Data Structure Project #2

컴퓨터정보공학부

2024402049 정하영

1. Introduction

<개요>

이번 프로젝트는 B+ Tree, Selection Tree, Max Heap 등의 자료구조를 사용하여 직원 데이터 관리 시스템을 설계하고 구현하였다. 사용자는 텍스트 파일에서 직원 정보를 불러와 여러 자료구조에 저장하고, 이름 검색, 범위 검색, 부서별 소득 순위 관리 등의 기능들을 효율적으로 할 수 있다.

<사용된 자료구조>

BpTree (B+ Tree)

직원의 이름을 키로 사용하는 B+ Tree 자료구조이다. 순서를 유지하면서 빠른 검색과 범위를 가진다. 노드는 인덱스 정보를 가지고 있으며, 실제 데이터는 리프 노드에 저장된다. 리프 노드들은 연결 리스트로 이루어져 있어서 순차적으로 접근이 가능하다. Order 값에 따라 노드가 초과되면 자동으로 분할되어 트리의 균형을 유지할 수 있다.

SelectionTree (Selection Tree)

8개 부서(100번부터 800번)의 직원 데이터를 관리하는 토너먼트 형태를 가지고 있는 트리 구조이다. 각 리프 노드는 하나의 부서에 대응하고, 해당 부서 직원들을 Max Heap으로 저장한다. 내부 노드는 자식 노드들 중 소득이 가장 높은 직원을 선택하여 저장한다. 루트 노드에는 항상 전체 직원 중 소득이 가장 높은 사람이 위치한다.

EmployeeHeap (Max Heap)

각 부서의 직원들을 소득을 기준으로 내림차순으로 저장하는 Max Heap이다. 소득이 같으면 이름의 사전 순서로 정렬된다. 동적 배열로 구현되어 용량이 부족하면 자동으로 크기가 두 배로 확장된다. Heap 구조를 유지하기 위해 삽입 할 때에는 UpHeap, 삭제 할 때에는 DownHeap 연산을 사용한다.

<동작 명령어>

LOAD : employee.txt 파일에서 직원 데이터를 읽어 B+ Tree에 저장한다.

ADD_BP : 사용자가 직접 입력한 직원 데이터를 B+ Tree에 추가한다.

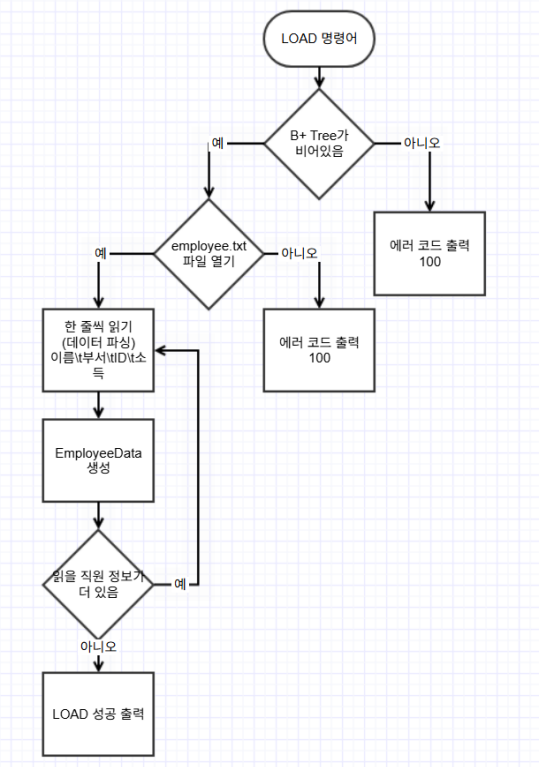
SEARCH_BP : 이름 또는 이름 범위로 B+ Tree에서 직원을 검색한다.

PRINT_BP : B+ Tree에 저장된 모든 직원을 이름순으로 출력한다.

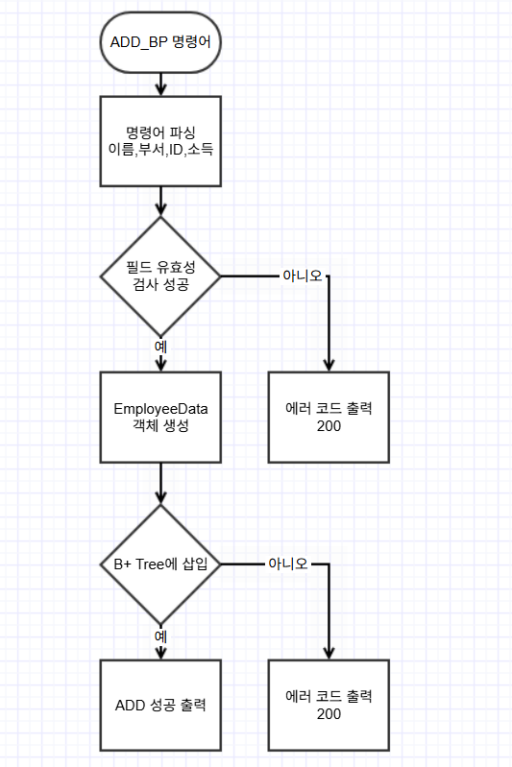
ADD_ST : B+ Tree의 직원 데이터를 부서별로 Selection Tree에 추가한다.
PRINT_ST : 특정 부서의 직원들을 소득순으로 출력한다.
DELETE : Selection Tree의 루트를 삭제한다.
EXIT : 모든 메모리를 해제하고 프로그램을 종료한다.

