데이터구조설계 및 실습

프로젝트#2 보고서

학과: 컴퓨터정보공학부

담당교수: 최상호 교수님

분반: 목요일

학번: 2024402055

성명: 박현지

1. Introduction

본 프로젝트는 B+ Tree, Selection Tree, Heap 자료구조를 활용하여 회사 직원 관리 시스템을 구현하는 것을 목표로 한다. 시스템은 employee.txt 파일로부터 직원의 이름(name), 부서코드(dept_no), 사번(ID), 연봉(annual_income) 정보를 읽어들이며, 각 명령어를 통해 데이터를 탐색, 추가, 삭제, 출력한다. 프로그램의 모든 실행 결과는 log.txt 파일에 기록된다.

A. 프로젝트 개요

i. B+ Tree

직원의 이름을 기준으로 오름차순 정렬하여 데이터를 저장한다. 각 노드는 이름(key)을 기준으로 정렬된 map 형태로 구현되며, 탐색 및 삽입 시 노드 분할(split)을 수행한다. 이를 통해 이름 기반 검색(SEACH_BP)과 전체 출력(PRINT_BP) 기능을 지원한다.

ii. Selection Tree

각 부서별(100~800)로 구성된 Heap들을 관리하며, 부서 내 연봉 순위를 관리한다. Selection Tree의 루트 노드는 전체 부서 중 가장 높은 연봉을 가진 직원을 나타내며, 부서별 Heap의 재정렬을 자동으로 반영한다.

iii. Heap

각 부서의 연봉 정보를 저장하는 최대 힙(Max Heap)으로, 삽입(Insert) 시 상향 조정(UpHeap), 삭제(Delete) 시 하향 조정(DownHeap)을 수행한다.

B. 프로그램 구성 및 명령어

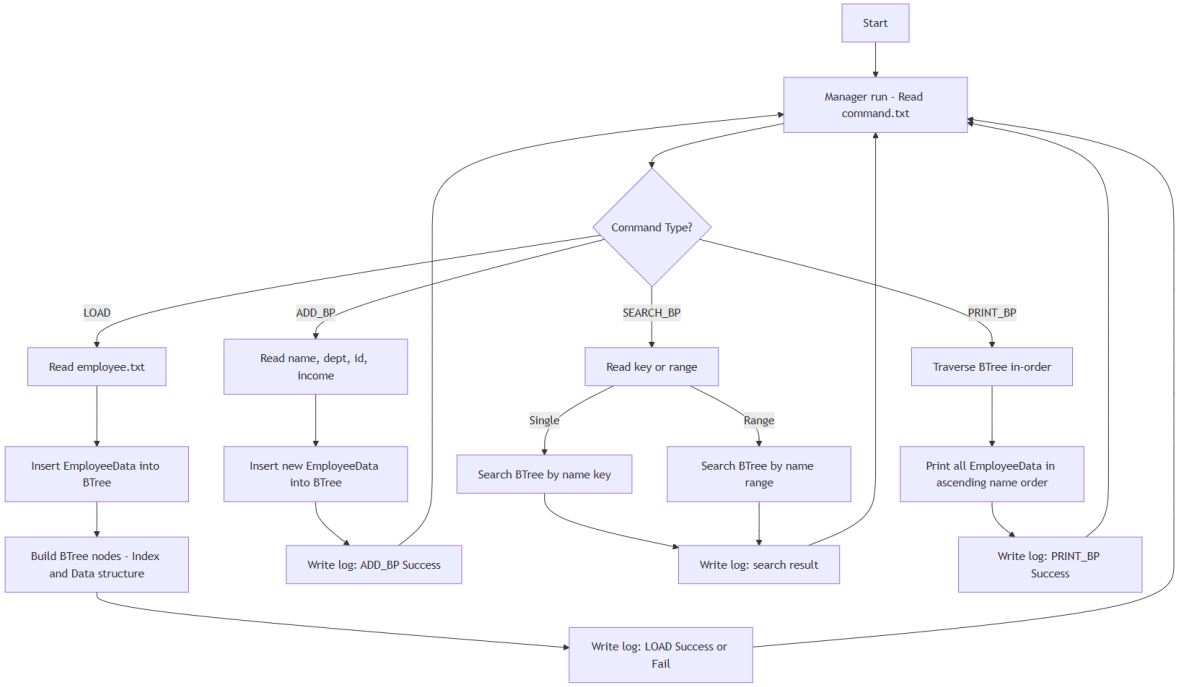
Manager 클래스가 전체 프로그램의 제어를 담당하며, command.txt에 기록된 명령어를 순차적으로 읽어 각 자료구조의 기능을 호출한다. 지원되는 주요 명령어는 다음과 같다.

