데이터구조설계 보고서

Project 2

이　　름 : 김태관

학　　과 : 컴퓨터정보공학부

학　　번 : 2022202067

과　　목 : 데이터구조설계

담 당 교 수 : 이기훈 교수님

실 습 분 반 : 수 7, 8교시

A. Introduction

B+Tree와 Selection Tree, Heap을 이용하여 도서 대출 관리 프로그램을 구현합니다. 대출 관리 프로그램은 도서명, 분류 코드, 저자, 발행 연도, 대출 권수를 관리하며, 이를 이용해 대출 중인 도서와 대출 불가 도서에 대한 정보를 관리합니다.

프로그램은 도서 정보가 저장된 파일 loan_book.txt를 LOAD 명령어를 통해 읽어 해당 정보를 LoanBookData 클래스에 저장합니다. loan_book.txt에는 도서에 관한 정보가 '₩t'탭으로 구분되어 있습니다. 도서명은 항상 고유하며, 50자 이하로 가정합니다.

도서 분류 코드는 000~600번대까지 있으며, 각 분류 코드에 따른 대출 가능 권수는 아래와 같습니다.

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

B+ Tree

B+ Tree의 차수는 3으로 구현합니다. 주어진 loan_book.txt에 저장된 데이터를 읽은 후, 대출 중인 도서는 B+ Tree에 저장합니다.

ADD 명령어로 추가되는 데이터를 읽은 후, B+ Tree에 저장합니다. B+ Tree에 존재하지 않으면 Node를 새로 추가하고, 존재하는 경우에만 대출 권수를 증가시킵니다. 이후 대출 불가 도서가 되는 경우는 Selection Tree로 해당 도서를 전달하고, B+ Tree에서 제거합니다.

B+ Tree에 저장되는 데이터는 LoanBookData로 선언되어 있으며, 멤버 변수로는 도서명, 도서 분류 코드, 저자, 발행 연도, 대출 권수가 있습니다.

B+ Tree는 도서명을 기준으로 하며, 대소문자는 구별하지 않습니다. B+ Tree는 인덱스 노드와 데이터 노드로 구성되며, 각 노드 클래스는 B+ Tree 노드 클래스를 상속받습니다.

B+ Tree의 데이터 노드는 도서 정보를 저장한 LoanBookData를 map 컨테이너 형식으로 가지고 있습니다.

Selection Tree

Selention Tree는 도서명을 기준으로 Min Winner Tree를 구성합니다.

Selection의 run 개수는 도서 분류 코드의 개수와 동일합니다.

ADD 명령어로 B+ Tree에 저장된 책의 대출 권수를 업데이트한 후에 대출 불가 도서는 도서 분류 코드에 따라 Min Heap으로 구현된 Selection Tree의 각 run에 저장합니다. B+ Tree에 저장되어 있던 데이터는 삭제합니다.

