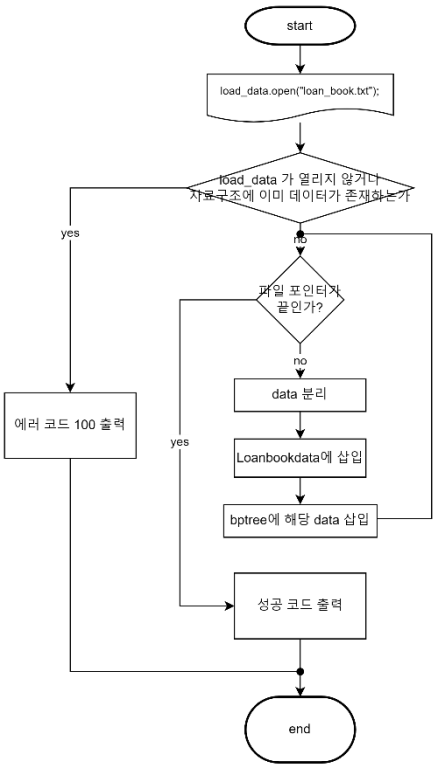
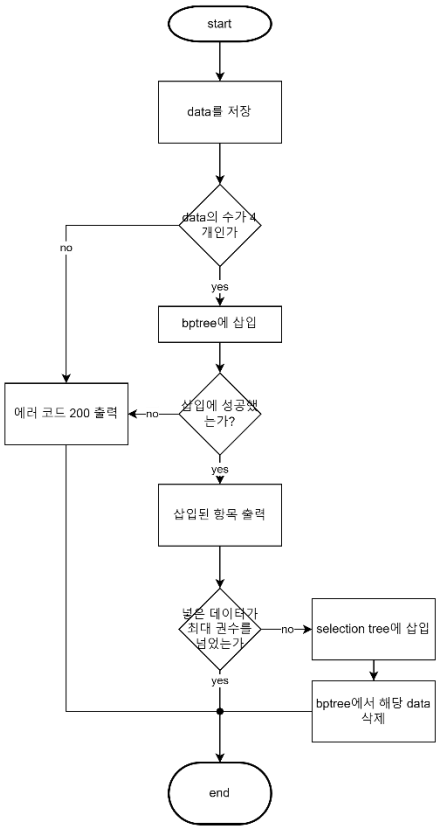
Heap을 재정렬할 때, 모든 부모 노드의 도서명이 자식 노드의 도서명보다 작거나 같은 경우 정렬을 완료한다. 이때 Heap이 재정렬 되는 경우 Selection Tree역시 도서명을 기준으로 재정렬하도록 한다.

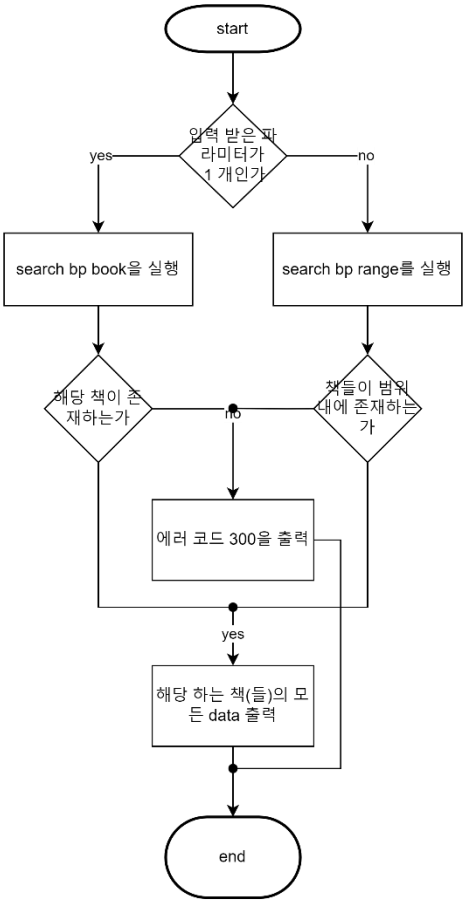
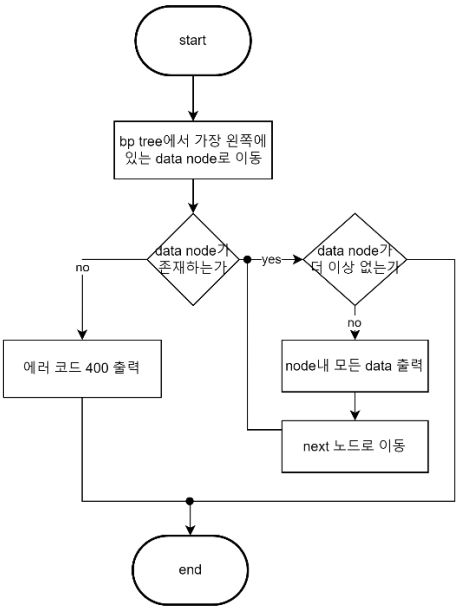
LoanBookHeap에 저장되는 데이터는 LoanBookData class로 선언되어 있으며, 멤버 변수는 도서명, 도서 분류 코드, 저자, 발행 연도, 대출 권수를 갖고 있다.

