REPORT

광운대학교
KwangWoon University

과목명 | 데이터구조설계

담당교수 | 최상호

학과 | 컴퓨터정보공학부

학년 | 4학년

학번 | 2020202066

이름 | 장지호

제출일 | 2025-11-06

1. Introduction

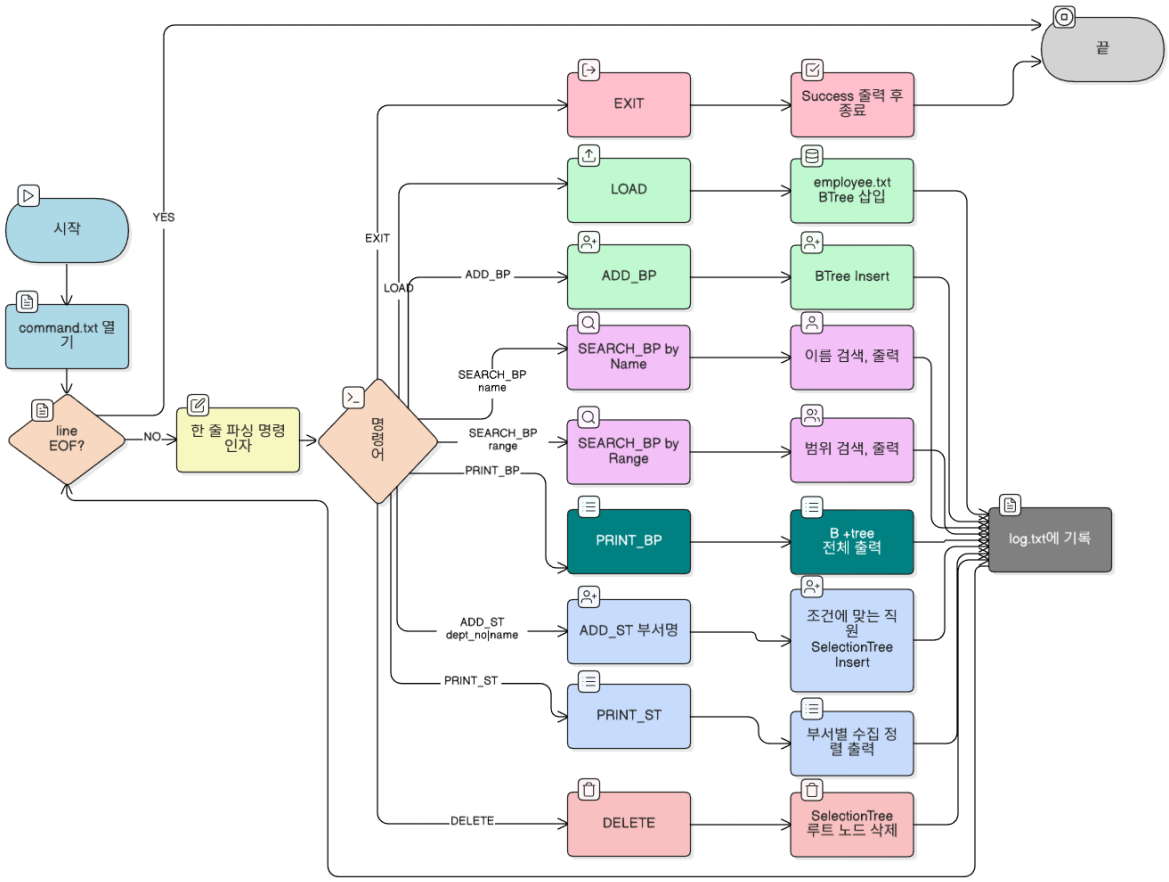
본 프로젝트는 B+ Tree, Selection Tree, Heap 과 같은 자료구조를 구현하여 회사 직원 관리에 이용할 수 있는 프로그램을 구현하였다. 직원의 이름, 부서 번호, ID, 연봉을 저장 및 관리한다.