LoanBook Heap

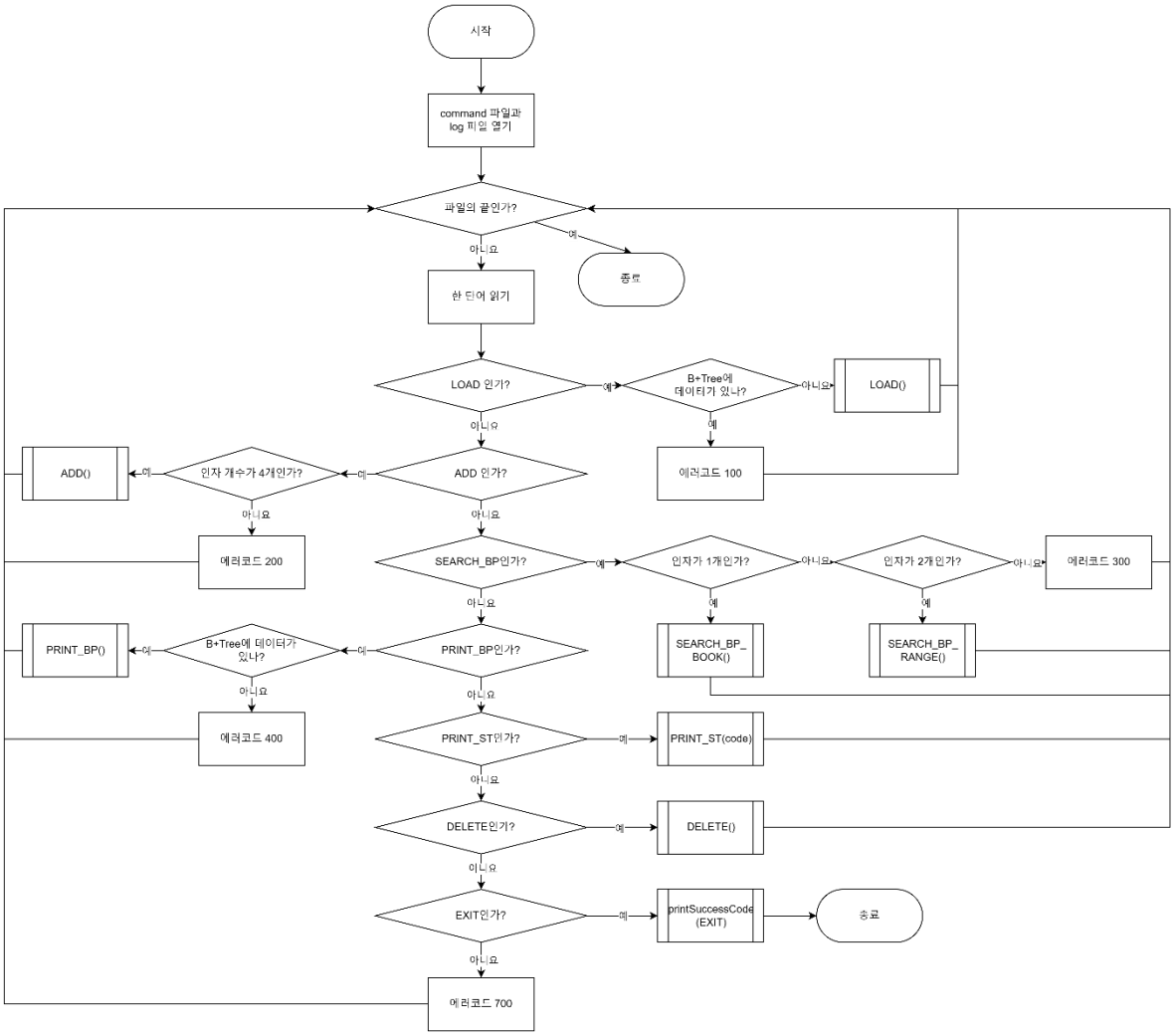
LoanBook Heap은 Min Heap으로 도서명을 기준으로 정렬됩니다.

대출 불가 도서의 정보를 받아와 Heap을 생성합니다.

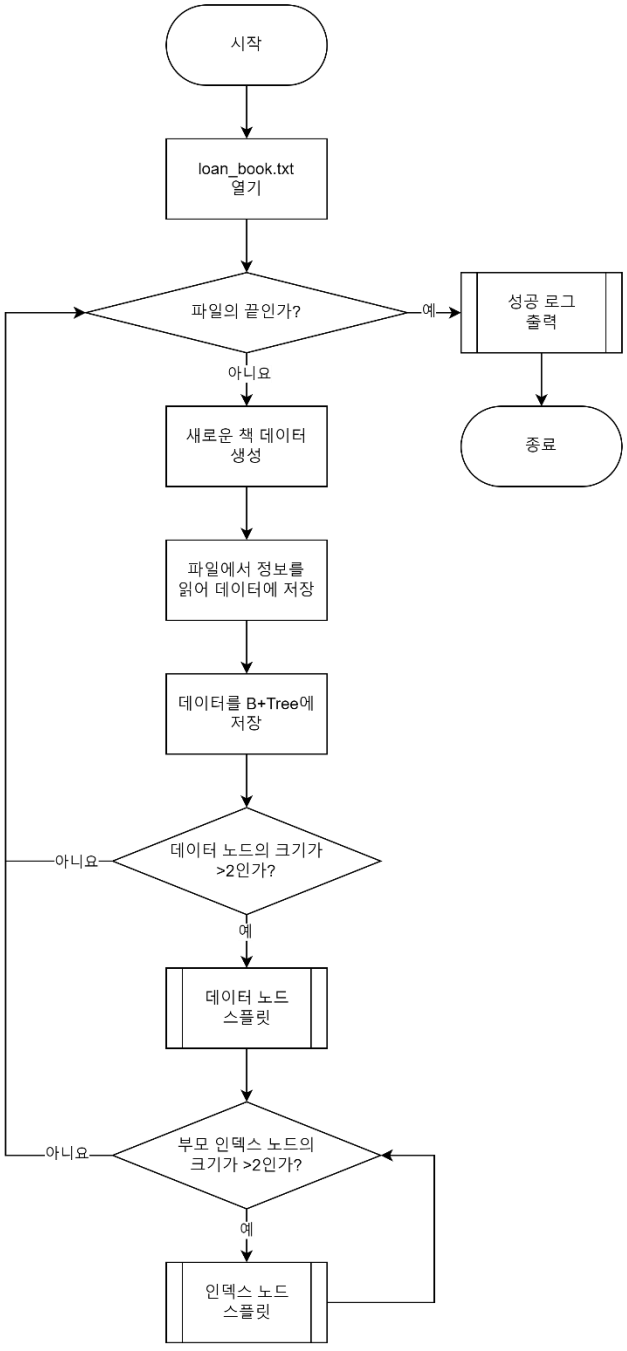
Heap을 재정렬할 때마다 Selection Tree도 재정렬합니다.