2. Flowchart

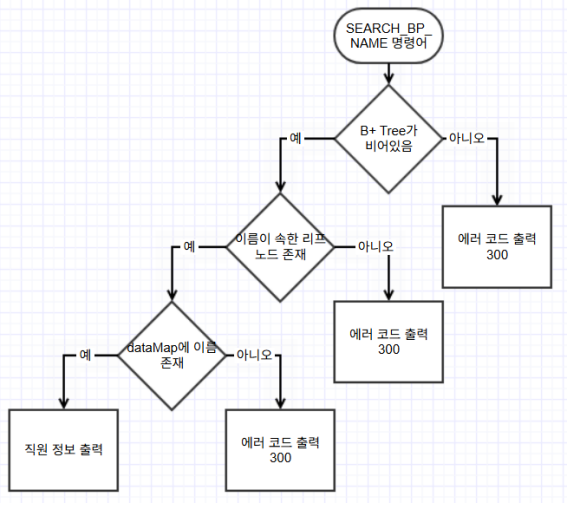
<LOAD 프로세스>



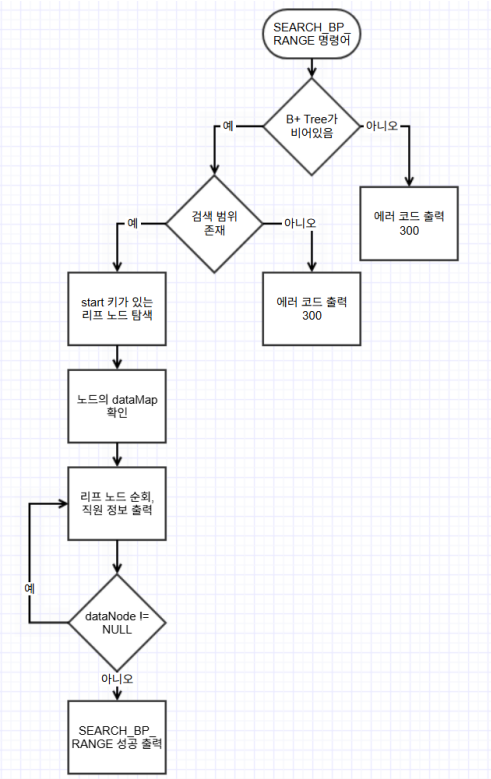
<ADD_BP 프로세스>



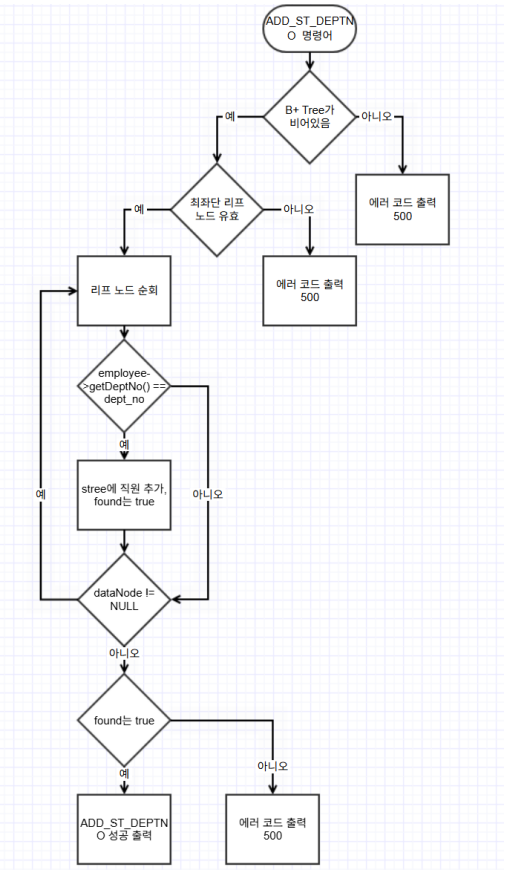
<SEARCH_BP_NAME 프로세스>



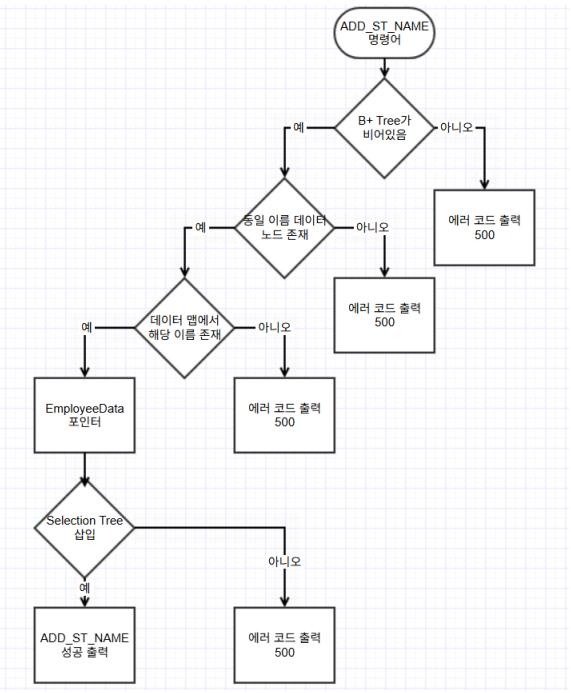
<SEARCH_BP_RANGE 프로세스>



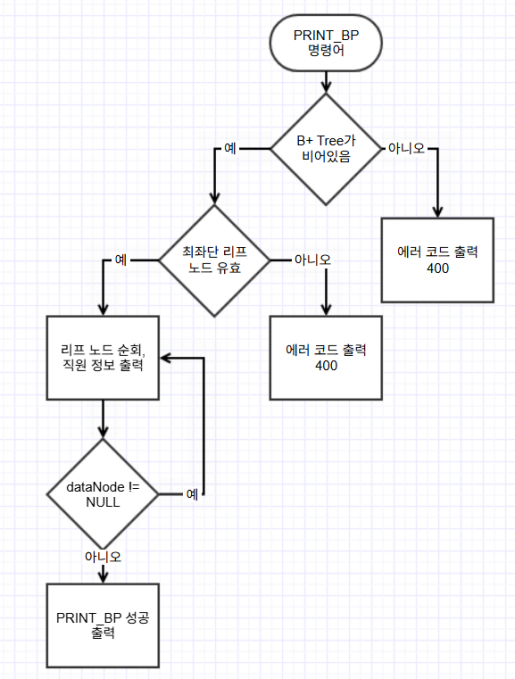
<ADD_ST_DEPTNO 프로세스>



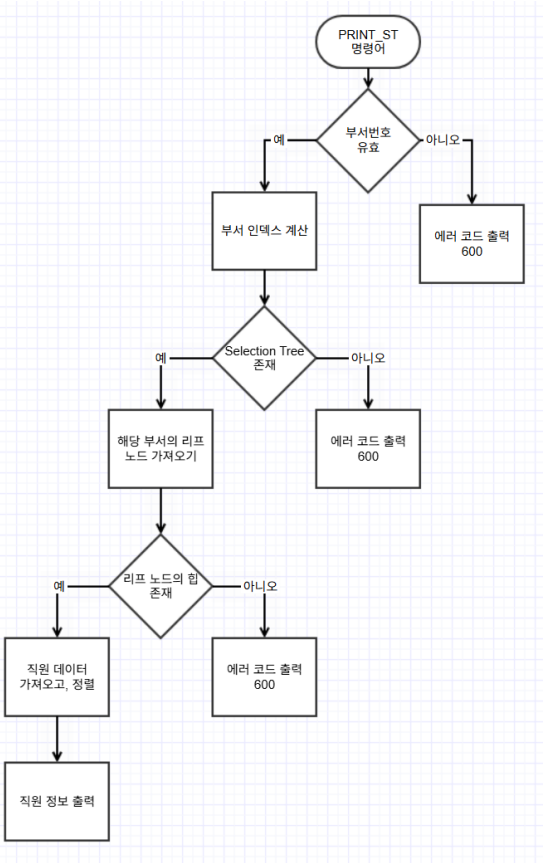
<ADD_ST_NAME 프로세스>



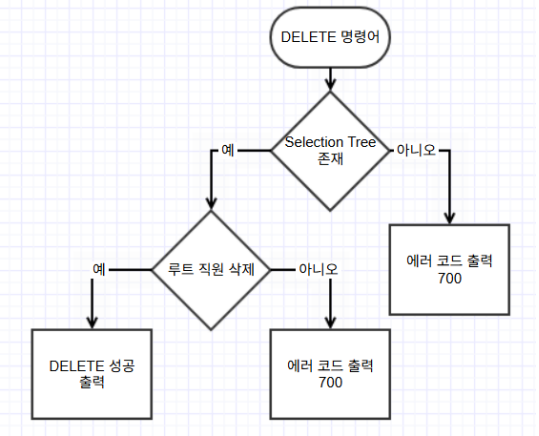
<PRINT_BP 프로세스>



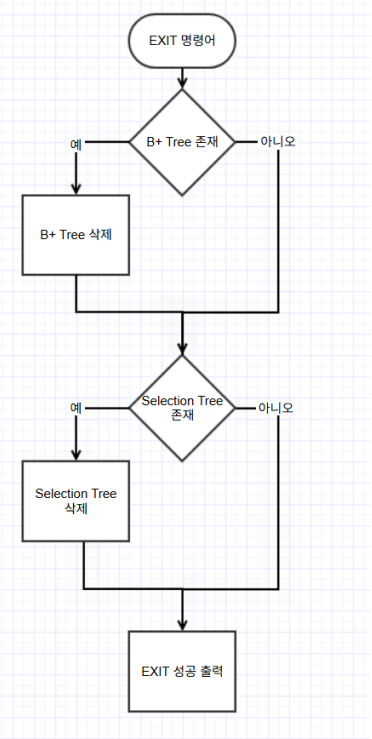
<PRINT_ST 프로세스>



<DELETE 프로세스>



<EXIT 프로세스>



3. Algorithm

<전체 프로그램 구조>

Manager 클래스를 중심으로 BpTree, SelectionTree, EmployeeHeap의 3가지 핵심 자료구조를 관리한다. command.txt 파일에서 명령어를 읽어 처리하며, 모든 실행 결과는 log.txt 파일에 기록된다.

