데이터 구조

2차

Project

학번:2024402041

실습 분반: 수

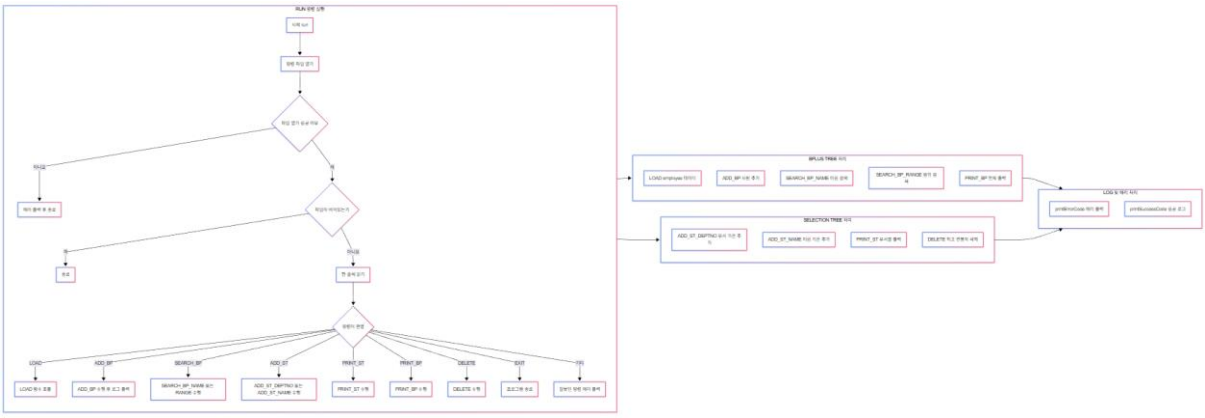
이름: 심수정

1. Introduction

본 프로젝트는 B+ tree, Selection tree, Max Heap를 이용해 사원의 부서코드, 사번, 연봉 정보를 효율적으로 관리하는 프로그램을 구현하는 것이다. 프로그램은 employee.txt 파일로부터 사원 데이터를 불러와 B+ tree에 저장하며, B+ tree에서는 이 사원 데이터를 이름 기준 오름차순으로 정렬하여 관리한다. Selection tree에서는 부서별 데이터를 기반으로 연봉 내림차순으로 정렬해 각 부서 내 최고 연봉자 선별 및 삭제 기능을 수행한다.

2. Flow Chart

A. Manager.cpp



command.txt 파일을 열어 한 줄씩 명령어를 수행한다.

LOAD 명령어는 employee.txt 파일에서 사원 데이터를 읽어 B+ 트리에 삽입한다.

ADD_BP 명령어는 새로운 사원 데이터를 직접 추가하는 기능으로, 중복되는 사원이 입력으로 들어올 경우에는 연봉만 업데이트 하고, 중복되지 않을 경우에는 B+ 트리에 삽입한다.

SEARCH_BP 명령어는 이름이나 범위를 기준으로 B+ 트리 안의 사원 정보를 검색하여 그 정보를 로그 파일에 출력한다.

PRINT_BP 명령어는 B+ 트리에 저장된 모든 사원 정보를 이름 오름차순으로 로그 파일에 출력한다.