B. Flow chart

Manager

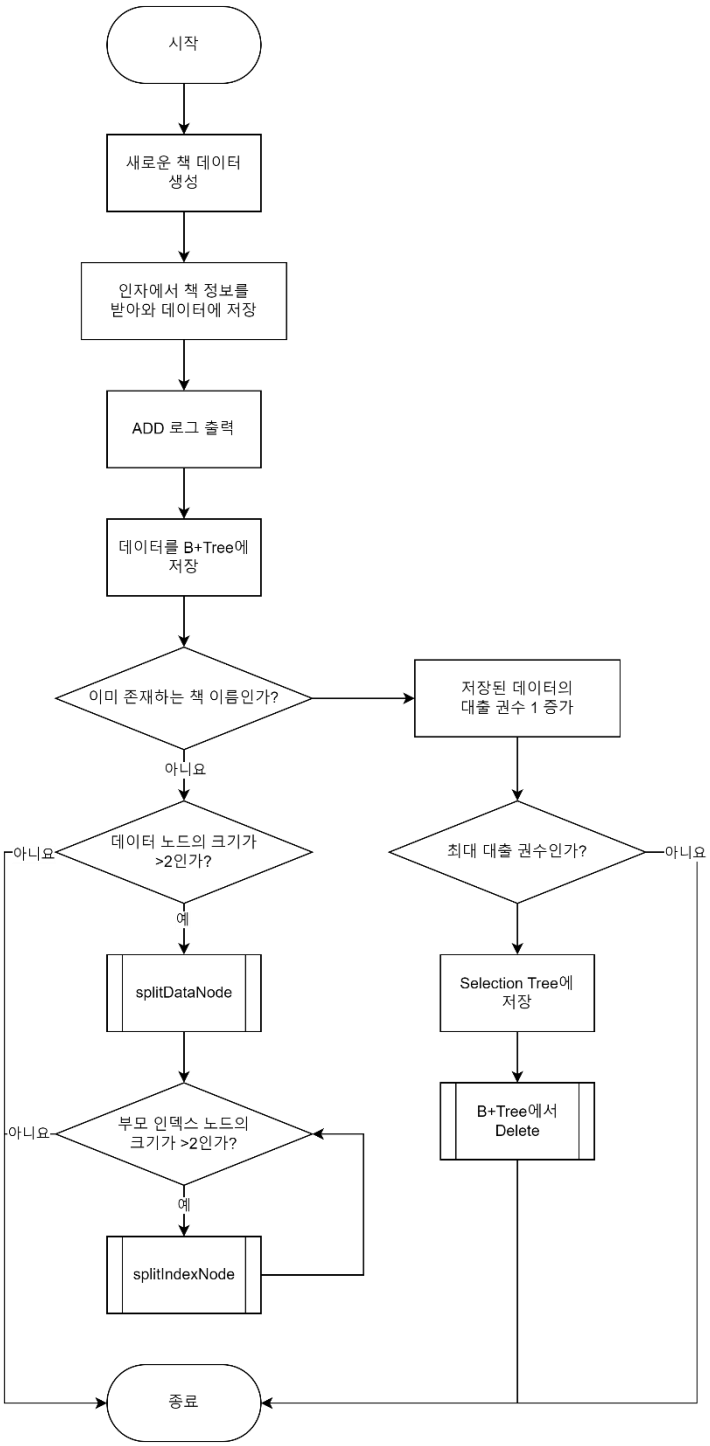


LOAD



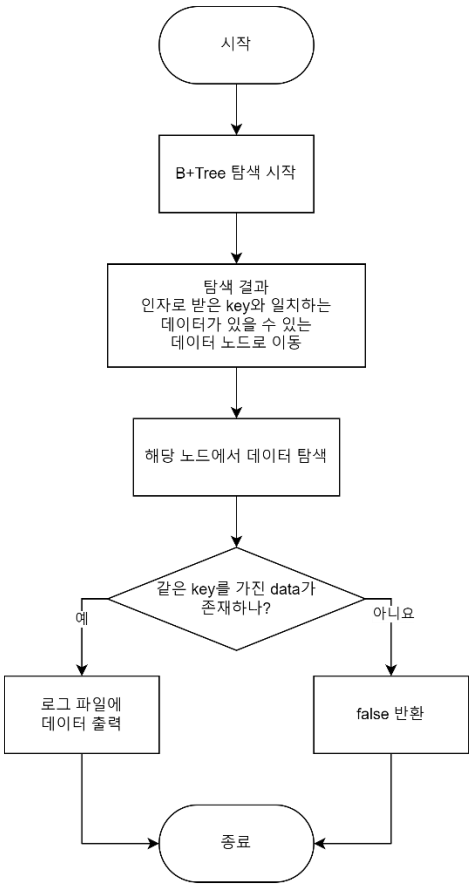
loan_book.txt 파일을 열어 책 정보를 하나 씩 읽고 LoanBookData 클래스에 저장합니다. 그리고 해당 LoanBookData 객체를 B+Tree에 저장합니다. 이때 데이터 노드 오버플로우가 발생하면 데이터 노드를 스플릿합니다. 여기서 다시 부모 인덱스 노드에 오버플로우가 발생하면 부모 인덱스 노드를 스플릿합니다. 오버플로우가 생기지 않을 때까지 스플릿을 반복합니다. 그리고 이 동작을 파일의 끝까지 반복합니다.