employee.txt → BpTree (LOAD)

BpTree → SelectionTree (ADD_ST)

SelectionTree → 최고 소득자 삭제 (DELETE)

<Manager 알고리즘>

Manager.h, Manager.cpp

과제에서 메인으로 제어하는 클래스이다. 명령어 파일을 입력 받아서 BpTree와 SelectionTree 자료구조를 관리한다. 각 명령 실행 결과와 오류 메시지를 log.txt 파일에 출력한다.

- run(const char* command)

  외부 명령어 파일을 읽어서 한 줄씩 실행한다.

  ① log.txt 파일을 truncate 모드로 열어 기존 로그를 삭제한다.

  ② command.txt 파일을 열고 한 줄씩 읽는다.

  ③ 각 명령어를 파싱하여 적절한 함수를 호출한다.

  ④ 파일 처리가 끝나면 로그 파일을 닫는다.

  시간 복잡도: O(n), n = 명령어 개수

- LOAD() 텍스트 파일의 직원 정보를 불러오는 명령어

  ① B+ Tree가 비어있지 않으면 에러를 출력하고 리턴한다.

  ② 파일을 열고 각 줄을 탭으로 구분하여 파싱한다.

  ③ 이름, 부서번호, ID, 소득 정보를 추출한다.

  ④ EmployeeData 객체를 생성하여 B+ Tree에 삽입한다.