명령어	기능
LOAD	Employee.txt의 데이터를 읽어 B+ Tree에 삽입
ADD_BP	새 직원 데이터를 직접 추가
SEARCH_BP	이름 또는 범위로 직원 검색
PRINT_BP	B+ Tree의 모든 직원 정보를 출력
ADD_ST	특정 부서 또는 이름으로 Selection Tree에 직원 삽입
PRINT_ST	선택된 부서의 직원을 연봉 내림차순으로 출력
DELETE	Selection Tree의 루트(최고 연봉자) 삭제
EXIT	프로그램 종료 및 메모리 해제

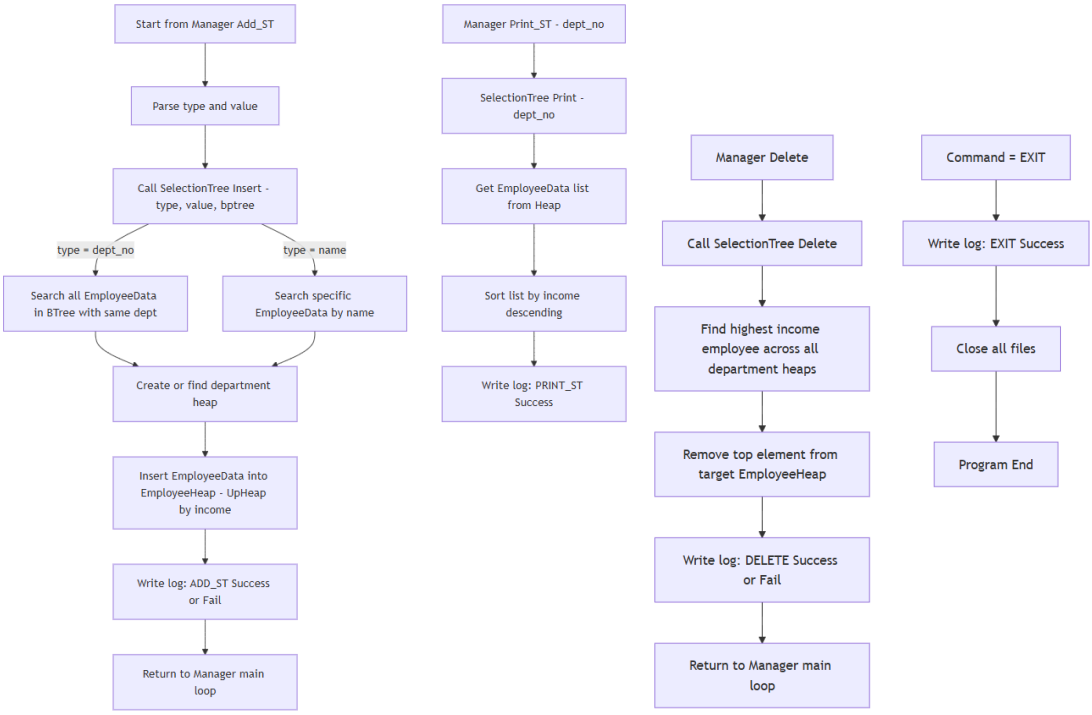
B+ Tree의 효율적인 정렬 및 탐색 기능과, Selection Tree를 통한 실시간 연봉 순위 관리 기능을 결합하여, 대규모 직원 데이터를 구조적으로 관리할 수 있는 시스템을 구현하였다.

2. Flowchart

A. B+Tree 관련 명령어 (LOAD / ADD_BP / SEARCH_BP / PRINT_BP)



B. Selection Tree 관련 명령어(ADD_ST / PRINT_ST) / 삭제 및 종료 (DELETE / EXIT)



3. Algorithm

본 프로젝트는 B+ Tree, Selection Tree, Heap 세 가지 자료구조를 연동하여 작동한다. 각 구조는 서로 다른 기준으로 데이터를 관리하며, Manager 클래스가 모든 명령어를 제어한다. 아래에서는 각 자료구조의 핵심 알고리즘과 동작 방식을 설명한다.