ADD



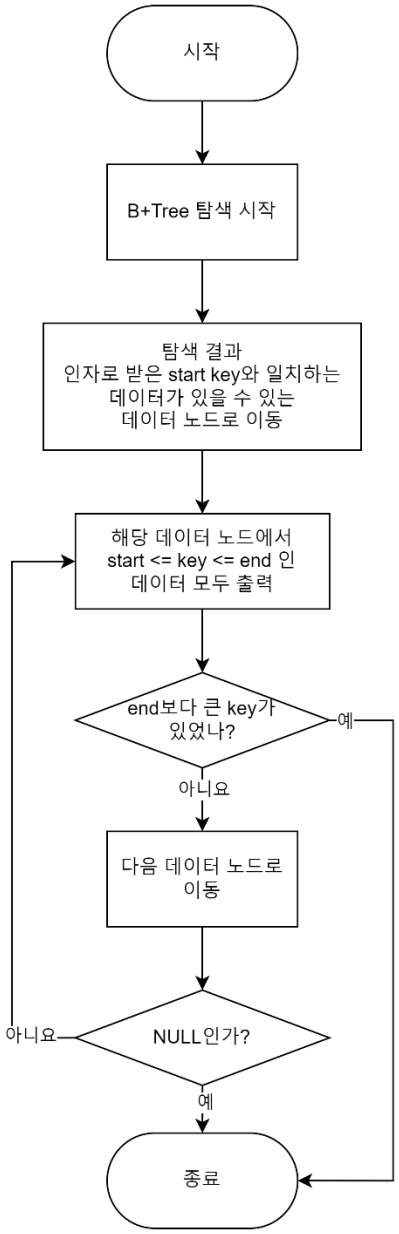
인자로 입력 받은 책 정보를 LoanBookData 클래스에 저장하고 이 객체를 B+Tree에 저장합니다. 만약 같은 이름을 가진 책이 있다면 해당 책의 대출 권수를 1 증가시킵니다. 그리고 해당 책의 대출 권수가 최대 대출 권수와 같아지면 해당 책의 LoanBookData를 Selection Tree에 저장합니다. 그리고 B+Tree에서는 해당 데이터를 삭제합니다. 같은 책이 존재하지 않는다면 LOAD와 동일하게 저장하고 노드 오버플로우가 발생하면 스플릿을 진행합니다.

