Data Structure Lab. Project #1

제출일자: 2016년 09월 29일 (금)

학 과: 컴퓨터공학과

담당교수: 이기훈 교수님

실습분반: 화요일 3,4

학 번: 2013722004

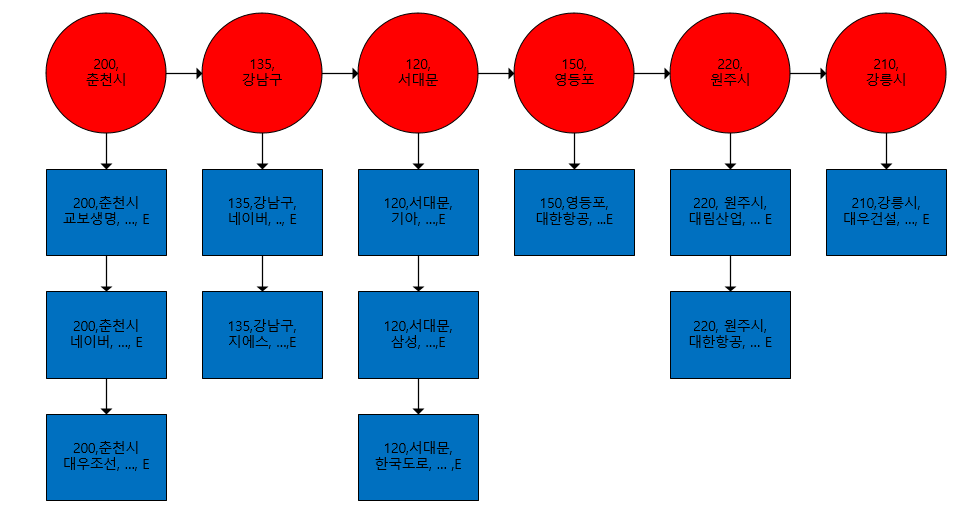
성 명: 최민기

**Introduction**

**지역별 채용정보 관리 프로그램**

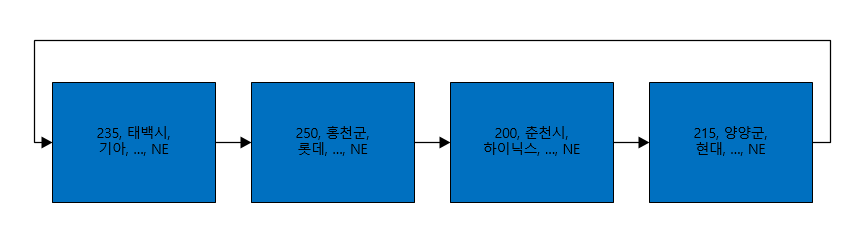
지역별 채용정보 관리 프로그램은 파일로부터 제공되는 지역 번호와 기업 별 TO(빈자리) 유무를 이용하여 지역별 채용정보를 관리하는 프로그램이다. TO유무별로 2D Linked List와 Circular Linked List를 구성한다. TO가 존재하는 채용정보는 2D Linked List로 먼저 구성한 뒤, 지역 코드 별로 BST(Binary Search Tree)를 구성한다. 또, TO가 존재하지 않는 채용정보는 Circular Linked List로 구성한다. 명령어 파일에서 요청하는 명령에 따라 각각의 기능을 수행하고 그 결과를 파일로 저장한다.

1. **2D Linked List**



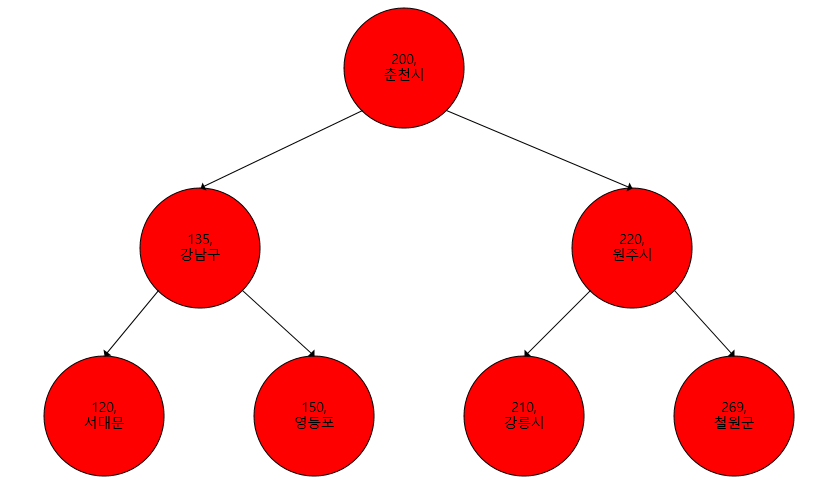
2D Linked List는 IN\_LOAD.txt 파일에서 읽어오는 Data 순서대로 TO(빈자리) 유무가 “E”인 경우 Region Node가 단일 Linked List로 연결되어 있다. 그리고 해당 Region Node 밑에 Region Code와 Region Name이같은 Employment Node가 Company 이름에 따라서 오름차순으로 단일로 연결되어 있다.