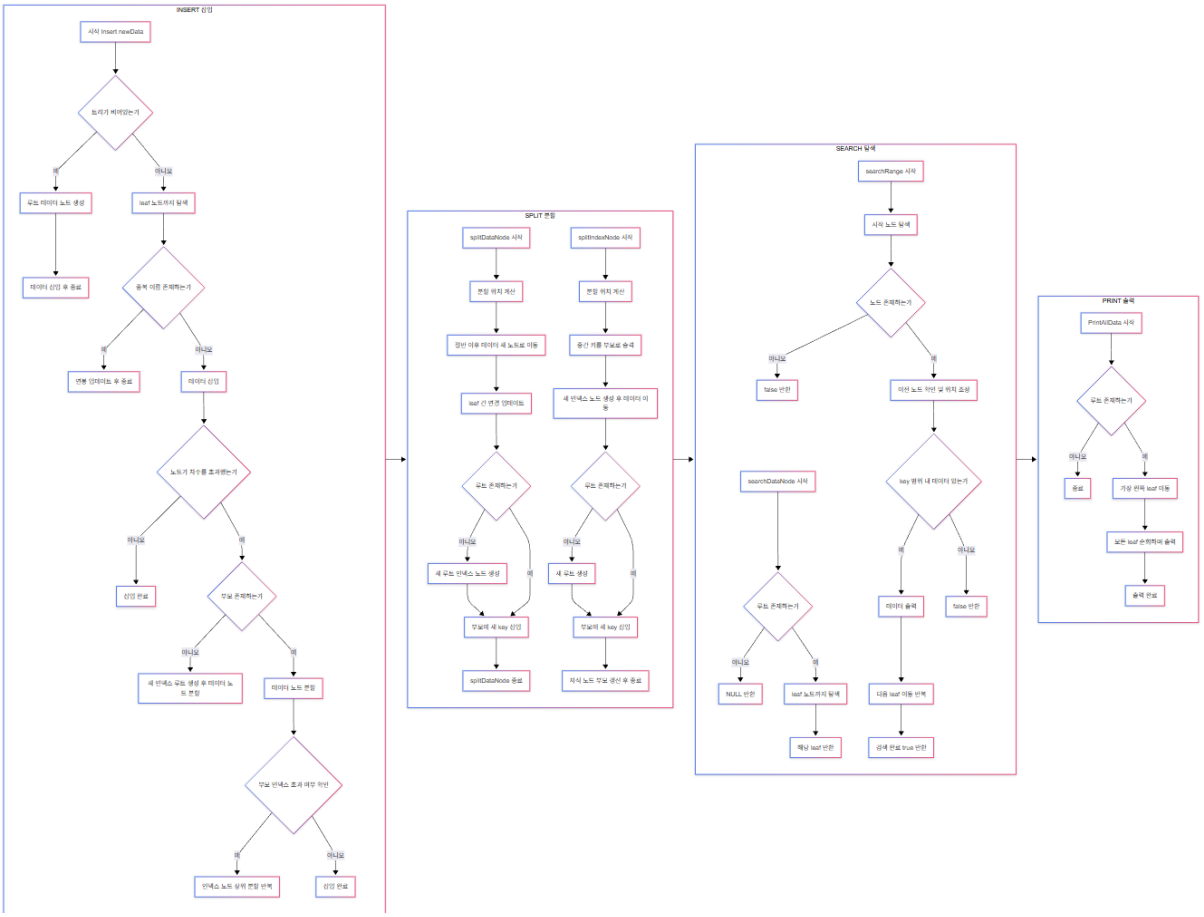
ADD_ST 명령어는 Selection 트리에 삽입하는 기능으로 부서 번호나 이름을 기준으로 삽입을 수행한다. 이때 Selection 트리는 연봉 내림차순으로 데이터

를 정렬해 관리한다.

PRINT_ST 명령어는 특정 부서의 사원 정보를 연봉 순으로 출력한다.

DELETE 명령어는 Selection 트리의 최고 연봉자를 삭제한다.

B. BpTree.cpp



트리는 이름을 key로 하여 데이터를 오름차순으로 정렬하고 노드의 차수가 초과될 경우 자동으로 분할되어 균형을 유지한다.