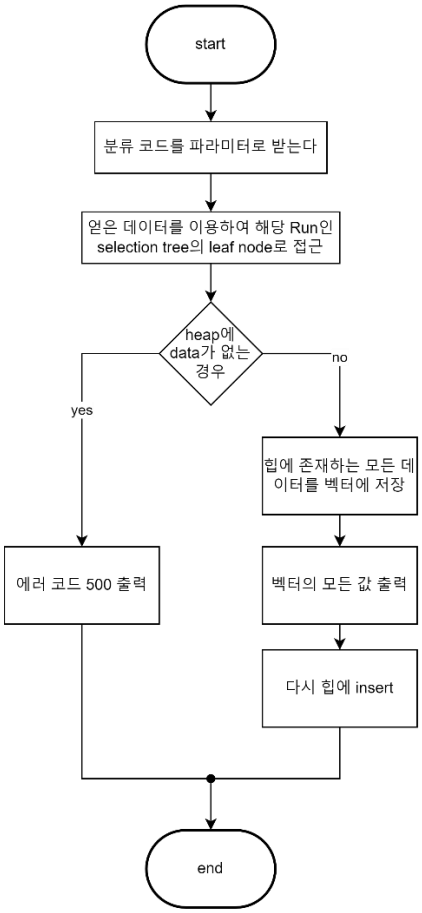
2. Flowchart



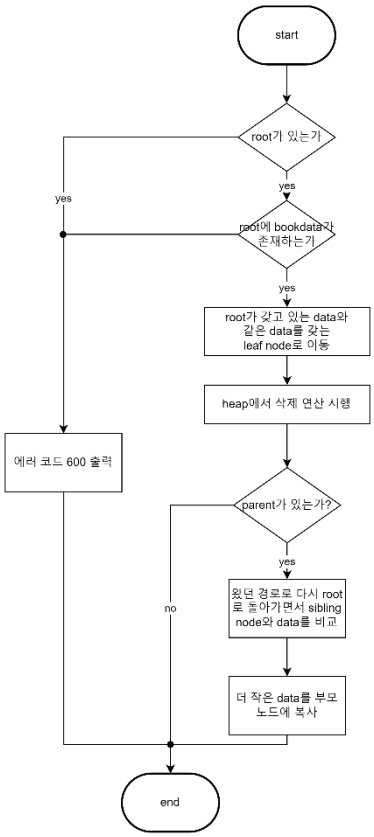
해당 flow chart는 프로젝트를 전체적으로 구성한 내용으로 각 명령어를 입력 받을 때 어떤 식으로 처리되는 과정을 보여주고 있다. 프로그램이 시행하고 프로그램이 ifstream으로 받은 데이터를 대략적으로 어떤 함수를 통해 처리되는가에 대해 보여주며, 모든 데이터를 다 받고 나서 종료되는 모습을 보여준다.

	해당 플로우 차트는 LOAD 실행시 나오는 차트로 조건에 부합하지 않은 경우 에러 코드 100을 출력, 조건에 부합하는 경우 data를 bptree에 삽입하고 모든 삽입이 종료된 후 성공 코드를 출력한다.
	해당 플로우 차트는 ADD 커맨드를 입력시 옆과 같이 동작한다. 먼저 데이터를 loanbookdata에 저장하는데 인자가 부족한 경우, 에러 코드 200을 출력한다. 그렇지 않은 경우에도 bptree에 삽입에 실패하는 경우에도 에러코드를 200 출력한다. 인자가 정확히 4개 있는 경우 분류 코드에 따른 최대 권수를 파악하여 selection tree에 삽입할지 결정한다.

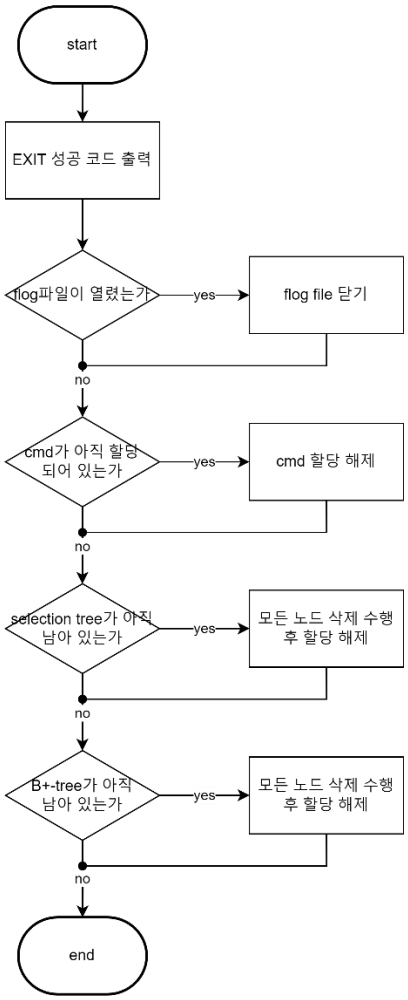
<table>
<tr>
<td> 
</td>
<td>해당 플로우 차트는 SEARCH_BP 커맨드를 입력시 옆과 같이 동작한다. 데이터를 입력 받을 때 입력 받은 데이터 수의 차이를 둔다. 만약 입력 받은 데이터가 1인 경우, 입력 받은 데이터의 책을 검색한다. 그 외에는 2인 경우로 범위 검색을 시행한다. 조건에 부합하는 경우 책(들)의 모든 data(제목, 코드, 작가, 출판 년도, 대출 권수에 대한 정보)를 출력한다. 아닌 경우는 에러 코드 300을 출력한다.</td>
</tr>
<tr>
<td> 
</td>
<td>해당 플로우 차트는 PRINT_BP 커맨드 입력시 다음과 같이 동작한다. 가장 왼쪽에 있는 data node로 이동하면서 data가 존재하지 않는 경우 에러 코드를 400 출력한다. Data node가 존재하면, 노드가 갖고 있는 data를 모두 출력하고 링크드 리스트로 연결된 다음 노드에 접근하여 출력한다.</td>
</tr>
</table>