1. **Circular Linked List**

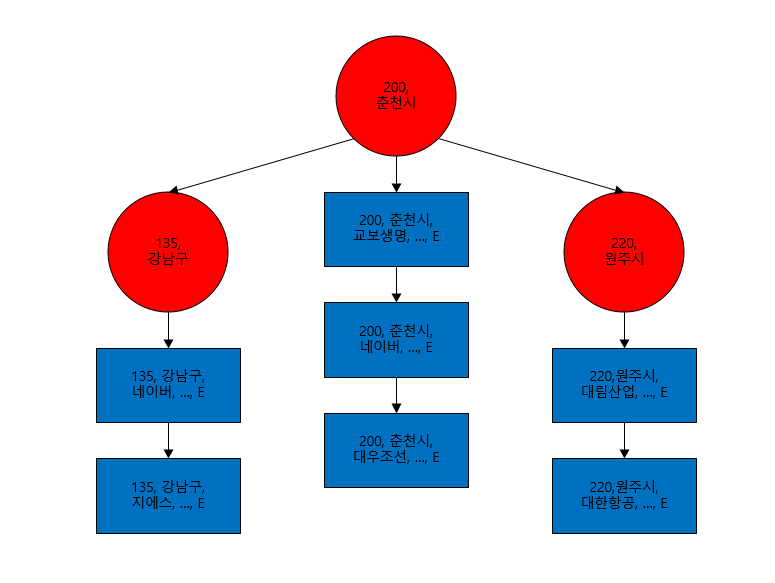


Circular Linked List는 IN\_LOAD.txt 파일에서 읽어오는 Data 순서대로 TO(빈자리) 유무가 “NE”인 경우 Employment Node를 사용해 Circular Linked List로구성되어 있다.

1. **Binary Search Tree**



Binary Search Tree는 위에서 구성했던 2D Linked List의 Region Node들로 구성되어 있다. Binary Search Tree는 Binary Tree의 한 종류로써 Region Code가 중복되는 Node가 존재하지 않고, Node의 왼쪽 Subtree는 해당 Node의 값보다 작은 것들로 구성되어 있다. 또, Node의 오른쪽 Subtree는 해당 Node의 값보다 큰 값들을 가진 Node로 구성되어 있다. 따라서 이 프로그램에서는 Region Code의 값에 따라 BST가 구성된다.



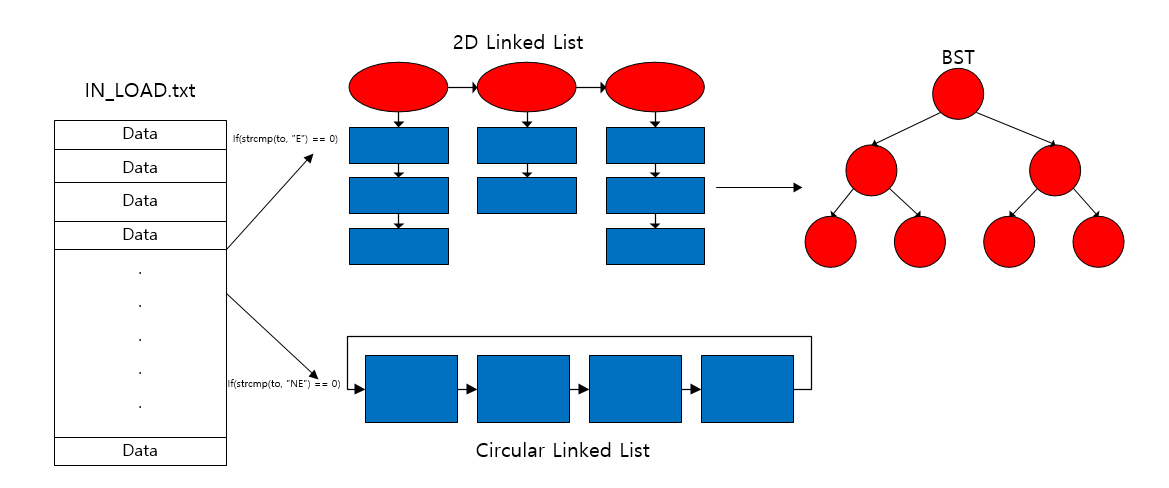
따라서 앞에서 2D Linked List에 Region Node들로 BST를 구성했기 때문에 BST를 구성할 시, 각 Region Node에 같은 Region Code와 Region Name을 갖는 Employment Node가 단일연결로 구성되어 있다.