SEARCH_BP_BOOK



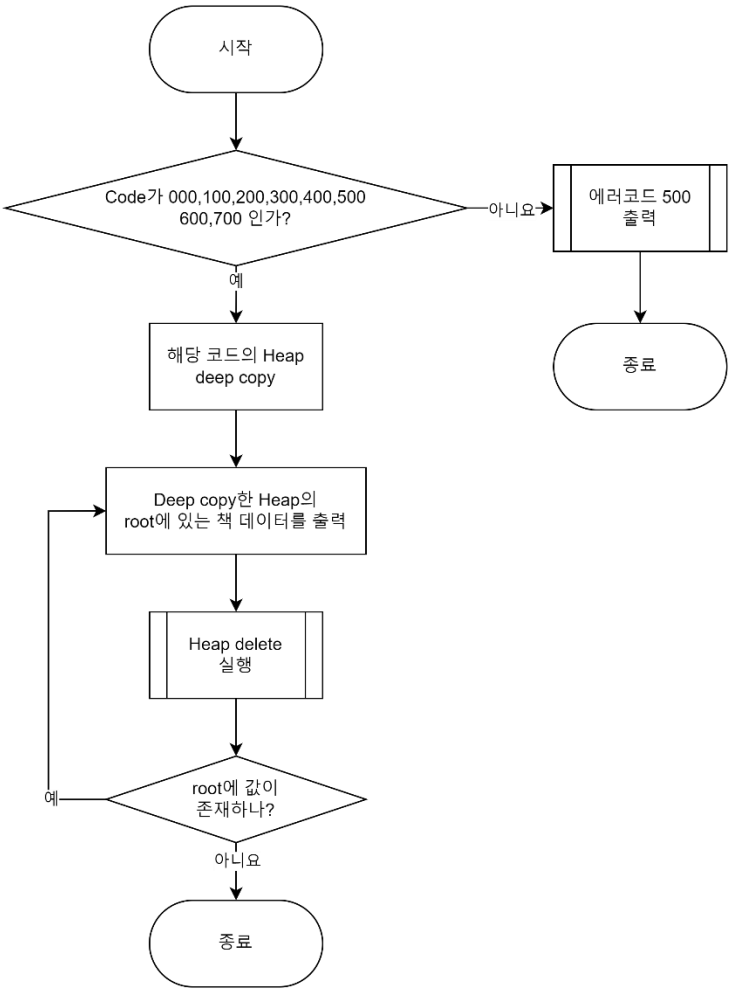
인자의 책 이름을 key로 하여 B+Tree를 탐색합니다. Index 노드의 key보다 작으면 왼쪽 노드로, key보다 크거나 같으면 오른쪽 노드로 이동하여 최종적으로 같은 key를 가진 데이터가 있을 수 있는 데이터 노드에 도달합니다. 해당 노드에 입력 key와 같은 key를 가진 데이터가 존재하면 해당 책의 데이터를 로그 파일에 출력합니다. 존재하지 않으면 0을 반환하고 Manager에서 에러를 출력합니다.

SEARCH_BP_RANGE



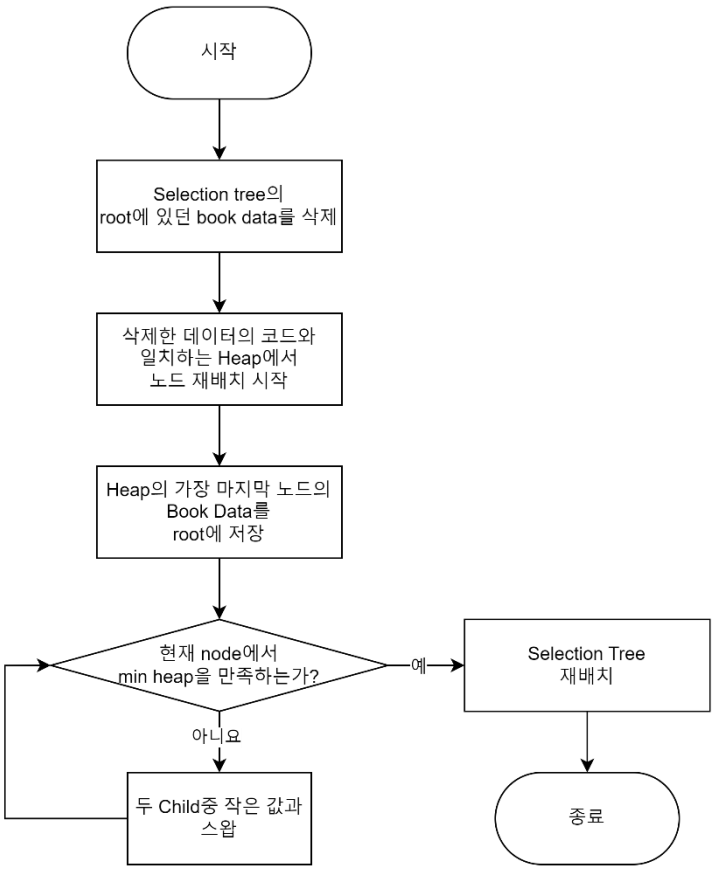
인자의 start를 key로 하여 B+Tree를 탐색합니다. key를 비교할 때 가장 첫 문자만 비교합니다. Index 노드의 key보다 작으면 왼쪽 노드로, key보다 크거나 같으면 오른쪽 노드로 이동하여 최종적으로 같은 key를 가진 데이터가 있을 수 있는 데이터 노드에 도달합니다. 해당 노드부터 start보다 크거나 같고 end보다 작거나 같은 모든 데이터를 출력합니다.