  ⑤ 삽입 성공 시 로그 파일에 기록한다.

  시간 복잡도: O(n log n), n = 직원 수

- ADD_BP(const string& cmd) 직원 데이터를 추가하는 명령어

  ① 명령어 문자열을 탭 기준으로 파싱한다.

  ② 네 개의 필드가 모두 채워져 있는지 검증한다.

  ③ EmployeeData 객체를 생성한다.

  ④ B+ Tree의 Insert 함수를 호출한다.

  ⑤ 삽입 성공 여부에 따라 로그를 기록한다.

시간 복잡도: O(log n)

- SEARCH_BP_NAME(string name) 특정 이름 직원 검색 명령어
  ① B+ Tree가 비어있는지 확인한다.
  ② searchDataNode 함수로 해당 이름이 속한 리프 노드를 찾는다.
  ③ 데이터 맵에서 정확히 일치하는 이름이 있는지 확인한다.
  ④ 직원 정보를 로그 파일에 출력한다.

  시간 복잡도: O(log n)

- SEARCH_BP_RANGE(string start, string end) 이름 범위 내의 직원 검색 명령어
  ① B+ Tree가 비어있거나 범위 내 데이터가 없으면 에러를 출력한다.
  ② start 키가 속한 리프 노드를 찾는다.
  ③ 연결 리스트를 따라 순차적으로 탐색한다.
  ④ 범위 내의 모든 직원 정보를 출력한다.
  ⑤ end를 초과하면 탐색을 중단한다.