A. **B+ Tree**

i. 구조 및 역할

B+ Tree는 직원의 이름(name)을 기준으로 데이터를 오름차순 정렬하여 저장한다. 각 데이터 노드는 map<string, EmployeeData*> 형태로 이름(key)과 객체(value)를 관리하며, 탐색 효율성을 높이기 위해 인덱스 노드와 데이터 노드로 분리되어 있다.

ii. 삽입 (Insert)

LOAD나 ADD_BP 명령어에서 호출되며 다음 단계에 따라 수행된다.

1. 루트가 비어 있을 경우 새로운 BpTreeDataNode를 생성하고 첫 데이터를 삽입한다.
2. 루트가 존재하면 searchDataNode(name) 함수를 통해 해당 이름이 속할 위치의 데이터 노드를 찾는다.
3. 찾은 노드에 데이터를 삽입하고, 노드의 데이터 개수가 차수(order-1)를 초과하면 분할(split) 수행.
4. 분할 시 중앙 인덱스를 기준으로 왼쪽/오른쪽 노드로 나누고, 부모 노드에 분기 키를 삽입한다.
5. 부모 노드 역시 초과 시에는 재귀적으로 상위 인덱스 분할을 수행하여 균형을 유지한다.

iii. 탐색 (Search / Range Search)

1. **단일 검색 (SEARCH_BP name)**

   인자로 받은 이름을 기반으로 데이터 노드를 탐색 후 일치하는 key를 찾아 해당 직원 정보를 출력한다.

2. **범위 검색 (SEARCH_BP start end)**

   Start ≤ name ≤ end 범위 내의 모든 이름을 순차 탐색한다. 데이터 노드 간 연결 리스트(pNext, pPrev)를 활용하여 순차 접근으로 효율적인 검색을 수행한다.

iv. 출력 (Print)

PRINT_BP 명령어는 루트에서 가장 왼쪽 데이터 노드로 이동한 후, 연결된 모든 노드를 순차적으로 순회하며 데이터를 이름 오름차순으로 출력한다.

B. **Selection Tree**

i. 구조 및 역할

Selection Tree는 부서별 연봉 순위를 관리하기 위한 Max Winner Tree 구조로 구현되었다. 각 부서(100~800)는 Leaf 노드(run)로 대응되며, 각 run 내부에 EmployeeHeap이 존재한다 내부 노드는 자식 노드 중 더 높은 연봉을 가진 직원을 부모로 올려, 루트가 전체 부서 중 최고 연봉 직원을 나타내게 된다.

ii. 삽입 (Insert)

ADD_ST 명령어를 통해 실행되며, 두 가지 방식으로 동작한다.

1. 부서 코드 입력 (ADD_ST dept_no 100)

1) B+ Tree의 모든 직원 데이터를 순회.

2) 해당 부서 코드와 일치하는 직원을 찾으면 run[부서번호]의 Heap에 삽입.

3) 각 삽입마다 Heap이 정렬되고, Selection Tree는 상향 승자 갱신을 수행한다.

2. 이름 입력 (ADD_ST name alex)

1) B+ Tree에서 해당 이름을 탐색.

2) 직원 데이터를 찾아 해당 부서의 Heap에 삽입.

3) Heap 삽입 후 부모 노드 방향으로 승자 전파가 이루어져 루트까지 갱신된다.

iii. 삭제 (Delete)

DELETE 명령 시 루트 노드의 직원(최고 연봉자)을 삭제한다.

1. 루트의 직원 데이터를 확인하고, 해당 부서의 Heap에서 Top 데이터를 삭제.

2. Heap 재정렬 후, run 노드의 대표 직원을 다시 설정.

3. 부모 방향으로 재귀적으로 비교하여 새로운 승자를 갱신한다.

4. 데이터가 모두 비면 루트는 nullptr 상태로 초기화된다.

iv. 출력 (PRINT_ST)

PRINT_ST 명령은 입력된 부서 코드의 Heap을 복사한 후, 모든 데이터를 추출하여 연봉 기준으로 내림차순 정렬한다. 정렬된 결과를 log.txt에 출력한다. 이때 복사본을 사용하기 때문에 원본 Heap의 구조에는 영향을 주지 않는다. 이는 Selection Tree와 Heap 간 데이터 일관성을 유지하기 위한 설계적 고려이다.