PRINT_ST



인자로 입력 받은 Code에 대응하는 Heap의 내부 데이터를 복사해 새로운 heap을 만듭니다. 그리고 해당 Heap의 root를 출력하고 삭제하는 과정을 반복하여 Heap을 출력합니다.

DELETE



Selection Tree의 root에 있던 Heap에서 삭제를 진행하고 Selection Tree를 재배치합니다.

C. Algorithm

B+Tree

책 정보를 받아와 책 이름을 기준으로 저장합니다. 만약 저장한 노드에서 오버플로우가 발생하면 해당 데이터 노드를 Split 합니다. Split 한 이후 부모 인덱스 노드에서도 오버플로우가 발생하면 해당 인덱스 노드를 Split 합니다. 오버플로우가 발생할 때마다 Split 을 반복하여 진행합니다.

삭제 연산은 그냥 삭제해도 되는 케이스, 그 외 데이터 노드 케이스 4 가지로 크게 5 가지로 나눌 수 있습니다.

그냥 삭제해도 되는 케이스는 삭제하는 데이터가 노드의 첫 번째 값이 아니고 인덱스 노드에 해당 key 값이 없는 경우입니다.

그 외에는 데이터가 노드의 첫 번째 값이거나 인덱스 노드에 key 값이 있는 경우입니다.

데이터 노드의 다음 형제 노드가 존재하고, 형제 노드의 사이즈가 1 보다 클 때 형제 노드의 가장 작은 값을 가져옵니다.

데이터 노드의 이전 형제 노드가 존재하고, 형제 노드의 사이즈가 1 보다 클 때 형제 노드의 가장 큰 값을 가져옵니다.

데이터 노드의 다음 형제 노드가 존재하고, 형제 노드의 사이즈가 1 일 때 형제 노드와 현재 데이터 노드를 병합합니다.

데이터 노드의 이전 형제 노드가 존재하고, 형제 노드의 사이즈가 1 일 때 형재 노드와 현재 데이터 노드를 병합합니다.

데이터 노드를 병합한 후 부모 노드에서 언더플로우가 발생하면 다시 4 가지 케이스로 나뉩니다.

인덱스 노드의 다음 형제 노드가 존재하고, 형제 노드의 사이즈가 1 보다 클 때 형제 노드의 가장 작은 값을 가져옵니다.

인덱스 노드의 이전 형제 노드가 존재하고, 형제 노드의 사이즈가 1 보다 클 때 형제 노드의 가장 큰 값을 가져옵니다.

인덱스 노드의 다음 형제 노드가 존재하고, 형제 노드의 사이즈가 1 일 때 형제 노드와 현재 인덱스 노드를 병합합니다.

인덱스 노드의 이전 형제 노드가 존재하고, 형제 노드의 사이즈가 1 일 때 형재 노드와 현재 인덱스 노드를 병합합니다.

데이터 노드와 마찬가지로 부모 노드에서 언더플로우가 발생하면 재귀적으로 동작합니다.

Re-Balancing 이 끝난 뒤, 그냥 삭제해도 문제가 생기지 않는 상태가 되고 데이터 노드에서 값 삭제를 진행합니다. 이후 부모 노드를 재귀적으로 탐색하며 인덱스에 남아있는 key 값을 삭제하고 삭제 연산을 종료합니다.