  시간 복잡도: O(log n + k), k = 범위 내 결과 개수

- PRINT_BP() B+ Tree 직원 이름순으로 출력하는 명령어
  ① 트리가 비어있는지 확인한다.
  ② 루트에서 시작하여 최좌단 리프 노드로 이동한다.
  ③ 연결 리스트를 따라 모든 리프 노드를 순회한다.
  ④ 각 노드의 모든 직원 정보를 출력한다.

  시간 복잡도: O(n)

- ADD_ST_DEPTNO(int dept_no) 특정 부서의 직원들을 Selection Tree에 추가하는 명령어
  ① B+ Tree가 비어있는지 확인한다.
  ② 제일 왼쪽에 있는 리프 노드로 이동한다.
  ③ 모든 리프 노드를 순회하며 해당 부서 직원을 찾는다.
  ④ Selection Tree의 Insert 함수를 호출하여 추가한다.
  ⑤ 추가되면 성공 메시지를 출력한다.

  시간 복잡도: O(n + k log k), k = 추가되는 직원 수

- ADD_ST_NAME(string name) 특정 이름의 직원을 Selection Tree에 추가하는 명령어
  ① B+ Tree에서 해당 직원을 검색한다.
  ② 직원이 존재하면 Selection Tree에 삽입한다.

③ 삽입 성공 여부를 로그에 기록한다.

시간 복잡도: O(log n)

- PRINT_ST(int dept_no) 특정 부서의 직원들을 소득 순으로 출력하는 명령어
  ① 부서 번호의 유효성을 검증한다.
  ② 해당 부서의 리프 노드와 Heap을 가져온다.
  ③ Heap의 모든 데이터를 벡터로 복사한다.
  ④ 소득 내림차순, 이름 오름차순으로 정렬한다.
  ⑤ 정렬된 직원 정보를 출력한다.

  시간 복잡도: O(k log k), k = 부서 직원 수

- DELETE() Selection Tree의 루트를 삭제하는 명령어
  ① Selection Tree가 비어있는지 확인한다.
  ② 루트 노드의 직원 정보를 확인한다.
  ③ 해당 직원이 속한 부서의 Heap에서 삭제한다.
  ④ updateTree 함수로 트리를 갱신한다.
  ⑤ 삭제 성공 여부를 로그에 기록한다.

  시간 복잡도: O(log h + log d), h = heap 크기, d = tree 깊이

- EXIT() 프로그램을 종료하고 메모리 해제 명령어
  ① 성공 메시지를 로그에 기록한다.
  ② B+ Tree의 소멸자를 호출하여 메모리를 해제한다.
  ③ Selection Tree의 소멸자를 호출하여 메모리를 해제한다.

  시간 복잡도: O(n)

- printErrorCode(int n) 에러 코드를 로그 파일에 기록하는 함수
  시간 복잡도: O(1)
- printSuccessCode(string success) 성공 메시지를 로그 파일에 기록하는 함수
  시간 복잡도: O(1)

<BpTree 알고리즘>

BpTree.h, BpTree.cpp

B+ Tree 자료구조를 구현한 클래스이다. 직원 데이터를 이름 기준으로 정렬해서 저장하고, 효율적인 검색과 범위를 지원한다.

- Insert(EmployeeData* newData)
  새로운 직원 데이터를 B+ Tree에 삽입하는 함수
  ① 트리가 비어있으면 루트를 데이터 노드로 생성한다.
  ② searchDataNode로 적절한 삽입 위치를 찾는다.
  ③ 해당 노드에 데이터를 삽입한다.
  ④ 노드가 order를 초과하면 splitDataNode를 호출한다.

  시간 복잡도**:** O(log n)

- searchDataNode(string name)
  특정 이름이 속하는 리프 노드를 찾는 함수
  ① 루트에서 시작하여 인덱스 노드를 따라 내려간다.
  ② 각 레벨에서 키 값을 비교하여 방향을 결정한다.
  ③ 리프 노드에 도달할 때까지 반복한다.
  ④ 찾은 리프 노드를 반환한다.

  시간 복잡도: O(log n)

- searchRange(string start, string end)
  범위 내에 데이터가 존재하는지 확인하는 함수
  ① start 키가 속한 리프 노드를 찾는다.
  ② 연결 리스트를 따라 순회하며 범위 내 데이터를 확인한다.
  ③ 하나라도 존재하면 true를 반환한다.

  시간 복잡도: O(log n + k)

- splitDataNode(BpTreeNode* pDataNode)
  용량을 초과한 데이터 노드를 분할하는 함수
  ① 새로운 데이터 노드를 생성한다.
  ② 중간 지점을 계산하여 데이터를 분할한다.
  ③ 중간 키를 부모 노드로 승격시킨다.
  ④ 연결 리스트의 링크를 업데이트한다.
  ⑤ 부모 노드도 초과되면 재귀적으로 분할한다.