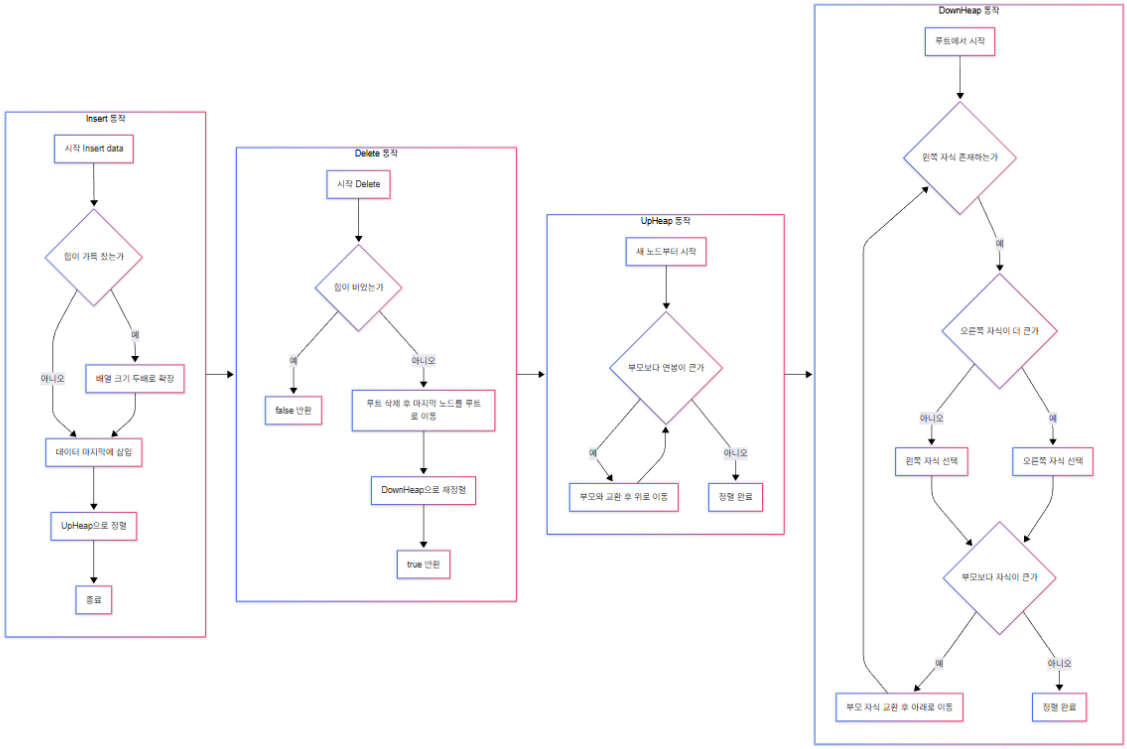
먼저 삽입(Insert) 단계에서는 새로운 데이터를 B+ 트리의 리프 노드까지 탐색하고, 중복 여부를 검사해 중복되지 않을 때에 삽입한다. 삽입한 후 노드의 데이터 수가 차수를 초과하면 노드는 분할된다.

분할(Split) 단계에서는 노드의 중간 위치를 기준으로 데이터를 절반으로 나누어 새 노드를 생성하고, 부모 인덱스 노드에 분할된 키를 삽입한다.

검색(Search) 단계에서는 루트 노드부터 시작해 리프 노드로 향하며 키 값을 비교하며 원하는 이름이나 범위에 해당하는 데이터를 찾는다. 검색 결과는 log.txt 파일에 출력한다.

출력(Print) 단계에서는 가장 왼쪽 리프 노드부터 순차적으로 모든 데이터를 In-Order 방식으로 출력해 트리 안의 모든 사원 정보를 이름 오름차순으로 확인할 수 있다.