**Program Command & Function**

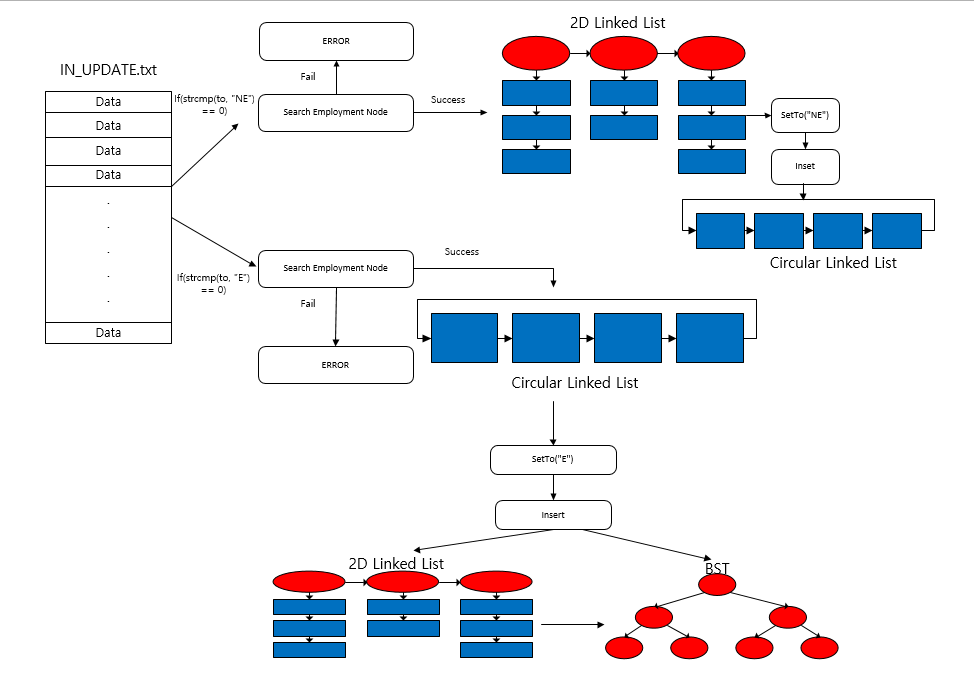
|  |  |
| --- | --- |
| **Command** | **Function** |
| Read Command | IN\_Command.txt 파일을 읽어서 Text내에 존재하는 명령어들을 읽어오는 기능을 한다. |
| LOAD | IN\_LOAD.txt 파일을 읽어서 채용 정보 파일을 불러온다. TO(빈자리) 유무에 따라서 다른 자료구조에 구성된다. 만약 텍스트 파일이 존재하지 않거나 같은 지역에 기업명이 중복된 경우 ERROR를 출력한다. |
| UPDATE | IN\_UPDATE.txt 파일을 읽어서 TO(빈자리) 값이 변경된 파일을 읽어오는 기능을 한다. 데이터를 변경하려면 Data가 있다면 TO(빈자리)에 따라서 다르게 기능 한다. “E”에서 “NE”로 변경되었다면 2D Linked List와 Binary Search Tree에서 삭제되고 Circular Linked List로 이동하고, “NE”에서 “E”로 변경된 경우는 반대이다. 또, 변경하려는 Data가 존재하지 않거나 텍스트 파일이 존재하지 않다면 ERROR를 출력한다. |
| FIND | 입력된 지역코드의 채용 중인 정보를 찾는 기능을 한다. Binary Search Tree에서 Region Code를 Search해서 해당 지역의 모든 채용정보를 출력한다. 또, Search해서 찾은 해당 지역의 모든 채용 정보들은 OUT\_FIND.txt에 저장된다. 만약 찾으려는 Region Code가 존재하지 않는다면 ERROR를 출력한다. |
| DELETE | 입력된 지역코드의 채용 중, 채용 마감 정보를 모두 삭제하는 기능을 한다. 해당하는 지역코드가 존재한다면 모든 자료구조에서 삭제된다. Binary Search Tree에서는 삭제된 Node를 대체하는데 있어서 왼쪽 Subtree에서 가장 큰 값으로 대체된다. 또, 삭제된 채용 정보들은 OUT\_DELETE.txt에 저장된다. 만약 해당하는 지역코드가 존재하지 않는다면 ERROR를 출력한다. |
| PRINT | STL Stack을 사용하여 채용중인 정보를 모두 출력하는 기능을 한다. 이 때, 채용중인 모든 정보는 OUT\_PRINT.txt에 저장된다. 만약 채용 중인 정보가 존재하지 않을 경우 ERROR를 출력한다. |
| EXIT | 프로그램 상의 메모리를 해제하며, 프로그램을 종료하는 기능을 한다. |