해당 flow chart는 PRINT_ST 커맨드 입력 시 작동하는 것으로, 파라미터로 분류코드를 받고 이를 바탕으로 leaf node로 접근한다. leaf node를 제외한 다른 노드에는 LoanbookData만 가지고 있으며 leafnode는 data와 heap에 대한 정보를 담고 있다. Heap에 data가 없는 경우 에러 코드 500을 출력한다. Data가 존재하는 경우 모든 값들을 삭제하면서 벡터에 저장한다. 삭제할 때는 min heap의 특성으로 자동으로 정렬되어서 나오므로 벡터는 특별한 sort의 과정 없이 모든 값을 출력하고, 이를 다시 힙에 넣는 과정을 갖는다.



해당 flow chart는 DELETE 커맨드 입력 시 작동하는 것으로, root가 없는 경우 에러 코드 600을 출력한다. 또한 root내에 data가 없는 경우에도 마찬가지이다. 이는 data를 삭제한 후에는 무조건 하나 이상의 data가 남는 다면 root로 데이터가 복사되기 때문이다. 조건에 부합하는 경우 root가 갖고 있는 data와 같은 data를 갖는 leaf node로 이동한다. 이는 leaf node를 제외한 다른 노드에서는 data를 복사할 뿐이지만 leaf node에서만 heap을 접근할 수 있기 때문이다. heap에서 삭제연산을 시행 후, 셀력션 트리를 재정렬하면서 root까지 가는 경로에 data를 복사하면서 진행한다.



해당 flow chart는 EXIT 커맨드 입력 시 수행하는 것으로 먼저 EXIT 성공 코드를 출력한다. 그리고 열려있는 파일은 닫고 동적할당 것들은 동적할당을 해제하는 작업을 진행한다.

3. Algorithm

manager.cpp

해당 파일에서는 입력 커맨드에 따른 명령어를 수행하게 된다. 각 커맨드가 저장되어 있는 파일을 읽어와 해당 파일에 적힌 명령어를 수행하게 되는데, 해당 파일 안에는 LOAD, ADD, SEARCH_BP, PRINT_BP, PRINT_ST, DELETE, EXIT에 관한 명령어가 존재한다. 각 명령어에 따른 함수를 실행한다.

LOAD의 경우에는 각 구별자가 '₩t'으로 구분되면서 제목, 코드, 작가, 출판 년도, 대출 가능 권수에 대한 정보가 B+-tree로 해당 정보를 전달하면서 B+-tree를 구축하도록 한다. 다만 log.txt파일이 열리지 않거나 B+-tree나 selection tree 내에서 data가 존재하는 경우 에러코드 100을 출력하도록 한다.

ADD의 경우 LOAD와 비슷하게 제목, 코드, 작가, 출판 년도 이 4개에 대한 정보를 받는다. 하지만 정보의 개수가 하나 이상 부족한 경우에 에러코드 200을 출력하도록 한다. 해당 정보 역시 '₩t'으로 구분되어 있다. B+-tree에 해당 데이터를 입력하도록 하는데, 해당 책의 정보를 책의 코드와 코드에 따른 대출 가능 최대 권수를 이용하여 더 이상 빌릴 수 없는 책인 경우 selection tree에 해당 데이터를 삽입하게 된다. 삽입에 성공한 경우 success code로 해당 책의 제목, 저자, 출판 년도, 코드를 출력하도록 한다.

SEARCH_BP에 대해서는 해당 커맨드에 있는 인자의 개수를 파악한다. 인자의 개수를 파악하는데에도 구분자 '₩t'을 이용하여 구분하게 된다. 파악된 인자의 개수가 1개인 경우 해당 함수는 search_bp_book으로 이동하여 해당 인자를 제목으로 하는 책의 데이터를 찾아 출력하도록 한다. 이는 bptree에 있는 SEARCH_BP_BOOK을 이용하여 출력하도록 한다. 만약 해당 인자로 하는 제목의 책이 없는 경우 오류 코드 300을 출력하도록 한다. 인자의 개수가 2개인 경우 해당 함수는 search_bp_range로 책의 이름이 사전순으로 배열했을 때, 첫 번째 인자보다 크거나 같은 경우와 두 번째 인자의 작거나 같은 경우를 만족할 때 탐색을 한 책들의 정보를 출력하도록 한다. 이 역시 해당 범위 내에서 책을 찾지 못한 경우 오류 코드 300을 출력하도록 설계하였다.