Selection Tree

Selection Tree 가 처음 생성될 때만 Array 를 이용하여 트리 구조를 만들고 이후 재정렬할 때에는 Left, Right Child 와 Parent 노드를 활용하도록 구현하였습니다. 000 부터 700 까지 8 개의 Min Heap 을 run 으로 하여 Min Winner Tree 를 구현합니다. Selection Tree 에서 delete 를 진행할 때 Heap 에서 delete 를 먼저 진행한 후 Selection Tree 를 가장 밑에서부터 재정렬하여 다시 Selection Tree 를 구성합니다.

Min Heap

Heap 에서 Array 를 쓰지 않고 Index 만으로 노드의 위치를 찾기 위해 이진수를 사용하였습니다.

만약 루트가 1, 왼쪽 노드는 2n, 오른쪽 노드는 2n+1 이라는 규칙으로 노드 위치를 정한다고 할 때 인덱스 7 의 노드를 찾아보겠습니다. 7 은 이진수로 111 이고 가장 왼쪽 1 비트는 버리고 탐색을 시작합니다. 그러면 나머지 비트는 11 이고 왼쪽부터 1 비트씩 확인합니다. 0 이면 왼쪽, 1 이면 오른쪽으로 이동합니다. 결과적으로 인덱스 7 이라는 정보 하나만으로 루트부터 시작하여 오른쪽-오른쪽 위치가 인덱스 7 노드의 위치라는 것을 찾을 수 있습니다. 해당 알고리즘을 사용하여 Array 를 사용하지 않고 노드의 위치를 정확하게 찾을 수 있고, Heap size 와 같이 활용하여 마지막 노드의 위치를 찾아 삽입하거나, 삭제할 수 있습니다.

D. Result Screen

command.txt

```
LOAD
LOAD
PRINT_BP
SEARCH_BP       bbbbb
SEARCH_BP       zzzzz
SEARCH_BP       c       f
ADD     aaaaa   200     abcd    2022
ADD     ddddd   500     abcd    2022
ADD     fffff   500     abcd    2022
ADD     aaaa    000     abcd    2022    2
SEARCH_BP       aaaaa
ADD     aaaaa   200     abcd    2022
PRINT_BP
PRINT_ST        500
PRINT_ST        200
DELETE
DELETE
DELETE
DELETE
PRINT_ST        500
PRINT_ST        200
EXIT
```

loan_book.txt

```
aaaaa   200     abcd    2022    1
bbbbb   500     abcd    2022    1
ccccc   500     abcd    2022    1
ddddd   500     abcd    2022    1
eeeee   500     abcd    2022    1
fffff   500     abcd    2022    1
ggggg   500     abcd    2022    1
hhhhh   500     abcd    2022    1
```

결과 화면에 사용된 command 와 loan_book 파일입니다.

log.txt	설명
========LOAD======== Success ========== ========ERROR======== 100 ==========	두 번의 LOAD 명령을 실행합니다. 첫 LOAD 는 성공 로그를 출력하고 두 번째 LOAD 는 이미 B+ Tree 가 존재하므로 에러를 출력합니다.
========PRINT_BP======== aaaaa/200/abcd/2022/1 bbbbb/500/abcd/2022/1 ccccc/500/abcd/2022/1 ddddd/500/abcd/2022/1 eeeee/500/abcd/2022/1 fffff/500/abcd/2022/1 ggggg/500/abcd/2022/1 hhhhh/500/abcd/2022/1 ==========	PRINT_BP 명령어를 실행하여 B+Tree 에 LOAD 된 책 정보를 오름차순으로 출력하였습니다.