**Flow Chart**

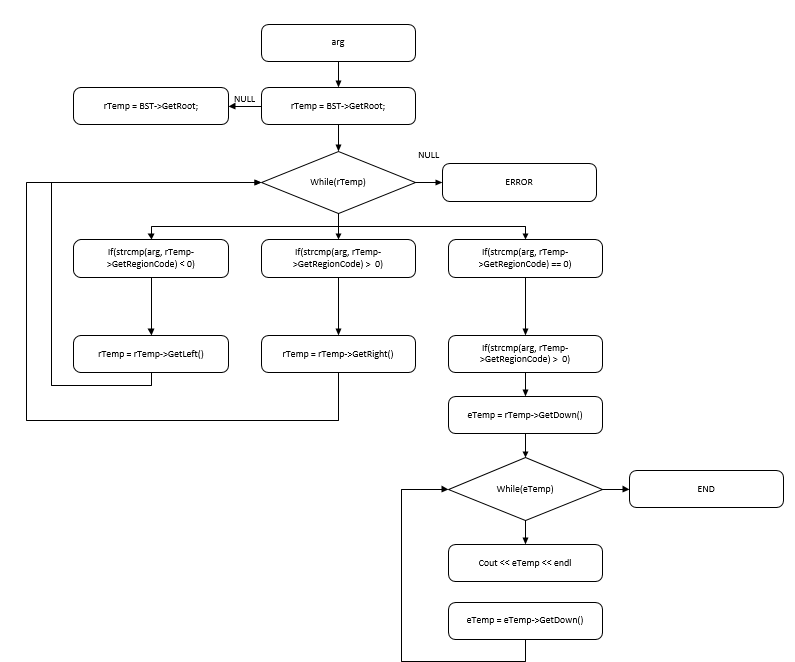
**Load**



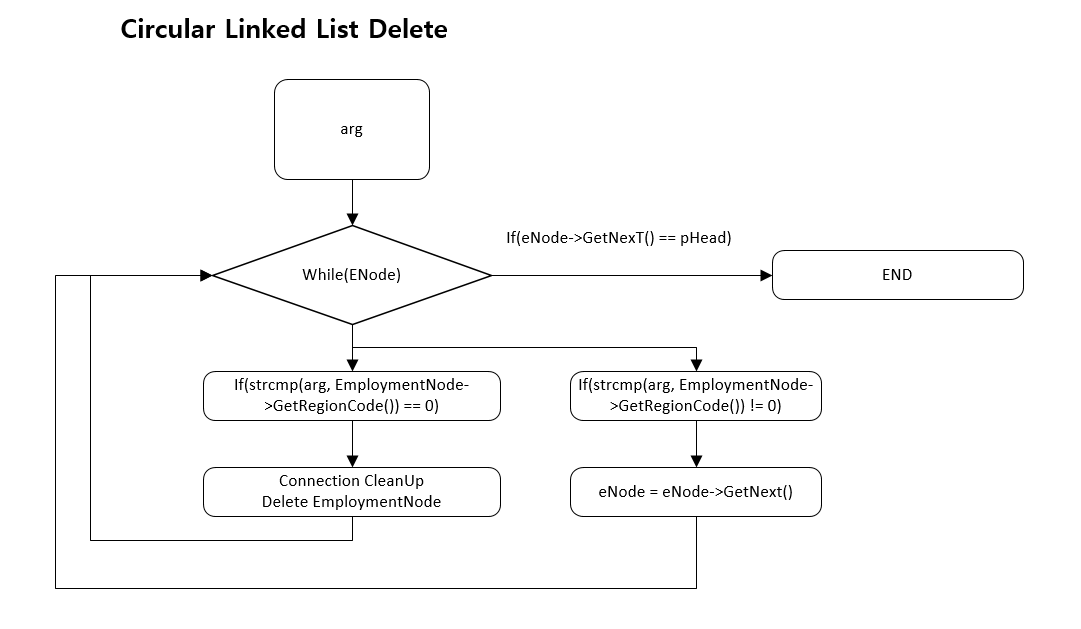
**Update**

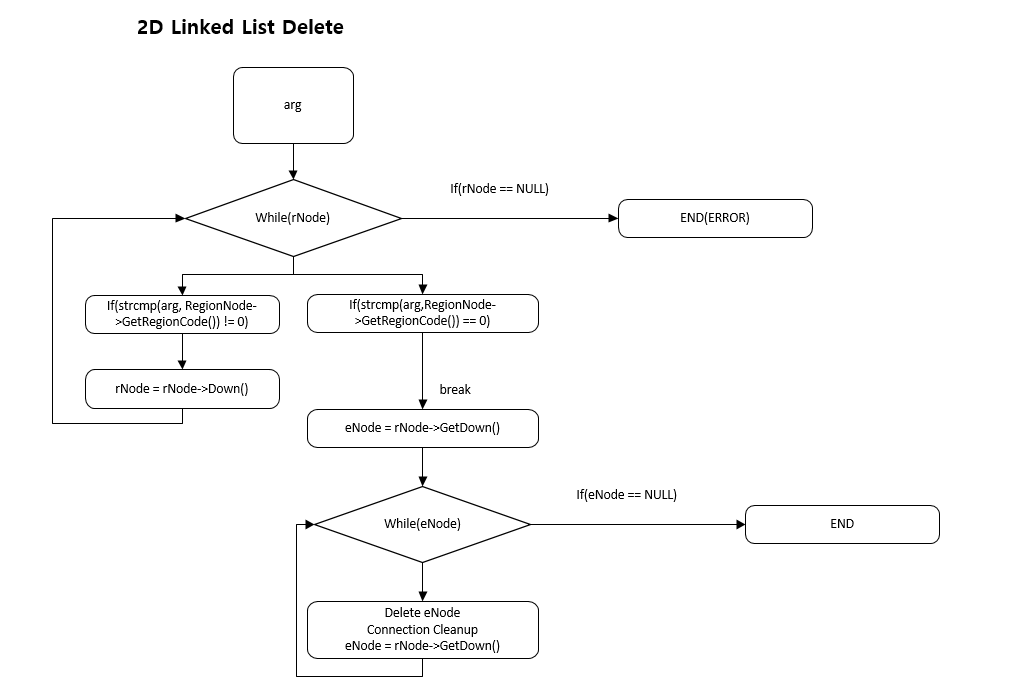


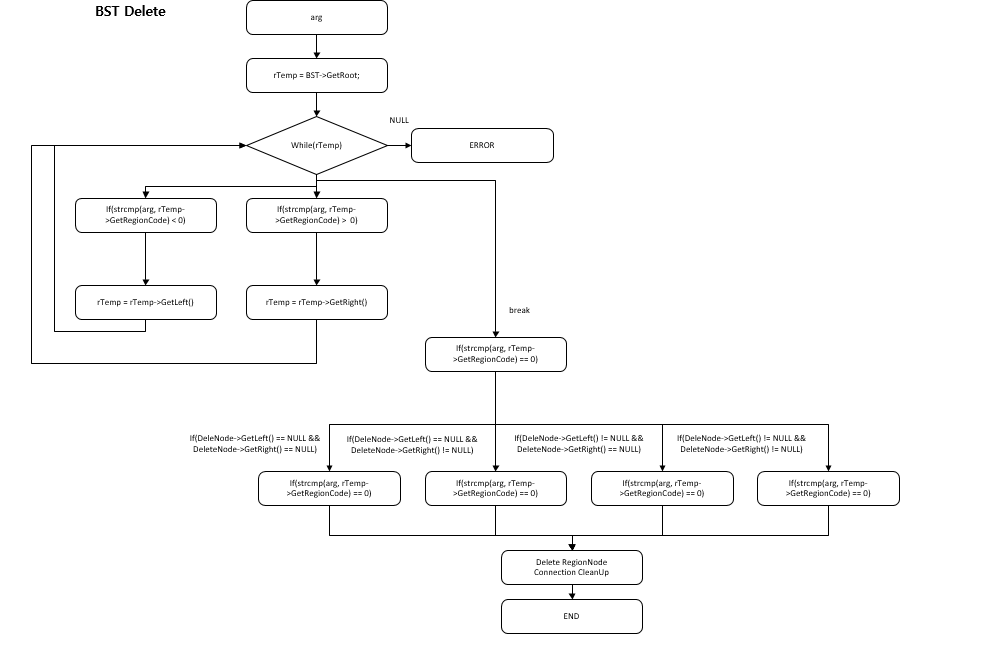
**FIND**



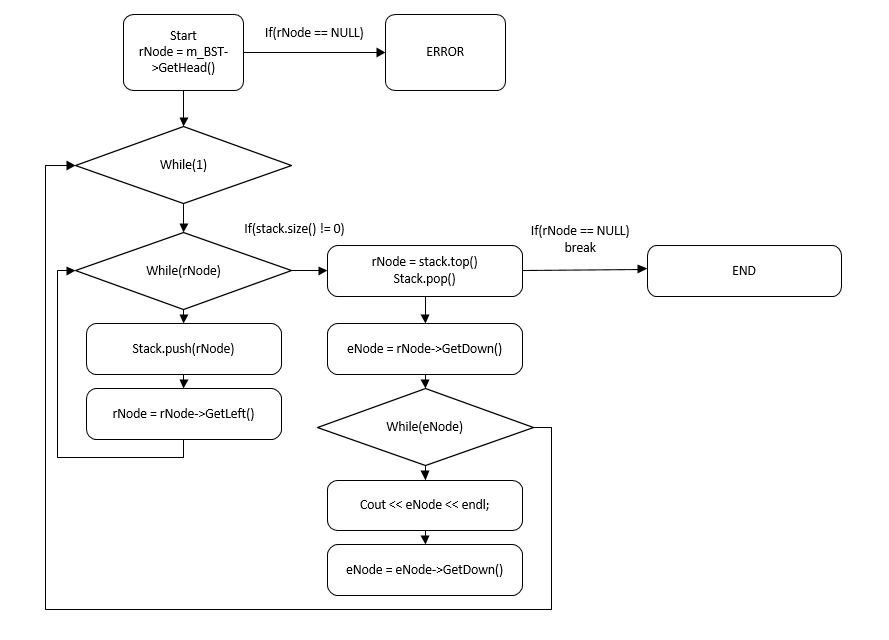
**DELETE**



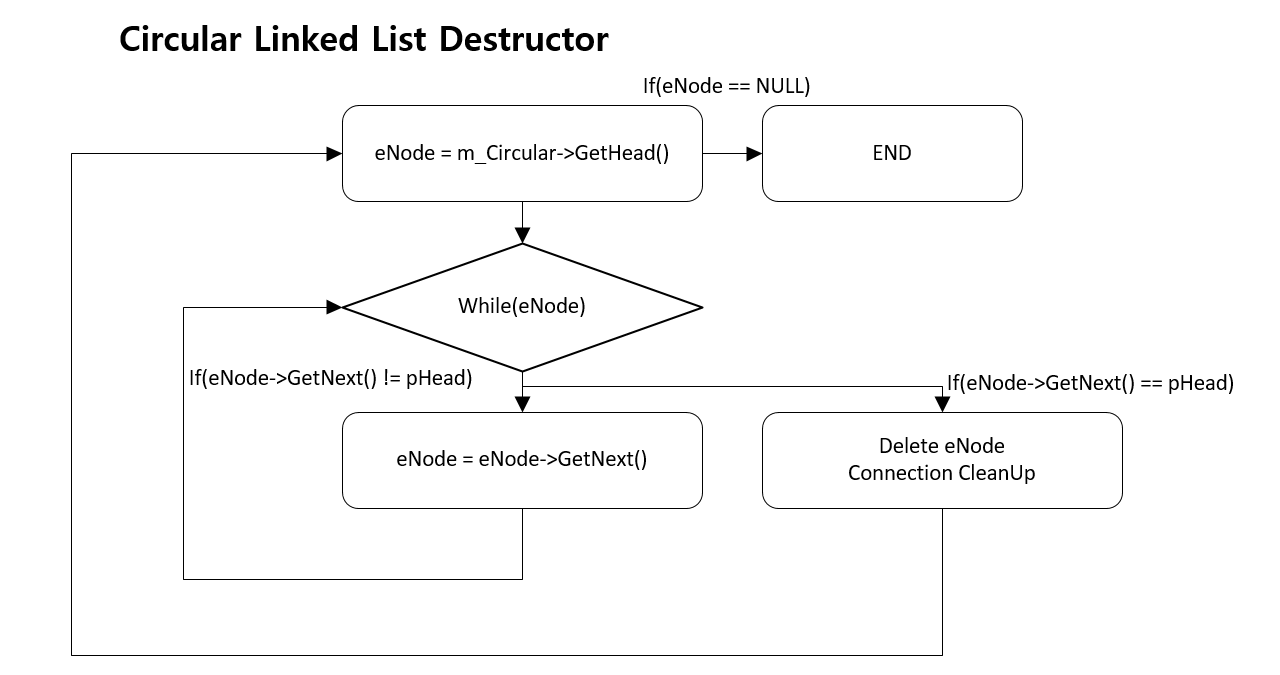


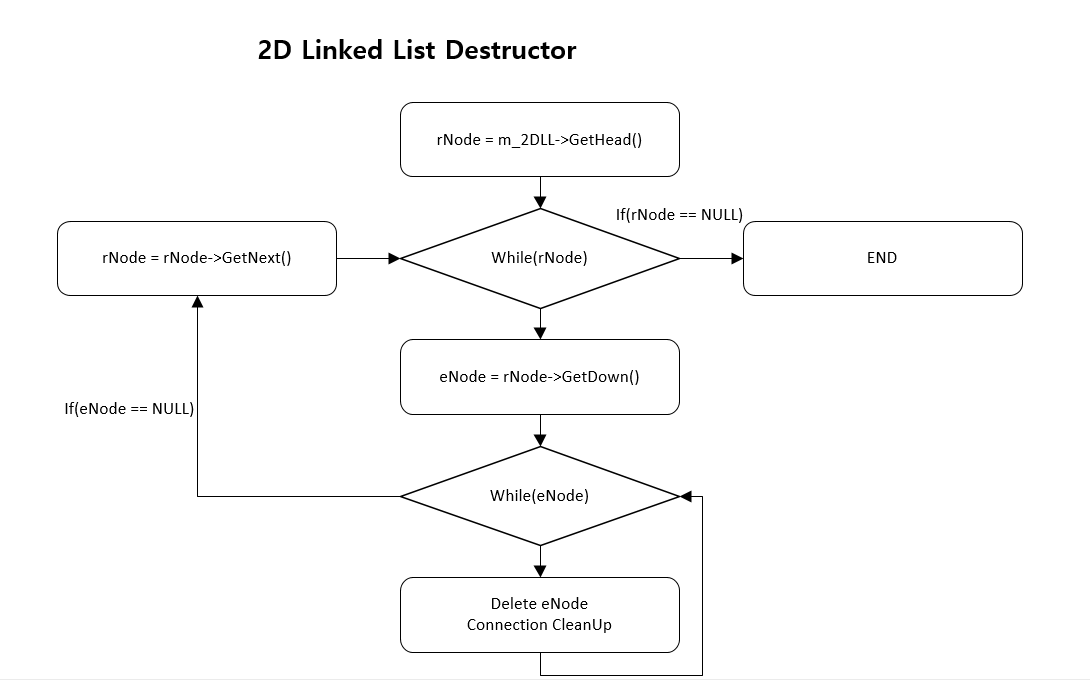