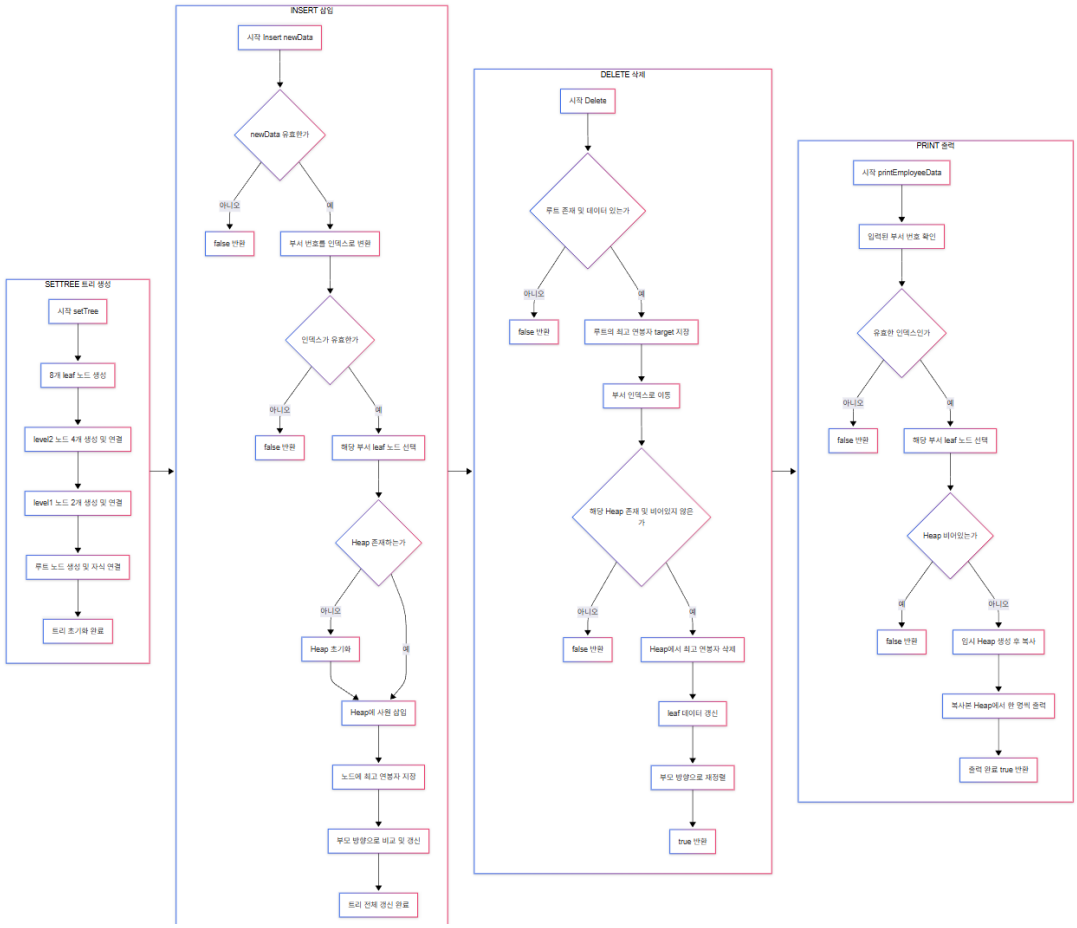
C. EmployeeHeap.cpp



EmployeeHeap은 Max Heap으로, 연봉 기준으로 내림차순 정렬해 관리한다. 삽입(Insert) 단계에서는 새로 입력된 사원 데이터를 heap의 가장 마지막 위치에 삽입하고, 부모 노드와 비교하여 부모 노드보다 연봉이 더 높은 경우엔 위로 이동시키는 UpHeap 연산을 수행한다. 이 과정을 통해 부모 노드의 연봉이 자식 노드의 연봉보다 항상 높게 유지하게 한다.

삭제(Delete) 단계에서는 최고 연봉자에 해당하는 루트를 삭제하고 마지막 노드를 루트로 이동시켜 DownHeap 연산을 통해 다시 정렬 조건을 만족하도록 한다.

D. SelectionTree.cpp



Selection 트리는 사원 데이터를 부서 번호로 구분해 관리하고 각 부서 내에서는 연봉 내림차순으로 정렬하는 과정을 수행한다.

트리 초기화(setTree) 단계에서는 트리의 전체 구조를 초기화한다. Run에 해당하는 8개의 리프 노드를 생성하고, 이를 내부 노드(internal node)와 루트로 연결해 완전 이진 트리를 구성한다.

삽입(Insert) 단계에서는 새로운 사원 데이터를 해당 부서의 리프 노드에 삽입해 Heap을 이용하여 부서 내에서 연봉 내림차순 정렬을 유지한다. 삽입이 끝나면 위로 올라가 왼쪽과 오른쪽 자식 노드의 최고 연봉자를 비교해 부모 노드에 더 높은 연봉자를 저장한다. 이 과정을 반복해 루트에는 전체 사원 중 최고 연봉자가 있게 된다.

삭제(Delete) 단계에서는 최고 연봉자에 해당하는 루트를 삭제하고 해당 사원이 속하는 부서의 힙에서 루트 데이터를 제거한다. 그리고 힙에서 새로운 최고 연봉자를 갱신하고 다시 비교하며 올라가 selection tree의 구조를 유지한다.

3. Algorithm

<B+ tree>

B+ tree는 두 가지 유형의 노드로 구성된다. 먼저 인덱스 노드는 트리의 내부 노

드에 해당하며, 실제 데이터를 저장하지 않고 key와 자식 노드에 대한 포인터를 저장한다. 탐색할 때에는 데이터가 저장된 리프 노드로 이동하기 위한 경로 정보를 제공한다. 데이터 노드는 실제 데이터를 저장하는 리프 노드로, 각 데이터 노드는 사원 객체를 포함한다. 모든 데이터 노드는 같은 레벨에 존재한다.