  시간 복잡도: O(m), m = 노드 내 키 개수

- splitIndexNode(BpTreeNode* pIndexNode)
  용량을 초과한 인덱스 노드를 분할하는 함수
  ① 새로운 인덱스 노드를 생성한다.
  ② 중간 지점을 계산하여 인덱스를 분할한다.

③ 중간 키를 부모로 승격시킨다.
④ 자식 노드들의 부모 포인터를 업데이트한다.
⑤ 부모도 초과되면 재귀적으로 분할한다.

시간 복잡도**:** O(m)

- excessDataNode(BpTreeNode* pDataNode)
  데이터 노드가 order를 초과했는지 확인 함수
  시간 복잡도: O(1)
- excessIndexNode(BpTreeNode* pIndexNode)
  인덱스 노드가 order를 초과했는지 확인 함수
  시간 복잡도: O(1)
- deleteTree(BpTreeNode* node)
  트리의 모든 노드를 재귀적으로 삭제 함수
  ① 노드가 NULL이면 리턴한다.
  ② 인덱스 노드인 경우 모든 자식을 먼저 삭제한다.
  ③ 현재 노드를 삭제한다.

  시간 복잡도: O(n)

BpTreeDataNode.h

B+ Tree의 리프 노드를 나타내는 클래스로, 실제 직원 데이터를 저장하고 연결 리스트를 형성한다.

- 생성자
  prev와 next 포인터를 nullptr로 초기화
  시간 복잡도: O(1)
- 소멸자
  맵에 저장된 모든 EmployeeData 객체 삭제
  시간 복잡도: O(m), m = 노드 내 데이터 개수
- InsertDataMap(string name, EmployeeData* pN)
  데이터 맵에 새로운 직원을 추가하는 함수
  시간 복잡도: O(log m)
- deleteMap(string name)
  데이터 맵에서 특정 직원을 삭제하는 함수
  시간 복잡도: O(log m)
- Getter 함수
  next, prev, dataMap을 반환하는 함수들

시간 복잡도: O(1)

BpTreeIndexNode.h

B+ Tree의 내부 노드를 나타내는 클래스이다. 키와 자식 노드 포인터를 저장해서 트리 탐색을 한다.

- 생성자
  빈 인덱스 맵으로 초기화한다.
  시간 복잡도: O(1)
- 소멸자
  맵을 초기화하지만 자식 노드는 삭제하지 않는다.
  시간 복잡도: O(1)
- insertIndexMap(string key, BpTreeNode* node)
  인덱스 맵에 새로운 키와 자식 노드 포인터를 추가하는 함수
  시간 복잡도: O(log m), m = 노드 내 키 개수
- deleteMap(string key)
  인덱스 맵에서 특정 키를 삭제하는 함수ㅜ
  시간 복잡도: O(log m)
- Getter 함수
  indexMap을 반환하는 함수
  시간 복잡도: O(1)

<SelectionTree 알고리즘>

SelectionTree.h, SelectionTree.cpp

토너먼트 형태의 트리로 8개의 부서의 가장 높은 소득을 가진 직원을 관리하는 자료구조이다.

- setTree()
  8개의 리프 노드로 구성된 완전 이진 트리를 생성하는 함수
  ① 8개의 리프 노드를 생성하고 각각 Heap을 초기화한다.
  ② 4개의 레벨2 노드를 생성하여 리프들을 연결한다.
  ③ 2개의 레벨1 노드를 생성하여 레벨2를 연결한다.
  ④ 루트 노드를 생성하여 레벨1을 연결한다.

  시간 복잡도: O(1)

- Insert(EmployeeData* newData)
  직원을 해당 부서의 Heap에 추가하는 함수

① 부서 번호로 리프 노드 인덱스를 계산한다.
② 해당 Heap에 직원을 삽입한다.
③ Heap의 top을 리프 노드에 저장한다.
④ updateTree를 호출하여 트리① 를 갱신한다.

시간 복잡도: O(log h + log d)

- Delete()
  루트의 소득이 가장 높은 직원을 삭제하는 함수
  ① 루트가 비어있는지 확인한다.
  ② 해당 직원이 속한 부서의 Heap을 찾는다.
  ③ Heap에서 top을 삭제한다.
  ④ 새로운 top을 리프에 저장하고 updateTree를 호출한다.