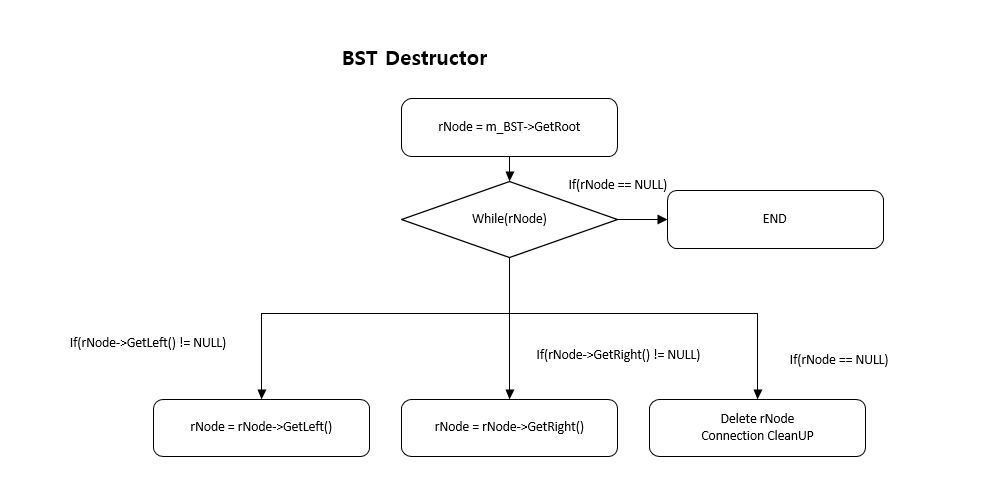
**PRINT**



**EXIT**







**Algorithm**

1. **Read Command**

IN\_Command.txt 파일을 읽어서 저장된 목록을 한 줄 읽어서 Command와 Argument로 분류해서 불러온 Command에 따라서 해당하는 기능을 하는 함수를 호출한다.

1. **LOAD**

먼저 IN\_LOAD.txt파일이 있는지 확인하고 없을 경우 해당 ERROR Code를 출력한다. IN\_LOAD.txt 파일이 존재하는 경우에는 한줄마다 Data를 불러와서 각각 내용에 해당하는 배열에 저장한다. 먼저 TO(빈자리)가 “E”인지, “NE”인지 비교해서 BST, 2D Linked List 와 Circular Linked List를 구성한다.