B+ Tree를 이용해서 직원의 정보를 이름을 기준으로 정렬(오름차순)하여 빠른 검색과 특정 범위내 검색을 구현하였다. 그리고 Selection Tree와 (Max)Heap을 이용해서 연봉을 기준으로 Max Winner Tree를 구현하여 전체, 부서별 연봉 순위를 확인할 수 있다.

command.txt에 작성된 명령어를 순차적으로 실행하며 각 명령어에 따른 결과를 log.txt에 기록한다.

2. Flowchart

Manager의 flowchart



Manager는 프로그램 전체의 흐름을 제어합니다. command.txt를 읽어 해당 명령어에 맞는 함수를 실행하고 파일이 끝날 때까지 명령어를 parsing해서 해당 명령어를 실행한다. 모든 명령어의 결과는 log.txt에 저장하고 만약 명령어 실행에 실패할 경우 에러 코드를

기록한다. 본 프로젝트에서 구현한 명령어는 아래와 같다.

- LOAD는 employee.txt파일을 읽어 B+ Tree를 구축한다. ERR CODE 100

- ADD_BP는 (이름, 부서 번호, id, 연봉)을 입력 받아 B+ Tree에 사원 정보를 추가한다. 만약, 이름이 같은 사원 정보가 이미 존재한다면, 연봉만 업데이트 한다. ERR CODE 200

- SEARCH_BP는 B+ Tree에서 특정 이름을 검색하거나 범위를 검색한다. ERR CODE 300

- PRINT_BP는 B+ Tree에 이름순으로 정렬된 전체 사원 정보를 출력한다. ERR CODE 400

- ADD_ST는 부서 번호나 이름을 기준으로 B+ Tree에서 사원 정보를 연봉순으로 Selection Tree 및 Max Heap에 저장한다. ERR CODE 500

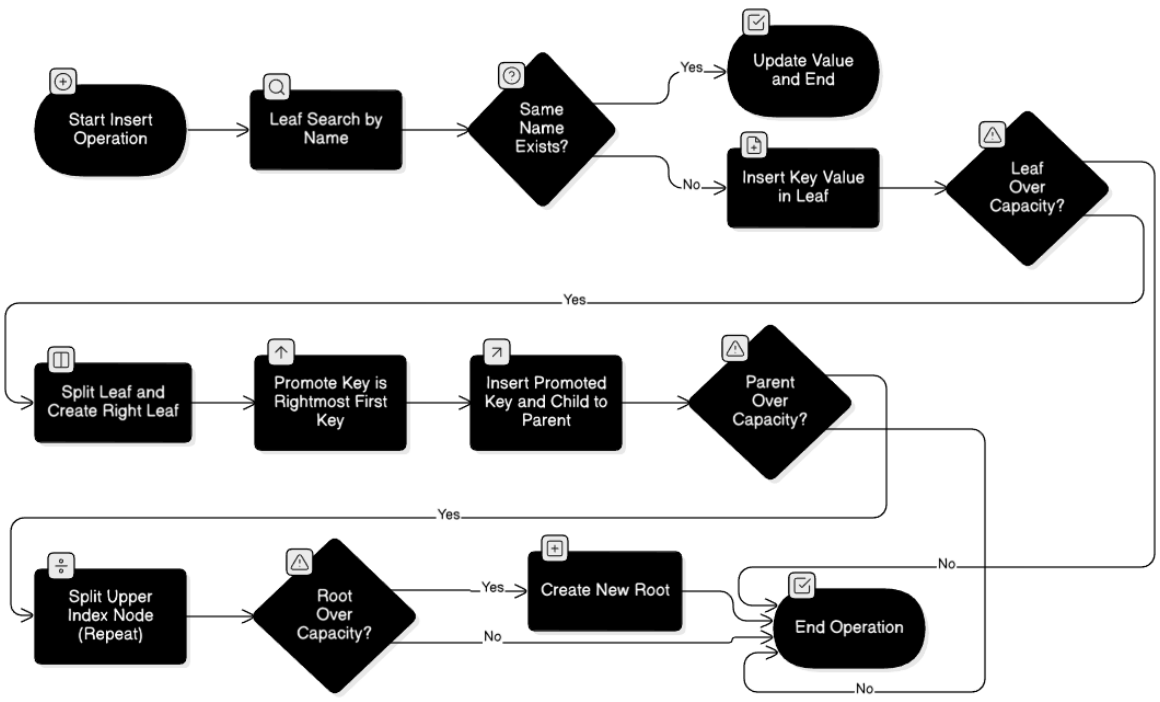
- PRINT_ST는 Max Heap에서 부서 번호에 해당하는 모든 사원 정보를 정렬(이름 오름차순)해 출력한다. ERR CODE 600