PRINT_BP 함수의 경우 B+-tree 내에 있는 모든 데이터를 사전순으로 출력하도록 설계한다. 만약, B+-tree에 데이터가 없는 경우 해당 함수는 에러 코드 400을 출력하도록 하였다.

PRINT_ST의 경우 파라미터로 코드를 입력 받도록 한다. 해당 코드와 매핑되는 힙이 가진 모든 데이터를 출력하도록 한다. 해당 힙에 아무런 데이터가 없는 경우 에러 코드 600을 출력하도록 한다.

DELETE의 경우 selection tree가 가지는 사전순으로 가장 작은 책의 데이터를 삭제를 진행한다. Selection tree에 아무런 데이터가 없는 경우 에러 코드 600을 출력하도록 한

다. 마찬가지로, 삭제 연산에 성공하는 경우 해당 함수가 성공했음을 알리는 코드를 출력하도록 한다.

Exit의 경우 Exit을 출력하도록 하고, manager 객체를 소멸하면서 동적할당 되어 있는 것들은 동적할당을 해제하고, 파일이 열려 있는 경우 파일을 닫도록 한다.

BpTree.cpp

Bptree의 삽입은 탐색부터 시작된다. 먼저 해당 책의 제목과 같은 data가 이미 존재한다면 해당 자료의 대출 권수를 증가시키고, 할당된 새로 받은 데이터는 삭제하고 돌아가는 것으로 끝낸다. 하지만 데이터가 없는 경우에는 해당 데이터를 추가하기 위해 BST처럼 데이터가 들어갈 자리를 탐색하면서 들어가도록 한다. 데이터는 data node로 통하는 leaf node영역으로 갈 때까지 삽입의 위치를 찾도록 한다. 포인터가 인덱스 노드에 위치(즉, 데이터 노드로 가지 못한 상태)일 때 다음과 같은 내용을 반복하도록 한다.

index node의 맵에 저장된 데이터와 비교한다. 작으면 포인터는 mostleftchild쪽으로 아닌 경우에는 맵의 사이즈를 확인하도록 한다. 맵의 사이즈가 1이라면 단순히 map이 저장하고 있는 첫 번째 data의 second로 포인터가 이동하면 되지만, 사이즈가 2인 경우에는 두 번째 data의 first즉, 이름과 비교하여 포인터의 방향을 결정한다. 위치를 찾은 경우에는 데이터를 삽입하고 split 해야 되는지 검사를 하며 해당 함수를 종료하도록 한다. 다만 split을 해야 하는 경우 split data node 함수를 활용하여 split을 하도록 한다.

Data node의 split은 다음과 같이 진행한다. 먼저 왼쪽으로 갈라질 node를 분할하며 원래 갖고 있던 data node의 정보를 1:2의 형태로 left와 파라미터인 pDataNode가 나누어 가지게 된다. Data node끼리는 서로 이중 연결 리스트로 연결되도록 만든다. 다음은 케이스를 나누게 된다. 이미 data node가 부모를 가진 경우와 부모를 가지지 않은 경우(즉, split이 tree에서 처음으로 진행되는 경우)로 볼 수 있다. 후자의 경우에는 parent를 하나 BpIndexNode로 할당해주면서 분할된 data node가 가진 첫 번째 data의 제목을 가지게 되고 second로 pDataNode의 포인터가 들어간다. MostLeftChild까지 세팅해주고 분할된 두 node의 parent를 설정해주고 root가 해당 Index node의 포인터를 가지면 끝이다.

이번에는 pDataNode가 이미 Parent를 가지고 있던 경우이다. 부모 노드에 pDataNode의 첫 번째(분할을 진행한 후)제목과 자신의 포인터를 삽입한다. 분할을 했던 left child의 제목이 부모 노드의 제목보다 사전순으로 작으면 부모 노드의 left child만 재정의해주는 것으로 끝난다. 그렇지 않은 경우에는 부모 노드의 사이즈에 따른 케

이스가 생긴다. 부모 노드가 2개의 데이터를 담고 있는 경우, 분할된 left child는 부모 노드의 첫 번째 data의 second로 들어가도록 한다. 3개인 경우에는 부모 노드의 두 번째 data의 제목과 비교하여 작으면 전처럼, left child가 부모 노드의 첫 번째 data의 second로 아닌 경우에는 두 번째 data의 second로 들어간다. Data split을 종료하고 나서는 Parent의 경우에서도 split이 필요한지 검사하고 필요한 경우에 IndexNode의 split을 진행한다.
IndexNode이 split은 전체적으로 보면 data node의 split과 유사하다고 볼 수 있다. 몇 안 되는 다른 점이 존재한다. index node의 부모 노드에 data를 삽입하게 되면 pIndexNode가 가지고 있던 첫 번째 data(분할을 진행한 후)의 정보는 사라지고 해당 data가 가지고 있던 second는 남아 있는 pIndexNode의 left child로 세팅 된다. 또한, 경우의 수에 따라 left child가 재정의 되는 것이다.
Data의 탐색은 기본적으로 BST와 비슷하게 진행한다. 중간에 internal node에서 해당 값과 같은 data를 찾더라도 LoanBookData에 대한 정보를 담고 있지 않으니, Data node인 leaf node로 이동하여야 한다. Map에 contain을 얼마나 하고 있는가 와 찾고 있는 data를 따지며 leaf node까지 이동한다. 최종적으로 도달한 node에 해당 데이터가 없는 경우 nullptr을 반환, 아닌 경우에는 해당 node의 포인터를 반환한다.
모든 동적할당을 해제하기 위해 leaf node와 internal node를 구분하였다. Internal node를 큐에 삽입하면서 level order로 삭제할 수 있도록 설계하였다. 또한 가장 왼쪽에 존재하는 leaf node의 포인터를 먼저 저장하여 링크드 리스트의 특징을 이용해 동적할당을 진행한다. 또한 Data node에 map이 담고 있는 data이 second는 모두 동적할당 되어 있는 Loanbookdata*의 형태로 해당 data들을 모두 delete하고 data node역시 동적할당을 해제하며 모두 삭제할 수 있다. 해당 함수는 소멸자에서 진행하도록 설계했다.