  시간 복잡도: O(log h + log d)

- updateTree(SelectionTreeNode* leaf)
  리프에서 루트까지 승자를 갱신하는 함수
  ① 리프의 부모부터 시작한다.
  ② 양쪽 자식의 소득을 비교하여 높은 쪽을 선택한다.
  ③ 현재 노드에 승자를 저장한다.
  ④ 루트에 도달할 때까지 반복한다.

  시간 복잡도: O(log d), d = 트리 깊이(3)

- printEmployeeData(int dept_no)
  특정 부서의 직원들을 정렬해서 출력하는 함수
  ① 부서 번호로 리프 노드를 찾는다.
  ② Heap의 모든 데이터를 벡터로 복사한다.
  ③ 소득과 이름 기준으로 정렬한다.

  시간 복잡도: O(k log k)

<Employee 알고리즘>

EmployeeHeap.h, EmployeeHeap.cpp

Max Heap 자료구조로 각 부서의 직원들을 소득을 기준으로 관리한다.

- Insert(EmployeeData* data)
  새로운 직원을 Heap에 추가하는 함수
  ① 데이터가 유효한지 확인한다.

② 용량이 부족하면 ResizeArray를 호출한다.
③ 배열의 끝에 데이터를 추가한다.
④ UpHeap을 호출하여 Heap 속성을 복구한다.

시간 복잡도: O(log n)

- Delete()
  Heap의 루트를 삭제하는 함수
  ① Heap이 비어있는지 확인한다.
  ② 루트를 마지막 원소로 교체한다.
  ③ 크기를 감소시킨다.
  ④ DownHeap을 호출하여 Heap 속성을 복구한다.

  시간 복잡도: O(log n)

- UpHeap(int index)
  삽입 후 Heap 속성을 복구하는 함수
  ① 인덱스가 1 이하면 리턴한다.
  ② 부모 노드와 비교한다.
  ③ 소득이 크거나 같으면서 이름이 앞서면 교환한다.
  ④ 재귀적으로 위로 이동한다.

  시간 복잡도: O(log n)

- DownHeap(int index)
  삭제 후 Heap 속성을 복구하는 함수
  ① 자식이 없으면 리턴한다.
  ② 왼쪽과 오른쪽 자식 중 큰 값을 찾는다.
  ③ 현재 노드보다 크면 교환한다.
  ④ 재귀적으로 아래로 이동한다.

  시간 복잡도: O(log n)

- ResizeArray()
  배열 용량이 부족할 때 크기를 두 배로 확장하는 함수
  ① 새로운 용량을 계산한다.
  ② 새 배열을 할당한다.
  ③ 기존 데이터를 복사한다.
  ④ 이전 배열을 삭제한다.

  시간 복잡도: O(n)

- Top()
  Heap의 최댓값을 반환하는 함수
  시간 복잡도: O(1)
- IsEmpty()
  Heap이 비어있는지 확인하는 함수
  시간 복잡도: O(1)

EmployeeData.h

직원의 정보를 저장하는 클래스이다.

- 생성자
  이름, 부서 번호, ID, 소득을 초기화한다.
  시간 복잡도: O(1)
- print(ofstream& fout)
  직원 정보를 파일 스트림에 출력하는 함수
  시간 복잡도: O(1)
- Getter/Setter 함수
  각 멤버 변수에 접근하는 함수
  시간 복잡도: O(1)
- 비교 연산자
  이름을 기준으로 직원들을 비교하는 연산자들
  시간 복잡도: O(1)

4. Result Screen

LOAD 성공 출력

```
1  ========LOAD========
2  Success
3  ====================
```

ADD_BP 성공 출력

```
5  ========ADD_BP========
6  luis/100/230079/5000
7  ======================
8  ========ADD_BP========
9  alex/200/210038/5900
10 ======================
11 ========ADD_BP========
12 ryan/300/220094/8200
13 ======================
14 ========ADD_BP========
15 steven/400/170027/9700
16 ======================
```

SEARCH_BP 이름으로 검색 성공 출력

```
17  ========SEARCH_BP========
18  alex/200/210038/5900
19  =========================
```

SEARCH_BP c g 범위로 검색 성공 출력

```
21  ========SEARCH_BP========
22  cristiano/100/220058/9900
23  eric/100/250011/4000
24  florian/200/200719/1200
25  =========================
```

PRINT_BP 성공 출력

```
26  ========PRINT_BP========
27  alex/200/210038/5900
28  alice/300/220005/1000
29  bob/100/240011/5900
30  cristiano/100/220058/9900
31  eric/100/250011/4000
32  florian/200/200719/1200
33  lionel/100/250001/8000
34  luis/100/230079/5000
35  mohammed/400/190311/7600
36  ryan/300/220094/8200
37  steven/400/170027/9700
38  =========================
```