- DELETE는 Selection Tree에서 root node(가장 연봉이 높은 직원)을 제거하고 heap을 재정렬한다. (Max Heap 상태를 유지) ERR CODE 700

- EXIT는 프로그램을 종료하고 모든 메모리를 해제한다. ERR CODE 800

잘못된 명령어의 경우 ERR CODE 800

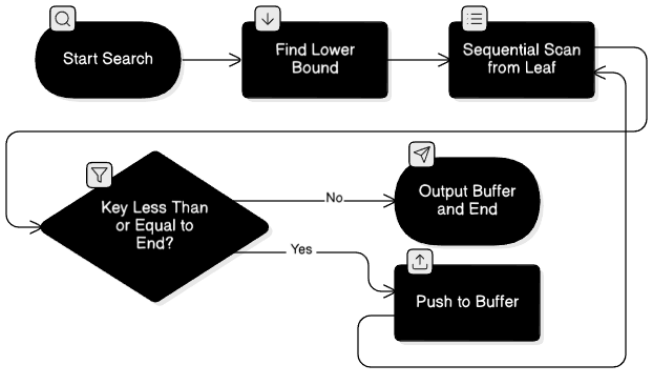
B+ Tree Insert(split)



B+ Tree에서 사원 정보(이름, 부서 번호, 사번, 연봉)이 추가되는 경우에 key는 name으로 하고 노드를 생성한다. Root node부터 시작하여 key ≤ name을 만족하는 마지막 childe node 로 이동한다. 이때, 도착한 leaf node가 삽입 대상인 node이다. Leaf node의 data map에서 name을 검색한다. 이때 동일한 name이 있는 경우에 income만 갱신하고 이름 및 사번(id)과 부서번호(dept_no)는 유지한다. Data map에 (key: name, value: ptr)을 삽입한 후에 용량을 초과하는 지 확인한다. 만약 leaf node의 key 수 ≥ 차수(order)이면 split leaf를 진행한다.

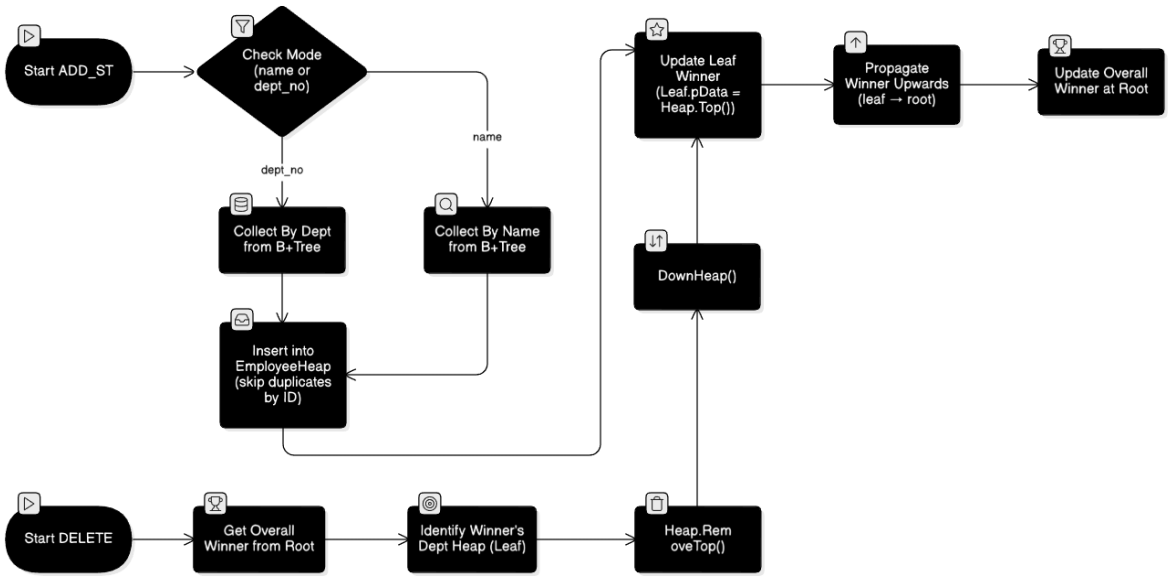
Split Leaf는 차수를 유지하기 위한 방법이다. leaf를 좌/우 로 분할하고, 오른쪽 leaf에 오른쪽 절반에 해당하는 key, value 값을 이동한다. 그리고 leaf간 prev, next 연결을 새롭게 갱신한다. 분할 후, 오른쪽 leaf의 첫 번째 key를 parent index node로 승격시킨다. 이때 parent가 또 차수를 초과하면 상위로 split을 반복하며 필요시, 새로운 root를 생성한다.