SelectionTree.cpp
Selection tree의 초기 구축은 완전 이진 트리의 특성을 이용한다. Level3의 full binary tree를 만들어야 하므로, node의 포인터 배열을 15개를 만들어 동적할당과 left child, right child, parent를 설정해줄 수 있다. Left child는 부모 인덱스의 2배, right child는 부모 인덱스의 2배 + 1로 이를 이용하여 서로 연결해 줄 수 있다. 마지막으로 첫 번째 인덱스의 포인터를 root로 연결하면 selection tree의 구축이 끝난다.
Selection tree의 삽입은 받은 data의 분류 코드를 읽고 selection tree의 leaf node로 접근한다. Root를 1로 가정했을 때, leaf node는 4bit binary number이다. 이를 분류 코드별 leaf node로 접근할 수 있다. MSB를 제외하고 0인 경우 left child, 1인 경우 right

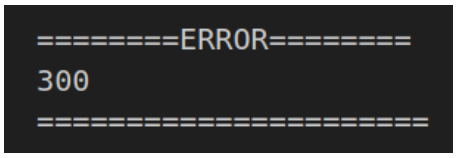
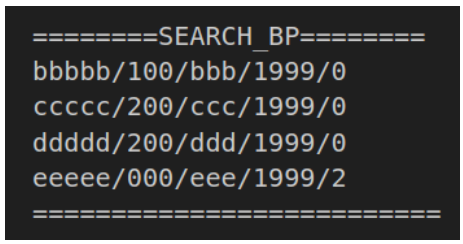
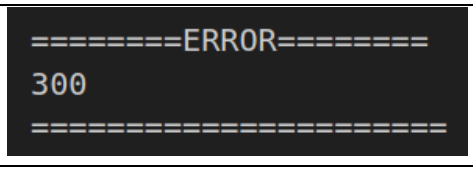
child로 이동하면 된다. Leaf node에 접근 이후 heap에 해당 data를 저장한다. Data를 저장하고 난 후, heap에 존재하는 root의 값을 기준으로 parent node와의 재정렬을 시행한다. Leaf node만 heap을 가지고 있으며, 상태에 따라 selection tree node는 loanbookdata를 갖는다. 부모 노드가 data를 이미 가지고 있는 상테에선 비교를, 없는 상태에선 자식 노드의 data를 복사한다.
삭제 연산은 다음과 같다. Root에 존재하는 data를 기준으로 leaf node까지 간다. 해당 selection tree는 Min winner tree로 root의 자취를 파악할 수 있다. 해당 기능을 이용하여 leaf node까지 접근하여, leaf node에 존재하는 heap에서의 data를 삭제한다. 이후 재정렬을 한다. 먼저 부모 노드로 이동하고 left child와 right child가 갖는 data를 비교, 작은 것을 부모 노드가 갖고 부모 노드가 nullptr을 가질 때, root까지 setting을 맞출 때 해당 함수는 true를 반환하며 종료한다.
Data를 출력할 때에 파라미터로 받은 분류 코드를 이용한다. 해당 코드의 이진수를 이용하여 leaf node까지 접근을 하여 heap에 대한 정보를 얻는다. 힙의 root를 vector container에 저장하면서 heap의 data를 삭제한다. Heap의 data는 삭제할 때 정렬되면서 나오므로 vector를 특별히 정렬할 필요는 없다. Vector에 있는 것을 출력하면서 다시 해당 data를 heap에 저장하고 vector에서는 data를 삭제한다.
Selection tree의 모든 node의 삭제는 먼저 delete 연산을 이용하여 모든 heap의 data를 삭제한다. Heap의 데이터가 남지 않아 selection tree의 root에 LoanBookData가 nullptr이 되면, selection tree에는 아무것도 남지 않은 것으로, 이제 노드들을 후위 순회하며 삭제한다.