C. **Heap (EmployeeHeap)**

i. 구조 및 역할

각 부서별 run은 내부적으로 Max Heap 구조를 사용하여 직원의 연봉을 기

준으로 정렬한다. 배열 기반으로 구현되며, index 1을 루트로 하여 자식 노드는 2*i 와 2*i+1 에 위치한다.

ii. 삽입 (Insert)

1. 새로운 데이터를 배열의 마지막 인덱스에 삽입.
2. 부모 노드의 연봉과 비교하여 더 큰 경우 위치를 교환.
3. 부모 방향으로 재귀적으로 올라가며 정렬을 완료한다.
   이 과정을 UpHeap이라 한다.

iii. 삭제 (Delete)

1. 루트 노드를 제거하고 마지막 노드를 루트 위치로 이동.
2. 두 자식 노드 중 더 큰 연봉과 비교하여 교환.
3. 하향식으로 반복하며 정렬을 복원한다.
   이 과정을 DownHeap이라 한다.

iv. 정렬의 유지

Heap의 삽입과 삭제는 Selection Tree 전체의 승자 계산에 직접 연결되어 있어, 각 run의 Heap 상태가 바뀔 때마다 Selection Tree의 루트 갱신이 즉시 이루어진다.

4. Result Screen

다음은 실제 실행 후 생성된 log.txt의 주요 내용과 각 명령어의 동작을 분석한 결과이다. 모든 명령어는 지정된 출력 포맷에 따라 정상적으로 기록되었으며, 프로그램은 요구된 기능을 순차적으로 수행하였다. 아래는 실행한 command와 그에 따른 실행 결과 log.txt이다.

<table>
<tr><th colspan="2">LOAD - employee.txt 로드</th></tr>
<tr><td>========LOAD========
Success
====================</td><td>LOAD
총 7명의 직원 데이터가 EmployeeData 객체로 생성되어 B+ Tree에 삽입되었다. Tree가 비어 있는 상태에서만 LOAD가 가능하므로, 정상 로딩 시 “Success” 메시지를 출력한다.</td></tr>
<tr><th colspan="2">ADD_BP - 신규 직원 4명 추가</th></tr>
</table>

<table>
<tr>
<td>

```
========ADD_BP========
luis/100/230079/5000
======================
========ADD_BP========
alex/200/210038/5900
======================
========ADD_BP========
ryan/300/220094/8200
======================
========ADD_BP========
steven/400/170027/9700
======================
```

</td>
<td>

```
ADD_BP  luis    100 230079  5000
ADD_BP  alex    200 210038  5900
ADD_BP  ryan    300 220094  8200
ADD_BP  steven  400 170027  9700
```

4명의 새로운 직원(luis, alex, ryan, steven)이 추가되었다. ADD_BP 명령은 이름이 기존 B+ Tree에 없을 경우 새 노드를 생성하고 삽입한다. 모두 유효한 부서 코드(100~800)을 가지고 있으므로, 에러 없이 성공적으로 추가되었다.
추가 이후 B+Tree에는 기존 7명 + 신규 4명 = 총 11명의 데이터가 저장되었다.

</td>
</tr>
<tr>
<td colspan="2">SEARCH_BP - 단일/범위 검색</td>
</tr>
<tr>
<td>

```
========SEARCH_BP========
alex/200/210038/5900
=========================
========SEARCH_BP========
alex/200/210038/5900
alice/300/220005/1000
bob/100/240011/5900
=========================
```

</td>
<td>

```
SEARCH_BP   alex
SEARCH_BP   a   c
```

1. 단일 검색
이름 “alex”를 입력하여 B+ Tree 내에서 정확히 일치하는 데이터를 탐색하였다. 해당 이름을 Key로 가진 노드가 존재하므로, 부서코드/사번/연봉 정보가 출력되었다.
2. 범위 검색
SEARCH_BP a c 명령은 이름이 a 이상 c 이하인 모든 직원을 범위 검색한다. 출력 결과 alex, alice, bob이 포함되며, 알파벳 순서 기준으로 정확히 범위 내의 이름들이 검색되었다. B+ Tree의 데이터 노드 간 연결(pNext)을 활용하여 순차적으로 탐색이 이루어졌음을 확인할 수 있다.