B+ Tree range search



B+ Tree에서의 범위 검색은 시작(start), 끝(end)을 입력 받는다. Index node까지 내려가 start가 될 leaf node를 찾아 해당 leaf의 data map에서 lower bound부터 순차적으로 검색한다. 만약 leaf의 끝까지 이동한 경우 next를 따라 다음 leaf로 이동한다. 이 과정에서 key가 end이하인 동안에 buffer에 추가하고 key ≥ end가 되면 검색을 종료한다.

Selection Tree insert, delete



ADD_ST(insertion)는 부서 번호(dept_no) 또는 이름(name)을 받는다. 부서 번호라면 해당 부서의 모든 사원을, 이름이라면 해당 사원 한 명을 B+ Tree에서 조회한 뒤 Selection Tree의 해당 부서 EmployeeHeap으로 복사하여 삽입한다. 이때 heap 삽입 과정에서 동일한 ID가 이미 존재하면(같은 부서 run) 중복을 건너뛴다. 삽입이 끝나면 leaf node의 winner 값(Leaf.pData)을 Heap.Top()으로 갱신하고, winner를 parent로 전파하여 leaf→root 경로의 internal node 및 root의 winner 정보를 갱신한다. Max-Heap은 UpHeap으로 보장되며, Selection Tree의 전역 최상위는 root winner에 반영된다.

DELETE는 root node에서 그 사원의 부서(run)를 식별하여 해당 EmployeeHeap에서 RemoveTop()을 수행한다. 이후 DownHeap()으로 heap을 재정렬하고 leaf의 winner를 Heap.Top()으로 갱신한다. 마지막으로 이 승자를 leaf→root 방향으로 전파해 경로상의 모든 internal node 및 root의 winner를 갱신한다.

3. Algorithm (pseudo code)

B+ Tree

```
function searchDataNode(name):
    node = root
    if node == null: return null

    while node is IndexNode:
        next = node.mostLeftChild
        for (key, child) in node.indexMap (ascending):
            if name >= key:
                next = child
            else:
                break
        node = next
    return node  // Data(leaf) node
```

root부터 시작해서 index node의 key를 순회하면서 name ≥ key를 만족하는 가자 오른쪽 childe node로 이동한다. 어떤 key도 만족하지 않으면 most left child로 이동, leaf에 도착할 때까지 반복한다. 복잡도는 O(log n)

```
function Insert(newData):
    if newData == null: return false

    if root == null:
        leaf = new DataNode()
        leaf.dataMap.insert(newData.name, newData)
        root = leaf
        return true

    leaf = searchDataNode(newData.name)  // DataNode
    iter = leaf.dataMap.find(newData.name)

    if iter exists:
        iter.value.income = newData.income  // 연봉만 갱신
        delete newData
        return true

    leaf.dataMap.insert(newData.name, newData)

    if leaf.dataMap.size >= order:
        splitDataNode(leaf)
    return true
```