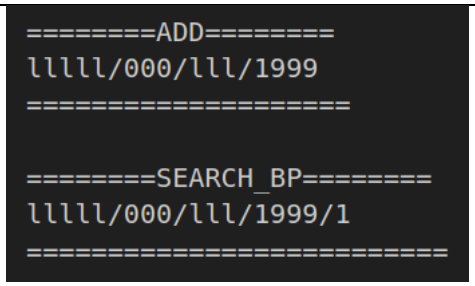
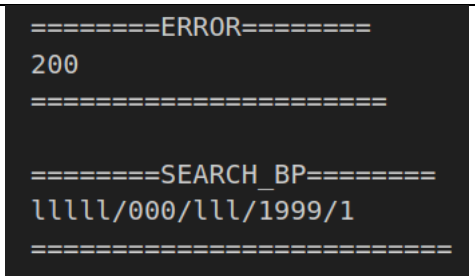
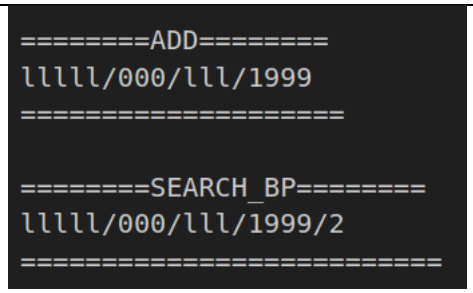
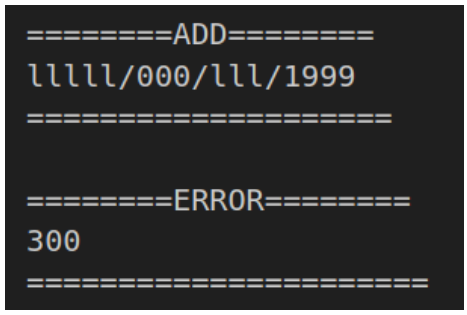
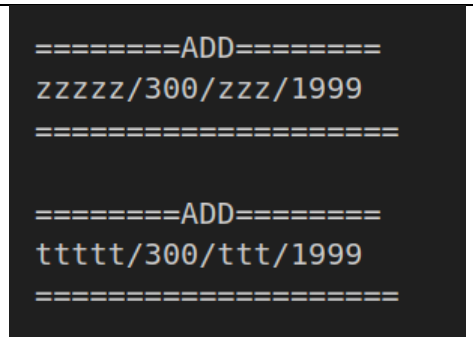
LoanBookHeap.cpp
다음은 heap에 삽입되는 함수이다. Root가 없던 경우에는 root에 해당 data를 넣으면서 size를 1로 만들어준다. Size는 private변수로 하나 만들었다. Heap의 level을 파악하여 추가될 노드의 위치를 파악한다. 파악한 이후 last element인 노드를 할당하여 추가될 노드의 data를 삽입하고 재정렬을 한다. 재정렬은 last element의 data를 root로 가져가는 과정에서 진행된다. 부모 노드보다 data가 큰 경우 서로의 data를 교환한다.
마지막으로 추가했던 element를 찾기 위해 size를 이용하여 노드의 위치를 파악한다. 마지막 노드를 파악한 후 heapifydown으로 root에 올라왔던 것을 다시 leaf node로 내리는 과정에서 정렬을 시행한다. Parent가 childe보다 작은 경우 parent와 child의 data를 교환한다. Child의 경우 parent 기준으로 left child와 right child를 비교 후 더 작은 쪽으로 child로 설정하여 data교환을 진행한다.

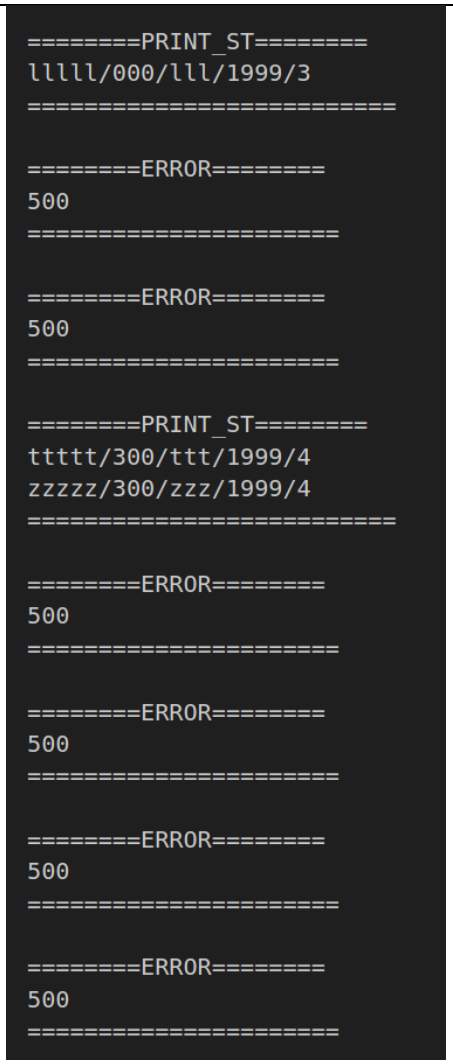
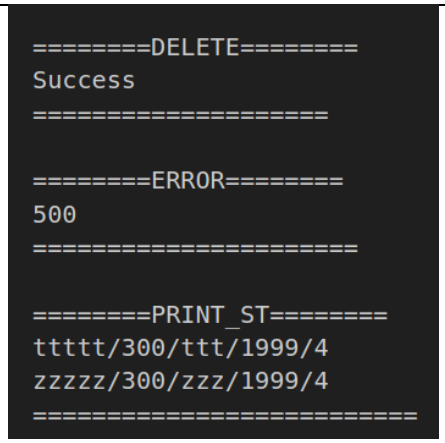
4. Result Screen