B+ 트리의 각 노드는 최대 차수(order)를 가지며 노드가 가득 차면 자동으로 분할 연산을 수행해 트리의 균형을 유지한다. 이 프로그램에서 차수는 기본적으로 3으로 설정되어 있지만 변경도 가능하도록 구현하였다.

삽입 과정은 루트에서 시작하여 적절한 리프 노드를 찾은 뒤 데이터를 추가하는 방식으로 진행된다. 삽입 후 노드의 데이터 수가 차수를 초과하면 중앙 키를 기준으로 노드를 분할하고, 중간 키를 부모 인덱스 노드로 올린다. 필요할 경우 이 분할은 상위 노드로 전파되어 루트까지 이어질 수 있으며, 이를 통해 트리는 항상 균형을 유지한다.

탐색은 루트에서 시작해 키 값을 비교하며 자식 포인터를 따라 이동하는 방식으로 이루어진다. 리프 노드에 도달하면 해당 키에 해당하는 데이터를 직접 검색할 수 있다. 이때 트리의 높이가 일정하게 유지되므로 탐색의 시간 복잡도는 O(log N)에 해당한다.

범위 검색의 경우 시작 키를 포함한 리프 노드에서부터 오른쪽으로 연결된 리프 노드들을 순차적으로 탐색하며, 지정된 끝 키에 도달할 때까지 데이터를 연속적으로 검색한다. 이때 리프 노드 간의 연결 구조를 이용하므로 연속된 데이터 검색이 매우 빠르게 이루어진다.

전체 데이터를 출력할 때는 가장 왼쪽 리프 노드에서 시작해 오른쪽으로 이동하면서 모든 데이터를 순차적으로 출력한다. 이 과정은 트리 전체를 In-Order 방식으로 순회하는 것으로, 데이터가 항상 정렬된 상태로 출력된다.

<Selection Tree>
Selection Tree는 여러 개의 데이터 그룹 중에서 최댓값 또는 최솟값을 빠르게 선택하기 위해 사용되는 완전 이진 트리 구조이다. 본 프로젝트에서는 사원 데이터를 부서별로 구분하여 관리하기 위해 Selection Tree를 구현하였으며, 각 부서의 데이터는 Max Heap을 이용해 연봉 기준 내림차순으로 정렬된다. 따라서 Selection Tree는 여러 부서의 최고 연봉자를 비교하여 전체에서 가장 높은 연봉

을 가진 사원을 빠르게 찾아낼 수 있다.

Selection Tree의 리프 노드는 각 부서를 나타내며, 각각의 리프에는 해당 부서의 사원 정보를 저장하는 EmployeeHeap이 존재한다. 내부 노드는 두 자식 노드의 데이터를 비교하여 연봉이 더 높은 사원의 정보를 저장하고, 이러한 과정이 루트 노드에 도달할 때까지 반복된다. 결과적으로 루트 노드에는 모든 부서 중 최고 연봉자의 정보가 저장된다.

트리의 초기화 과정에서는 8개의 부서(100번부터 800번까지)를 기준으로 리프 노드 8개를 생성하고, 이를 묶는 중간 단계의 내부 노드를 생성하여 완전 이진 트리 형태로 구성한다. 이때 각 리프 노드는 자신의 부서 번호에 해당하는 힙을 초기화하고, 모든 리프 노드가 연결되면 최상위 루트 노드가 설정된다.

삽입 연산은 새로운 사원 데이터를 해당 부서의 리프 노드로 찾아가 힙에 삽입하는 것으로 시작한다. 삽입된 데이터는 UpHeap 과정을 통해 부서 내에서 연봉 내림차순 정렬을 유지하며, 삽입이 완료된 후에는 부모 방향으로 이동하면서 좌우 자식의 연봉을 비교하여 부모 노드의 데이터를 갱신한다. 이 과정을 루트까지 반복함으로써 트리 전체가 자동으로 갱신되고, 항상 루트에는 현재 전체 부서 중 최고 연봉자가 저장된다.

삭제 연산은 루트에 위치한 최고 연봉자를 제거하는 것으로 시작된다. 루트의 데이터가 속한 부서 리프를 찾아 해당 힙의 루트 데이터를 삭제하고, 그 결과 새로 갱신된 최고 연봉자를 leaf에 반영한다. 이후 부모 방향으로 이동하면서 다시 자식 간의 비교를 통해 트리를 재정렬한다. 이렇게 하면 삭제 후에도 트리의 구조와 정렬 상태가 유지된다.