searchDataNode(name)로 leaf 결정, 해당 leaf의 data map에서 name을 검색(이미 있으면 income만 갱신, 없으면 (name, ptr)삽입한다. Leaf의 크기 ≥ order면 splitDataNode 호출한다. 복잡도는 O(log N) (탐색, 삽입)

```
function splitDataNode(leaf):
    keys = leaf.dataMap.keys() (ascending)
    total = keys.size
    mid   = total / 2

    right = new DataNode()
    // move right half to 'right' node
    for i in [mid .. total-1]:
        move (keys[i], value) from leaf.dataMap to right.dataMap

    // fix leaf links
    right.next = leaf.next
    if leaf.next: leaf.next.prev = right
    leaf.next = right
    right.prev = leaf

    promoteKey = firstKey(right.dataMap)

    parent = leaf.parent
    if parent == null:
        root = new IndexNode()
        root.mostLeftChild = leaf
        leaf.parent = root
        root.indexMap.insert(promoteKey, right)
```

```
else:
    parent.indexMap.insert(promoteKey, right)
    right.parent = parent
    if parent.indexMap.size >= order:
        splitIndexNode(parent)
```

전체 원소 수의 절반(mid)기준으로 오른쪽 절반을 새로운 leaf로 이동한 후 prev, next 연결 갱신으로 leaf가 같은 level에 있을 수 있도록 한다. 승격시킬 key = 새(오른쪽) leaf의 첫번째 key, parent index에 삽입한다. Parent도 order를 초과하는 경우에는 splitIndexNode를 재귀. 복잡도는 O(order) (이동/삽입), 전파는 높이 O(log N)

```
function splitIndexNode(index):
    // flatten (keys, children)
    keys = index.indexMap.keys() (ascending)
    children = [ index.mostLeftChild ]
    for (k, child) in index.indexMap: children.push(chil
d)

    totalChildren = children.size
    leftCount = (totalChildren + 1) / 2
    if leftCount < 1: leftCount = 1

    promoteKey = keys[leftCount - 1]
    rightMostLeft = children[leftCount]

    // rebuild left(index)
    index.indexMap.clear()
    index.mostLeftChild = children[0]
    if children[0]: children[0].parent = index
    for i in [0 .. leftCount-2]:
        index.indexMap.insert(keys[i], children[i+1])
        if children[i+1]: children[i+1].parent = index
```

->

```
// build 'right' index
right = new IndexNode()
right.mostLeftChild = rightMostLeft
if rightMostLeft: rightMostLeft.parent = right
for i in [leftCount .. keys.size-1]:
    right.indexMap.insert(keys[i], children[i+1])
    if children[i+1]: children[i+1].parent = right

parent = index.parent
if parent == null:
    root = new IndexNode()
    root.mostLeftChild = index
    index.parent = root
    root.indexMap.insert(promoteKey, right)
    right.parent = root
else:
    parent.indexMap.insert(promoteKey, right)
    right.parent = parent
    if parent.indexMap.size >= order:
        splitIndexNode(parent)
```

(key, childe)를 평탄화하기 위해 좌/우로 분할한다. 승격될 key는 좌측에 남는 마지막 key이다. Childe의 parent pointer를 새로 연결하고 만약 parent가 없다면 새로운 root를 생성한다. 마찬가지로 parent가 order를 넘기면 재귀적으로 split한다.

복잡도: O(order), 전파 O(log N)

```
function collectRange(start, end, buffer):
    if start > end: return
    leaf = searchDataNode(start)
    if leaf == null: return

    firstLeaf = true
    node = leaf
    while node != null:
        dataMap = node.dataMap
        it = (firstLeaf) ? dataMap.lower_bound(start) : dataMap.begin()
        for (; it != dataMap.end(); ++it):
            if it.key > end: return
            buffer.push(it.value)
        firstLeaf = false
        node = node.next
```

searchDataNode(start)를 첫 leaf로 설정, 첫 leaf는 start부터 순회하고 이루 leaf는 begin 부터 순회한다. key>end면 즉시 종료, 아니면 buffer에 push. next 포인터로 다음 리프 이동 반복한다. 복잡도: O(log N + K) -> O(log N)

EmployeeHeap (Max Heap)

배열 기반 Heap은 완전 이진 트리를 배열로 구현한 것이며 노드별로 다른 index를 가진다. i번째 노드에서 다른 노드의 index는 다음과 같다.