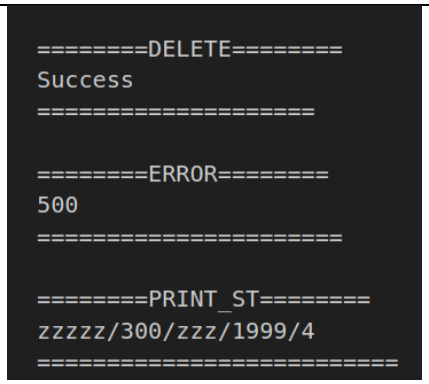
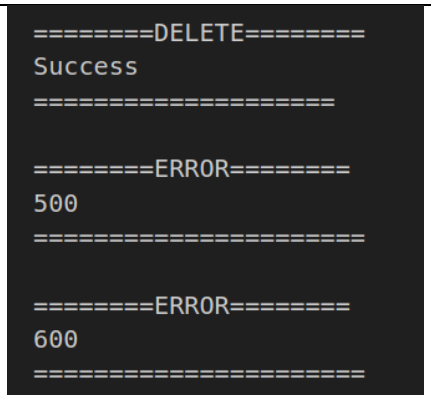
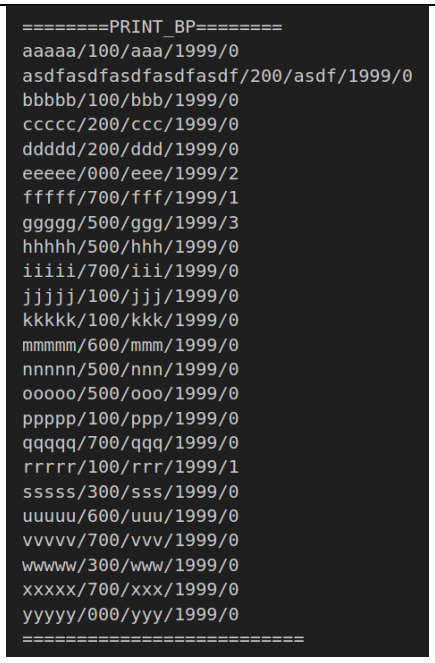
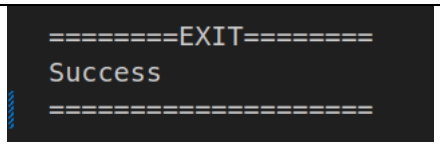
결과 화면	동작 설명
.txt ========LOAD======== Success ====================	첫 명령어 LOAD에 의한 실행 동작이다.
========PRINT_BP======== aaaaa/100/aaa/1999/0 bbbbb/100/bbb/1999/0 ccccc/200/ccc/1999/0 ddddd/200/ddd/1999/0 eeeee/000/eee/1999/2 fffff/700/fff/1999/1 ggggg/500/ggg/1999/3 hhhhh/500/hhh/1999/0 iiiii/700/iii/1999/0 jjjjj/100/jjj/1999/0 kkkkk/100/kkk/1999/0 lllll/000/lll/1999/0 mmmmm/600/mmm/1999/0 nnnnn/500/nnn/1999/0 ooooo/500/ooo/1999/0 ppppp/100/ppp/1999/0 qqqqq/700/qqq/1999/0 rrrrr/100/rrr/1999/1 sssss/300/sss/1999/0 ttttt/300/ttt/1999/3 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 zzzzz/300/zzz/1999/3 ==========================	값이 제대로 들어갔는지 확인하기 위해 PRINT_BP명령어를 이용하여, bptree 내 모든 값을 출력한 동작이다. 값이 모두 들어간 것을 확인할 수 있었으며, 또한 사전순으로 정렬되었음을 확인할 수 있다.
========ADD======== asdfasdfasdfasdfasdf/200/asdf/1999 ====================	해당 동작은 ADD를 사용했을 때 나오는 동작이다. 책의 제목, 분류 코드, 작가 이름, 출판 년도가 차례대로 나오는 것을 확인할 수 있다.
========ERROR======== 200 ======================	해당 동작은 ADD를 사용했지만 파라미터 수가 부족하여 나오는 에러 코드이다. 이 때 사용한 명령어는 책의 제목, 작가 이름, 출판 년도만 있었다. 책의 분류 코드가 없기 때문에 에러 메시지를 출력한다.

<table>
<tr>
<td colspan="2"> 


</td>
<td>해당 명령어는 SEARCH_BP를 이용하여 책의 이름을 검색하는 경우로, 탐색의 기능이 잘 되어지는 가를 확인한 명령어이다. LOAD할 때 넣었던 모든 데이터가 잘 삽입돼 있는 것을 확인할 수 있다.</td>
</tr>
<tr>
<td> 
</td>
<td>해당 오류 메시지는 search_bp 함수를 이용하지만 해당하는 내용의 책이 없는 경우 출력하는 메시지이다. 현재 bptree 내에 존재하지 않는 “asdfasdfasdfasdfa”를 제목으로 되어 있는 책을 찾으려 했으나 해당 데이터가 존재하지 않기 때문에 오류 메시지가 출력되었다.</td>
</tr>
<tr>
<td> 
</td>
<td>해당 동작은 SEARCH_BP이지만 범위로 검색하여 출력하는 경우이다. start인자를 b, end인자를 e로 주면서 그 사이에 존재하는 데이터의 정보를 출력한 것을 확인할 수 있다.</td>
</tr>
<tr>
<td> 
</td>
<td>해당 동작 역시 범위 출력을 이용하지만 범위내에 해당하는 데이터가 없는 경우로 오류 메시지를 출력한다.</td>
</tr>
</table>