특정 부서의 사원 정보를 출력할 때는 해당 부서의 힙을 복사한 임시 힙을 이용해 연봉 순으로 데이터를 하나씩 꺼내며 출력한다. 이때 원본 힙은 수정되지 않아 부서별 데이터 관리의 안정성이 유지된다.

<Heap>
EmployeeHeap은 사원의 연봉을 기준으로 데이터를 내림차순 정렬하여 관리하기 위한 최대 힙(Max Heap) 구조이다. 이 자료구조는 배열을 기반으로 구현되며, 각 요소는 사원 객체(EmployeeData)의 포인터를 저장한다. 힙의 루트에는 항상 가장 높은 연봉을 가진 사원이 위치하게 되며, 삽입과 삭제 시 자동으로 재정렬이 이

루어진다.

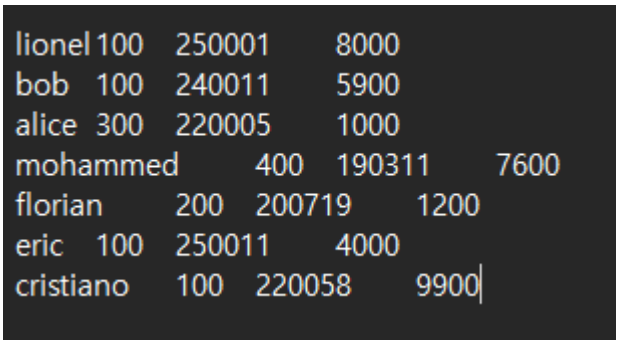
삽입 연산은 새로운 사원 데이터를 힙의 마지막 위치에 추가한 뒤, 부모 노드와 비교하면서 자신의 연봉이 더 클 경우 위치를 교환하는 UpHeap 과정을 수행한다. 이 과정을 반복하면 새로 삽입된 노드는 자신의 올바른 위치로 이동하게 되며, 결과적으로 힙의 구조적 특성과 최대 힙의 조건이 유지된다.

삭제 연산은 루트 노드(가장 높은 연봉자)를 제거한 후, 마지막 노드를 루트로 이동시키고 DownHeap 과정을 수행한다. DownHeap은 부모 노드와 두 자식 노드 중 연봉이 더 높은 자식 노드를 비교하여 부모보다 큰 경우 위치를 교환하며, 이 과정을 반복하여 힙의 정렬 상태를 복원한다.

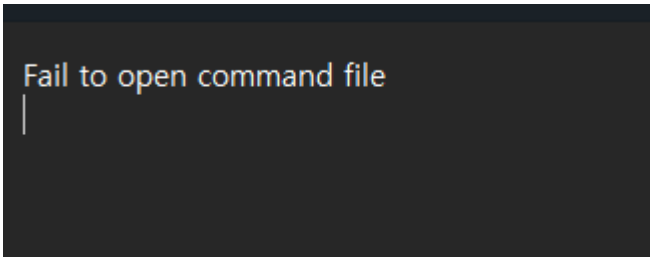
힙이 가득 찬 경우에는 배열의 크기를 두 배로 확장하는 ResizeArray 함수가 호출된다. 이 함수는 새로운 배열을 생성한 뒤 기존 데이터를 모두 복사하고, 기존 배열을 해제하여 메모리를 효율적으로 관리한다.

EmployeeHeap은 각 부서의 데이터를 정렬된 상태로 유지하기 위해 Selection Tree의 리프 노드에서 사용된다. 이를 통해 각 부서의 최고 연봉자를 빠르게 확인할 수 있으며, 삽입과 삭제 연산 모두 평균적으로 O(log N)의 시간 복잡도를 가진다.