Left child : 2*i + 1, right child : 2*I + 2, parent : (i – 1) / 2

```
function Insert(data):
    if data == null: return
    if ContainsById(data.id): delete data; return

    if data_num + 1 >= capacity: ResizeArray()
    heap[++data_num] = data
    UpHeap(data_num)
```

ContainsById(id)면 새 포인터 delete 후 삽입 스킵(중복 방지)

배열 끝에 삽입 → UpHeap으로 부모와 비교/교환

복잡도: O(log n)

```
function ContainsById(id):
    for i in [1 .. data_num]:
        if heap[i] && heap[i].id == id: return true
    return false
```

heap[1..data_num] 선형 스캔으로 동일한 id의 존재 여부 반환

복잡도: O(n) (부서별 run 크기 기준)

```
function UpHeap(i):
    while i > 1:
        p = i / 2
        if IsLeftGreater(heap[i], heap[p]):
            swap(heap[i], heap[p])
            i = p
        else:
            break
```

부모와 IsLeftGreater 비교, 크면 스왑하며 위로 이동한다.

복잡도: O(log n)

```
function DownHeap(i):
    while i * 2 <= data_num:
        L = i * 2; R = L + 1
        largest = L
        if R <= data_num && IsLeftGreater(heap[R], heap[L]): largest = R
        if IsLeftGreater(heap[largest], heap[i]):
            swap(heap[i], heap[largest])
            i = largest
        else:
            break
```

두 자식 중 더 큰 쪽과 비교후 스왑하며 아래로 이동한다.

복잡도: O(log n)

Selection Tree

```
function Insert(newData):
    if newData == null: return false
    idx = getRunIndex(newData.dept_no)  // 100..800 → 0..7
    if idx < 0: return false

    leaf = run[idx]
    if leaf.heap == null: leaf.HeapInit()
    leaf.heap.Insert(newData)  // 중복 ID는 내부에서 무시/삭제

    leaf.pData = leaf.heap.Top()
    updateWinners(leaf.parent)
    return true
```

대상 run(부서)의 heap에 삽입(중복 ID 방지) – command로 Heap중복이 발생하는 경우 있음(Consideration에서 추가 작성)

leaf.pData = heap.Top()

updateWinners(leaf.parent)로 leaf→root 승자 전파

복잡도: 힙 O(log n) + 전파 O(h)(트리 높이)

```
function Delete():
    if root == null or root.pData == null: return false
    target = root.pData
    idx = getRunIndex(target.dept_no)
    if idx < 0: return false

    leaf = run[idx]
    if leaf == null or leaf.heap == null or leaf.heap.IsEmpty(): return false

    removed = leaf.heap.RemoveTop()
    delete removed

    leaf.pData = leaf.heap.Top()
    updateWinners(leaf.parent)
    return true
```

root 승자 확인 → 해당 run 힙에서 RemoveTop()

DownHeap으로 재정렬 → leaf.pData 갱신 → updateWinners로 전파

복잡도: heap O(log n) + 전파 O(h)