ADD_ST 성공 출력

```
39  ========ADD_ST========
40  Success
41  ====================
42
43  ========ADD_ST========
44  Success
45  ====================
46
47  ========ADD_ST========
48  Success
49  ====================
50
51  ========ADD_ST========
52  Success
53  ====================
```

PRINT_ST 부서번호 100인 직원 성공 출력

```
55  ========PRINT_ST========
56  cristiano/100/220058/9900
57  lionel/100/250001/8000
58  bob/100/240011/5900
59  luis/100/230079/5000
60  eric/100/250011/4000
61  ========================
62  ========DELETE========
```

DELETE 성공 출력

```
62  ========DELETE========
63  Success
64  ====================
```

EXIT 성공 출력

```
72  ========EXIT========
73  Success
74  ====================
```

5. Consideration

이번 프로젝트는 B+ Tree, Selection Tree, Max Heap 등의 자료구조를 사용해서 직원 관리 시스템을 구현했다. 사용자는 명령어 파일을 통해서 데이터 로드, 추가, 검색, 부서별 관리, 삭제 등을 수행할 수 있다. 모든 결과는 log.txt라는 파일에 기록이 된다.

<구현 과정 어려움 & 해결 방법>

- B+ Tree 노드 분할의 복잡성

  B+ Tree에서 데이터나 인덱스 노드가 order을 초과할 때 분할하는 것을 구현하는 것이 복잡했다. 데이터 노드를 분할할 때 중간 키를 부모로 승격시키고, 리프 노드들 사이의 연결 리스트를 유지하면서 부모와 자식 관계를 정확하게 업데이트 하는 과정이 어려웠다. 루트 분할할 때에는 새로운 루트를 생성해야 했다.

  → splitDataNode와 splitIndexNode를 별도의 함수로 사용해서 각각 분할하는 과정을 명확하게 구현했다. 중간 지점 계산은 과제 제안서에 있던대로 ceil((order – 1) / 2.0) + 1 공식을 사용했고, 벡터를 사용해서 맵의 요소들을 순서대로 접근할 수 있었다. 분할할 때에는 prev/next 링크를 먼저 업데이트한 후에 부모 연결을 처리하도록 했다. 재귀적으로 부모 노드의 초과 여부를 확인해서 분할을 연쇄적으로도 처리할 수 있었다.

- Selection Tree와 Heap의 동기화 과정

  Selection Tree의 각 리프 노드가 Max Heap을 가지고 있기 때문에 Heap에 데이터를 추가하거나 삭제하라 때 트리의 모든 상위 노드를 갱신해야 한다. 특

히 루트에 있는 소득이 가장 높은 직원을 삭제한 후, 해당 부서의 Heap을 찾아서 삭제하고 트리를 다시 업데이트 해야했다.

→ updateTree 함수를 구현해서 리프에서 루트까지 부모 노드들을 순차적으로 갱신하도록 했다. 각 노드는 왼쪽과 오른쪽 자식 중 소득이 높은 직원을 선택했고, 소득이 같은 경우에는 이름의 사전순으로 비교하도록 했다. 이 방식을 통해 부서별 Heap과 Selection Tree의 상태가 항상 일치하도록 유지할 수 있었다.

- Heap 정렬 기준 처리

  Max Heap을 구현할 때 단순하게 소득만 비교하면, 동일한 소득을 가진 직원들 간의 순서가 불안정했다. Heap의 상위 노드가 갑자기 바뀌거나 출력할 때 정렬이 틀어지는 상황이 생겼다.

  → compare 조건을 수정해서 소득이 같을 경우에는 이름의 사전순으로 비교하도록 했다. 소득이 높을수록 우선이 되었고, 소득이 같은 경우에는 이름이 사전순으로 먼저 올수록 우선이 되도록 우선순위 규칙을 적용했다. 이 비교 로직을 UpHeap과 DownHeap 양쪽에 동일하게 적용했다.

- 트리 간의 데이터 이동의 비효율성 파악

  ADD_ST 명령을 수행할 때, 처음 코드 작성 시, B+ Tree의 모든 데이터를 순회하면서 특정 부서의 직원을 Selection Tree로 옮기도록 로직을 짰다. 그래서 전체 트리를 순회하게 되었고, 속도가 느리고 불필요한 비교가 생겼다.

  → 가장 왼쪽에 있는 리프 노드부터 시작해서 리프 노드 사이의 연결 리스트를 순회하는 방식으로 코드 로직을 변경했다. B+ Tree 전체 데이터를 탐색할 때 중복 접근을 막을 수 있었고, O(n) 시간 내에 부서별 데이터를 관리할 수 있었다. Selection Tree에 삽입할 때에는 Heap 내부에서 자동으로 정렬되었기 때문에 추가적인 정렬 과정이 필요하지 않았다.