4. Result Screen

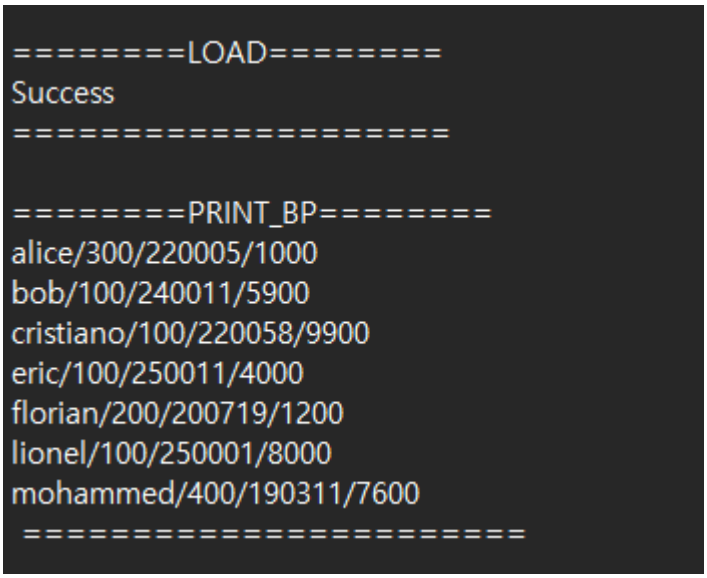


기본적으로 주어진 employee.txt를 사용하였다.

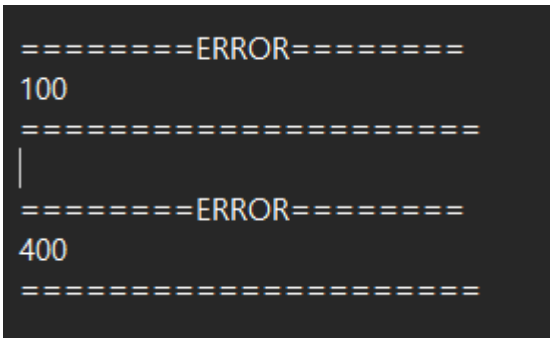


command.txt가 존재하지 않을 때에는 파일 열기를 실패했다는 메시지를 출력한다.

<LOAD>

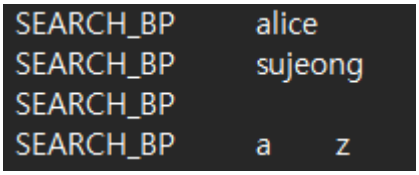


employee.txt 파일이 존재할 때에는 성공적으로 불러왔음을 PRINT_BP 명령어를 통해 확인할 수 있다.



employee.txt 파일이 존재하지 않을 때에는 ERROR 코드를 출력한다.

<SEARCH_BP>



Command에

1) 존재하는 이름을 검색한 경우
2) 존재하지 않는 이름을 검색한 경우
3) 인자를 넣지 않는 경우
4) 범위 검색

을 넣었을 때 예상되는 출력은

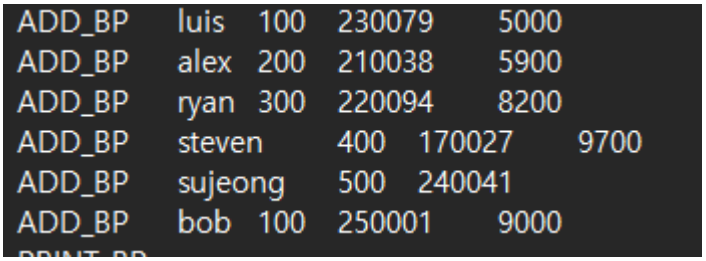
1) 정보를 성공적으로 출력함
2) 에러코드
3) 에러코드
4) 정보를 성공적으로 출력함

이다.

```
========SEARCH_BP========
alice/300/220005/1000
=========================

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

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

========SEARCH_BP========
alice/300/220005/1000
bob/100/240011/5900
cristiano/100/220058/9900
eric/100/250011/4000
florian/200/200719/1200
lionel/100/250001/8000
mohammed/400/190311/7600
=========================
```

로그 파일에서 예상 결과와 동일하게 출력되었음을 확인할 수 있다.

<ADD_BP>



1~4줄(luis부터 steven)은 정상적으로 데이터 추가가 일어나도록 하는 명령어이고

5줄 sujeong에 대한 데이터를 삽입하고자 하는 명령어는 연봉에 대한 정보가 빠져 있다.