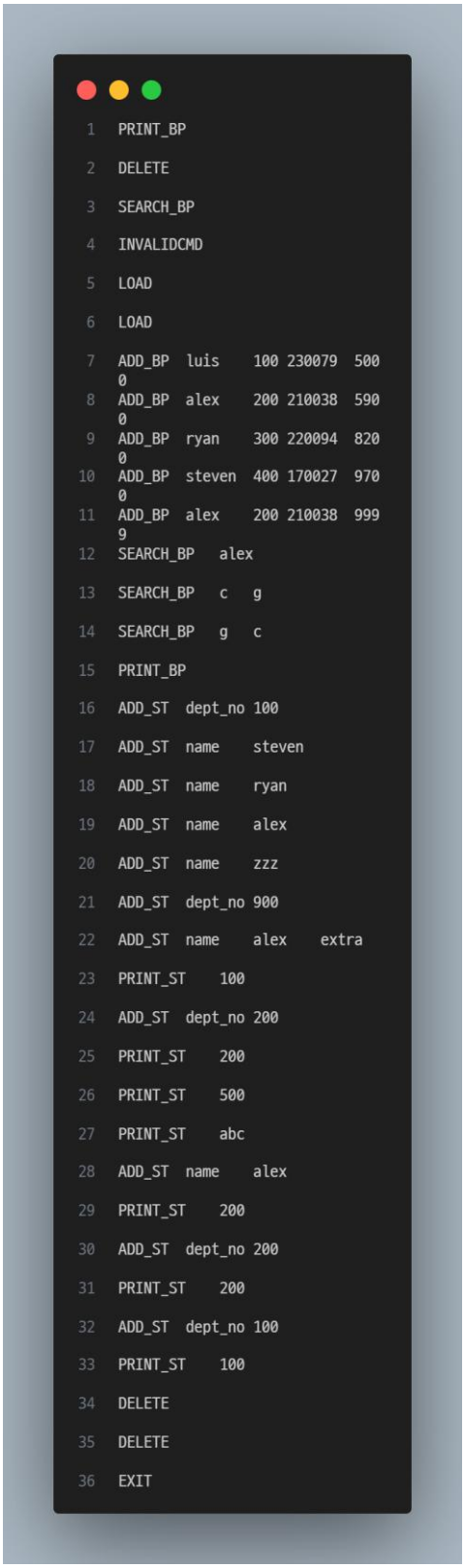
```
function updateWinners(node):
    while node != null:
        L = node.left?.pData
        R = node.right?.pData
        node.pData = chooseWinner(L, R)
        node = node.parent

function chooseWinner(L, R):
    if L == null: return R
    if R == null: return L
    // income desc, name asc, id asc
    if L.income != R.income: return (L.income > R.income) ? L : R
    if L.name   != R.name:   return (L.name   < R.name)   ? L : R
    return (L.id < R.id) ? L : R
```

node가 null이 아닐 때까지 반복: 두 자식 승자 중 더 우수한 EmployeeData 선택해 node.pData에 설정

chooseWinner 비교 규칙은 힙과 동일(연봉 내림차순, 이름 오름차순, ID 오름차순)

복잡도: O(h)

4. Result Screen.



```
PRINT_BP
DELETE
SEARCH_BP
INVALIDCMD
LOAD
LOAD
ADD_BP  luis    100 230079  5000
ADD_BP  alex    200 210038  5900
ADD_BP  ryan    300 220094  8200
ADD_BP  steven  400 170027  9700
ADD_BP  alex    200 210038  9999
SEARCH_BP   alex
SEARCH_BP   c   g
SEARCH_BP   g   c
PRINT_BP
ADD_ST  dept_no 100
ADD_ST  name    steven
ADD_ST  name    ryan
ADD_ST  name    alex
ADD_ST  name    zzz
ADD_ST  dept_no 900
ADD_ST  name    alex    extra
PRINT_ST    100
ADD_ST  dept_no 200
PRINT_ST    200
PRINT_ST    500
PRINT_ST    abc
ADD_ST  name    alex
PRINT_ST    200
ADD_ST  dept_no 200
PRINT_ST    200
ADD_ST  dept_no 100
PRINT_ST    100
DELETE
DELETE
EXIT
```

위에 command를 실행한 결과는 아래와 같다.

```
========ERROR========
400
====================

========ERROR========
700
====================

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

========ERROR========
800
====================

========LOAD========
Success
====================

========ERROR========
100
====================

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

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

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

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

========ADD_BP======
alex/200/210038/9999
====================

======SEARCH_BP=====
alex/200/210038/9999
====================

=====SEARCH_BP======
cristiano/100/220058/990
0
eric/100/250011/4000
florian/200/200719/1200
====================

========ERROR========
300
====================
```

BP구성 전 PRINT_BP에 대한 에러

ST구성 전 DELETE에 대한 에러

BP구성 전 SEARCH_BP에 대한 에러

지정되지 않은 command

LOAD

이미 LOAD된 상태에서 중복 실행에 대한 에러

ADD_BP luis 100 230079 5000

ADD_BP alex 200 210038 5900

ADD_BP ryan 300 220094 8200

ADD_BP steven 400 170027 9700

ADD_BP alex 200 210038 9999

SEARCH_BP alex, ADD_BP에서 넣은 정보와 같음

SEARCH_BP c g, cdefg로 시작하는 이름 cristiano, eric, florian 출력

SEARCH_BP g c, 범위의 에러(g>c)