</td>
</tr>
<tr>
<td colspan="2">PRINT_BP - 이름 오름차순 전체 출력</td>
</tr>
</table>

```
========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
========================
```

```
PRINT_BP
```

PRINT_BP 명령은 B+Tree의 모든 직원 데이터를 이름 오름차순으로 출력한다. 출력 결과가 정확히 알파벳 순으로 정렬되어 있으며, 삽입된 모든 직원(11명)이 누락 없이 기록되었다. 이는 데이터 노드 간 연결 리스트를 순차 순회하며 출력하는 알고리즘이 정상적으로 작동함을 의미한다.

ADD_ST - 부서 및 이름 삽입

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

```
ADD_ST  dept_no 100
ADD_ST  name    steven
ADD_ST  name    ryan
ADD_ST  name    alex
```

총 4회의 ADD_ST 명령이 실행되었다.

1. Dept_no 100
   부서코드가 100인 모든 직원 (Cristiano, lionel, bob, luis, eric)이 해당 Heap(run[0])에 삽입됨.
2. Name steven, name ryan, name alex
   각각 이름으로 개별 직원이 추가됨. 각 삽입 시마다 Selection Tree의 상위 노드가 재계산되어 전체 트리의 루트에는 최고 연봉 직원이 유지된다.

모든 삽입이 성공하였음을 "Success"로 그로 확인할 수 있다.

PRINT_ST - 연봉 내림차순 정렬 출력

========PRINT_ST======== cristiano/100/220058/9900 lionel/100/250001/8000 bob/100/240011/5900 luis/100/230079/5000 eric/100/250011/4000 =========================	PRINT_ST 100 부서 코드 100의 직원 목록을 출력한 결과이다. 출력 순서는 연봉 기준으로 내림차순 정렬되어 있으며, 이는 SelectionTree::printEmployeeData() 내에서 수행된다. 따라서 연봉이 높은 순서대로 출력되었음을 확인할 수 있다.
DELETE - 최고 연봉자 제거 및 Heap 재정렬	
========DELETE======== Success ===================== ========PRINT_ST======== lionel/100/250001/8000 bob/100/240011/5900 luis/100/230079/5000 eric/100/250011/4000 =========================	DELETE PRINT_ST 100 Selection Tree의 루트(최고 연봉자)를 삭제하였다. 루트의 직원(Cristiano)이 제거된 후, 해당 부서의 Heap이 재정렬되며 Selection Tree의 루트는 다음으로 높은 연봉을 가진 직원으로 갱신된다. 삭제 이후 동일한 부서(100)의 연봉 순위를 재출력하였다. 그 다음 순위인 lionel이 맨 위에 표시되었다.
EXIT - 프로그램 정상 종료	
========EXIT======== Success ====================	EXIT 모든 명령 실행이 완료된 후, EXIT 명령을 통해 프로그램이 종료되었다. Manager 소멸자에서 객체가 해제되어 메모리 누수가 없도록 마무리하였다.

5. Consideration

본 프로젝트를 통해 B+ Tree와 Selection Tree, Heap을 결합하여 효율적인 데이터 검색과 정렬 시스템을 구현하였다. 구현 과정에서 가장 큰 문제는 메모리 관리와 자료구조 간 데이터 공유였다. B+ Tree가 EmployeeData 객체를 소유하고, Heap과 selection Tree는 이를 참조하는 구조로 변경하여 중복 해제 및 포인터 오류를 해결하였다. 또한 Selection Tree의 Delete 명령어에서 발생한 shallow copy 문제를 방지하기 위해 Heap의 복사를 금지하고 안전한 접근 함수를 추가함으로써 안정성을 확보했다.

이번 과제를 통해 단일 자료구조의 구현보다 여러 구조를 유기적으로 연결하는 설계

가 훨씬 더 복잡하고 중요하다는 점을 체감했다. 각 자료구조의 역할과 소유권을 명확히 구분해야 전체 시스템이 안정적으로 동작할 수 있었다. 특히 B+ Tree의 검색 효율성과 Heap의 정렬 속도를 실제 프로그램 성능 향상으로 연결할 수 있었던 점이 의미 있었다.