`` ========SEARCH_BP======== bbbbb/500/abcd/2022/1 ========================== ========ERROR======== 300 ======================= ```	두 번의 SEARCH_BP_BOOK 을 실행합니다. 첫 명령어는 tree 에 존재하기 때문에 해당 책의 정보를 출력합니다. 두 번째 명령어는 z 를 찾는 명령어인데 존재하지 않으므로 에러를 출력합니다.
`` ========SEARCH_BP======== ccccc/500/abcd/2022/1 ddddd/500/abcd/2022/1 eeeee/500/abcd/2022/1 fffff/500/abcd/2022/1 ========================== ```	SEARCH_BP_RANGE 를 실행합니다. c 부터 f 까지 이므로 그 사이의 모든 책 정보를 출력합니다.
`` ========ADD======== aaaaa/200/abcd/2022 ==================== ========ADD======== ddddd/500/abcd/2022 ==================== ========ADD======== fffff/500/abcd/2022 ==================== ========ERROR======== 200 ======================= ```	네 번의 ADD 명령을 실행합니다. 처음 세 책은 인자가 정상적으로 입력되었기 때문에 에러가 발생하지 않습니다. 네 번째 명령어는 인자가 비정상적이기 때문에 에러가 발생합니다. 그리고 분류코드 500 은 최대 대출 권수가 2 이므로 ddddd 와 fffff 는 대출 불가 도서가 되어 Selection Tree 에 추가됩니다.
`` ========SEARCH_BP======== aaaaa/200/abcd/2022/2 ========================== ========ADD======== aaaaa/200/abcd/2022 ==================== ```	aaaaa 를 확인해보면 대출 권수가 1 증가한 것을 확인할 수 있고 한 번 더 ADD 하여 대출 불가 도서가 되었습니다.
`` ========PRINT_BP======== bbbbb/500/abcd/2022/1 ccccc/500/abcd/2022/1 eeeee/500/abcd/2022/1 ggggg/500/abcd/2022/1 hhhhh/500/abcd/2022/1 ======================= ```	B+Tree 를 확인해보면 이전 명령에서 대출 불가 도서가 된 책이 삭제된 것을 확인할 수 있습니다.

`` ========PRINT_ST======== ddddd/500/abcd/2022/2 fffff/500/abcd/2022/2 ======================== ========PRINT_ST======== aaaaa/200/abcd/2022/3 ======================== ```	PRINT_ST 명령으로 각각 500 번과 200 번의 Heap 을 오름차순으로 출력하였습니다. 책의 정보와 대출 권수가 정확하게 저장되어 있는 것을 확인할 수 있습니다.
`` ========DELETE======== Success ======================== ========DELETE======== Success ======================== ========DELETE======== Success ======================== ========ERROR======== 600 ======================== ```	이후 네 번의 DELETE 명령을 실행합니다. 세 번까지는 Selection Tree 에 정보가 있기 때문에 정상적으로 실행되지만, 네 번째 DELETE 는 Selection Tree 가 비어있기 때문에 에러를 출력합니다.
`` ========ERROR======== 500 ======================== ========ERROR======== 500 ======================== ```	이전과 같이 500 번과 200 번의 Heap 을 출력하려고 했을 때 Heap 이 비어있기 때문에 에러가 발생합니다.
`` ========EXIT======== Success ======================== ```	EXIT 명령을 실행하고 프로그램이 종료됩니다.

E. Consideration

B+Tree를 구현하면서 모든 order에서 실행되도록 구현하려 했지만 생각보다 어려워 order가 3일 때만 실행되도록 구현하였습니다. 이후 B+Tree를 개인적으로 한 번 더 구현해보며 모든 order에서 정상적으로 실행되도록 구현해 볼 것입니다.

Selection Tree와 Heap을 구현할 때 Array를 쓰지 않고 구현하는 것이 매우 어려웠습니다. Array를 사용할 때에는 index로 바로 노드에 접근하여 사용할 수 있지만 링크드리스트로 구현된 트리에서 index로 노드를 찾는 것이 불가능해 보였습니다. 여러가지 방법을 생각하던 중 트리의 자식 노드 개수는 2개니까 이진수로 접근할 수 있을 것 같았습니다. 간단한 완전 이진 트리를 그려 놓고 루트부터 1, 10, 11, 100, 101 … 써 내려가면서 규칙을 찾을 수 있었습니다. 가장 왼쪽 1을 지우고 왼쪽부터 차례대로 0이면 왼쪽 노드, 1이면 오른쪽 노드로 내려가면 최종적으로 원하는 노드에 도달할 수 있습니다. 해당 알고리즘으로 index를 이용해 해당 위치의 노드를 반환하는 함수를 만들어 사용하였습니다.

B+Tree의 삭제 연산을 구현하는 것이 가장 어려웠습니다. 기본적으로 고려해야 할 경우의 수가 매우 많고 경우의 수를 모두 알아낸다 하더라도 해당 경우에서 해야 할 작업이 매우 복잡했기 때문에 구현이 매우 어려웠습니다. 언더플로우가 발생하는 경우를 정리하며 merge할 때만 언더플로우가 발생할 수 있다는 것을 알게 되었고, merge함수에서 또 다른 함수를 재귀적으로 호출하여 모든 경우의 수를 처리할 수 있었습니다.