PRINT_BP, 현재 B+ Tree를 출력

ADD_ST dept_no 100, 부서 번호가 100인 사원들을 selection tree의 100번 run에 추가

ADD_ST, name, steven, 이름이 steven인 사원을 B+ tree에서 검색해서 heap에 삽입

ADD_ST, name, ryan, 이름이 ryan인 사원을 B+ tree에서 검색해서 heap에 삽입

ADD_ST, name, alex, 이름이 alex인 사원을 B+ tree에서 검색해서 heap에 삽입

ADD_ST, name, zzz, 이름이 zzz인 사원은 B+ tree에 없음

ADD_ST, dept_no, 900, 부서 번호가 999인 사원은 B+ tree에 없음

ADD_ST name alex extra, name후 인자가 하나만 있어야 하는데 2개가 들어옴

PRINT_ST 100, 100번 heap의 모든 사원을 이름순으로 정렬해서 출력

ADD_ST dept_no 200, 부서 번호가 200인 사원들을 selection tree의 200번 run에 추가

PRINT_ST 200, 200번 heap의 모든 사원을 이름순으로 정렬해서 출력

PRINT_ST 500, 부서 번호가 500인 사원 없음

PRINT_ST abc, 부서 번호(int)가 아닌 인자

ADD_ST name alex, 이름 alex인 사원을 heap에 추가

200번 heap의 모든 사원을 이름순으로 정렬해서 출력

Alex는 원래 heap에 있었기 때문에 변화 없음

ADD_ST dept_no 200, 부서 번호 200인 사원을 B+ tree에서 검색해 heap에 추가

PRINT_ST 200, 200번 heap의 모든 사원을 이름순으로 정렬해서 출력 (변화 없음)

ADD_ST dept_no 100, 부서 번호 100인 사원을 B+ tree에서 검색해 heap에 추가

PRINT_ST 100, 100번 heap의 모든 사원을 이름 순으로 정렬해서 출력

DELETE

DELETE

EXIT

```
LOAD
ADD_BP  luis    100 230079  5000
ADD_BP  alex    200 210038  5900
ADD_BP  ryan    300 220094  8200
ADD_BP  steven  400 170027  9700
SEARCH_BP   alex
SEARCH_BP   c   g
PRINT_BP
ADD_ST  dept_no 100
ADD_ST  name    steven
ADD_ST  name    ryan
ADD_ST  name    alex
PRINT_ST    100
DELETE
PRINT_ST    100
EXIT
```

기존의 command를 실행하면 아래와 같은 log.txt를 확인할 수 있다.

```
========LOAD========
Success
====================

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

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

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

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

======SEARCH_BP=====
alex/200/210038/5900
====================

=====SEARCH_BP======
cristiano/100/220058/9900
eric/100/250011/4000
florian/200/200719/1200
====================

======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
====================

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

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

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

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

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

=======DELETE=======
Success
====================

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

========EXIT========
Success
====================
```

5. Consideration

과제의 요구사항에서는 ADD_ST는 데이터가 없거나 인자가 잘못된 경우에만 에러코드 500을 출력하도록 되어있습니다. 즉, Selection Tree의 힙(Heap)에 데이터가 **이미 존재할 경우**의 중복 처리 방법이나 별도 에러 코드는 명시되지 않았습니다. 따라서 ADD_ST dept_no 200 (alex 포함)을 실행한 뒤 ADD_ST name alex를 또 실행하면, 힙 내에 동일한 직원이 중복으로 삽입되어 PRINT_ST 실행 시

과 같이 동일한 사원이 여러 번 출력되는 문제가 발생했습니다.

이러한 중복은 DELETE 명령어 수행 시에도 연봉 1위가 중복으로 존재하여 예기치 않은 동작을 유발할 수 있는 잠재적인 오류입니다. 따라서 이 문제를 해결하기 위해, ADD_BP가 중복 시 '업데이트' 처리를 하는 점을 참고하여, EmployeeHeap에 데이터를 삽입할 때 **내부적으로 중복 검사**를 수행하도록 구현했습니다. Heap에 이미 동일한 직원이 존재할 경우, ADD_ST 명령은 성공(Success)으로 처리하되 **실제 힙에 추가 삽입은 하지 않도록 (무시하도록)** 구현하였습니다.