6줄은 이미 존재하는 bob에 대한 데이터를 갱신하는 명령어이다.

```
 ========ADD_BP=======
luis/100/230079/5000
 ======================

 ========ADD_BP=======
alex/200/210038/5900
 ======================

 ========ADD_BP=======
ryan/300/220094/8200
 ======================

 ========ADD_BP=======
steven/400/170027/9700
 ======================

========ERROR========
200
====================

 ========ADD_BP=======
bob/100/250001/9000
 ======================

========PRINT_BP========
alex/200/210038/5900
alice/300/220005/1000
bob/100/240011/5900
cristiano/100/220058/9900
eric/100/250011/4000
florian/200/200719/1200
lionel/100/250001/8000
luis/100/230079/5000
mohammed/400/190311/7600
ryan/300/220094/8200
steven/400/170027/9700
 ======================
```

1~4줄에 해당하는 명령어는 정상적으로 처리되었고, 인자가 부족한 경우에는 에러 코드를 추가하며, 이미 존재하는 데이터에 대한 추가는 연봉을 갱신하는 동작을 수행했다. PRINT_BP 명령어를 통해 새로 추가된 데이터에 대해서도 이름 오름차순으로 정렬되었으며, 연봉도 갱신된 것을 확인할 수 있다.

<ADD_ST>

```
ADD_ST  dept_no 100
ADD_ST      dept_no
ADD_ST  name    steven
ADD_ST      namewoojin
```

1) 부서 코드 기준 추가
2) 인자 부족
3) 이름 기준 추가
4) 이름 기준 추가 존재하지 않는 이름)

명령어를 주었을 때

1) 정상 추가
2) 에러코드
3) 정상 추가
4) 에러코드

가 출력되어야 한다.

```
========ADD_ST========
Success
====================

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

========ADD_ST========
Success
====================

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

예상 결과대로 출력됨을 확인할 수 있다.

<PRINT_ST>

앞선 selection tree에 추가가 정상적으로 일어났는지 확인하기 위해 부서 코드 100과 steven이 속한 부서인 400에 대해 출력을 확인했다.

```
PRINT_ST   100
PRINT_ST   400
PRINT_ST
PRINT_ST   800
```

이외에도 인자부족, selection tree가 생성되지 않은 부서에 대한 결과도 확인했다. (예상: 에러코드)

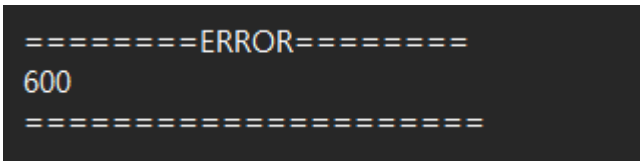
```
========PRINT_ST========
cristiano/100/220058/9900
lionel/100/250001/8000
bob/100/240011/5900
luis/100/230079/5000
eric/100/250011/4000
========================

========PRINT_ST========
steven/400/170027/9700
========================

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

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

정상적으로 연봉 내림차순으로 출력하고 있고, 인자가 부족하거나 selection tree가 생성되지 않은 부서에 대한 명령에 대해서는 에러코드를 출력한다.

<DELETE>

현재 selection tree에는 6명이 저장되어 있으므로 DELETE 명령어를 6번 실행했을 때는 success가 뜨지만 7번째에는 에러코드를 출력해야 한다.

정상적으로 동작함을 확인할 수 있다.



다시 PRINT_ST 명령어를 실행하면 에러코드를 출력한다.

5. Consideration

B+ 트리를 구현하면서 가장 어려웠던 부분은 노드 분할(split) 과정이었다.
데이터 노드와 인덱스 노드가 따로 존재하다 보니, 삽입 시 어느 시점에서 부모 노드에 새로운 키를 추가하고 분할을 수행해야 하는지를 판단하는 것이 쉽지 않았다.
또한 리프 노드들을 이중 연결 리스트로 연결하여 순차 검색과 범위 검색을 가

능하게 만드는 과정에서 포인터 관리의 어려움을 많이 느꼈다.
이 과정을 통해 트리 구조에서 균형을 유지한다는 것이 단순히 노드의 개수를 맞추는 것이 아니라, 전체적인 데이터의 연결성과 참조의 일관성을 유지하는 것이라는 점을 배울 수 있었다.
Manager를 구현할 때에도 여러 예외 상황에 대한 처리를 꼼꼼히 하지 않으면 프로그램이 비정상적으로 종료될 수 있다는 것을 깨달았고, 이를 통해 예외 처리의 중요성을 다시 한 번 느낄 수 있었다.