	해당 동작은 ADD명령어를 이용하고 잘 들어갔는지 확인한 작업으로, 이미 LOAD를 통해 해당 책은 존재했으며, ADD 작업을 거치며 대출 권수가 증가함을 확인할 수 있다.
	해당 동작은 ADD명령어를 사용했지만, 인자가 부족하여 ADD 명령어가 실행되지 않았고, 검색했을 때, 대출 권수도 변동이 없는 것을 확인할 수 있다.
	해당 동작은 ADD명령어를 이용하여 제대로 실행됐는지 확인한 작업이다. 대출 권수가 1에서 2로 증가한 것을 확인할 수 있다.
	해당 동작은 ADD명령어를 이용하여 bptree에 존재하던 것을 대출 최대 권수까지 빌려 selection tree로 옮기는 것을 보여준다. Seletion tree로 옮긴 후 bptree 내에 해당 데이터가 존재하지 않으므로, search_bp book을 했을 때, 에러 코드300을 출력하는 것을 확인할 수 있다.
	해당 데이터 역시 ADD 명령어를 활용하여 대출 가능한 권수를 최대로 빌리면서 selection tree에 데이터가 이동한다.

	해당 명령어는 분류 코드 000부터 700까지 출력하는 모습이다. IIII의 경우 코드 분류 번호가 000으로 selection tree에 있는 000의 경우에는 저거 하나가 제대로 들어간 모습을 확인할 수 있다. 100, 200, 400 ~ 700은 비어 있는 모습을 확인할 수 있다. 비어 있는 경우 출력할 수 없기 때문에 에러 메시지 500이 출력되는 모습을 확인할 수 있다. 코드 번호 300의 경우 이전의 ADD 명령어를 통해 데이터가 모두 selection tree로 넘어간 것을 확인할 수 있다.
	DELETE 명령어를 통해 사전순으로 가장 빨랐던 IIIII이 삭제되어 000번의 코드를 출력하는데, 에러 메시지인 500이 출력되는 것을 확인할 수 있다. 또한 데이터 삭제는 사전순으로 가장 빠른 것만 삭제되는 것으로 300번대의 데이터들은 삭제되지 않는 것을 확인할 수 있다.

	해당 결과화면은 앞과 비슷하게 DELETE 연산을 시행할 때로 위 에러 메시지는 코드 번호 000의 경우를 출력하고 아래의 경우 300일 때 출력하는 것이다. 사전순으로 빠른 ttttt가 삭제된 것을 확인할 수 있고, root가 삭제되더라도 zzzzz는 남아 있는 것을 확인할 수 있다.
	해당 결과 화면은 DELETE를 수행하고 난 후 코드 300에 대해 출력을 다시 시도했으나 에러 메시지인 500이 뜨는 것을 확인할 수 있다. 이후 다시 delete 연산을 시도했지만 root에 남은 data가 없으므로 에러 코드 600이 출력되는 것을 확인할 수 있다.
	해당 명령어는 IIIII, ttttt, zzzzz가 selection tree로 넘어감에 따라 B+-tree에 남아 있지 않아야 하므로, 초기 B+-tree에서 데이터가 빠져 있는 것을 확인할 수 있다.
	Exit 명령어에 따라 프로그램을 종료한다.

5. Consideration

이번 프로젝트를 구현하면서, 힘들었던 점은 B+-tree를 처음 접하면서 삽입에 대한 문제가 생기는 거였다. Data를 삽입하게 되는데 base class로 동적할당 받아 derived class의 함수를 사용할 수 없는데, 이를 체크하지 못해 모든 것을 다 구현하고, test하는 과정에서 계속하여 segment fault가 발생하여 디버깅을 하면서 겨우 찾았다. 또한 index를 split하는 과정에서 문제가 생겨 B+-tree의 모든 node의 할당을 해제하는 과정에서 오류가 났다. B+-tree의 모든 node를 할당해제 할 때 queue에 넣으면서 level traval의 형식을 취하는데, 이미 삭제된 데이터에서 함수를 시행하니, 오류가 발생하는 것이었다. Index split에서 확인한 결과 split을 진행한 후 pIndexNode에서 가운데에 있던 것이 지워지면서 부모 노드에 삽입되지만 지우는 과정을 넣어주지 않음으로써 해당 오류가 나는 것을 발견했다. 해당 내용을 수정 후, 모든 노드를 삭제할 수 있었고 메모리 누수를 없앨 수 있었다. BST와는 비슷하지만 다른 점이라면 데이터가 증가하면 할수록 tree의 높이가 증가하는 건 같지만, B+-tree의 경우 그 높이가 BST보다 적게 증가하는 것을 체감할 수 있었다.

heap의 경우 마지막으로 추가한 노드에 대한 포인터를 계속 가져가면서 사용하려 했으나, 삽입이던 삭제 던 계속된 업데이트가 필요했다. 때문에 삽입과 삭제에서 불가피하게 계속하여 마지막 노드를 찾도록 설계하게 되었다.