**“E”인 경우**

2D Linked List Insert 함수와 BST Insert함수를 사용한다. 2D Linked List Region Node Insert함수에서는 Region Node가 존재하지 않을 경우 Region Node를 Head로 Set해주고 Current Region Node를 만들어서 while문을 통해서 Region Node의 Next가 NULL일 때까지 탐색하고 마지막 Node의 다음에 Insert해준다. 만약 탐색도중에 같은 Region Code가 존재한다면 Node탐색을 멈추고 새로 삽입을 하지 않는다. 2D Linked List Employment Node Insert함수에서는 먼저 Region Node의 Head부터 Next하면서 탐색한다. 같은 Region Code를 찾았을 경우 해당 Region Node아래 아무것도 없다면 바로 밑에 Insert해주고 Employment Node가 존재할 경우 Company 이름을 비교해가면서 오름차순이 될 수 있도록 Insert해준다. 만약 Company이름이 같은 것이 있다면 Insert하지 않는다.

**“NE”인 경우**

Circular Linked List에 Insert함수를 사용한다. Circular Linked List Insert함수에서는 Employment Node가 존재하지 않을 때, 삽입되는 Node를 Head로 Set해주고, Node가 존재할 시 Current Employment Node를 생성해 while문을 통해 Next가 Head일 때까지 탐색한다. Next가 Head면 마지막 Node이기 때문에 마지막 위치에 Insert해준다. Circular Linked List에 특성상 마지막 Node와 Head가 연결되어 있기 때문에 삽입된 Node와 Head를 연결해준다.

1. **UPDATE**

먼저 “IN\_UPDATE.txt”파일이 있는지 확인하고 없을 경우 ERROR를 출력한다. 파일이 존재할 경우 ERROR코드를 먼저 출력해야 하기 때문에 IN\_UPDATE.txt파일을 불러와서 TO가 “E”일 경우 Circular Linked List에서 2D Linked List와 BST로 이동하기 때문에 Circular Linked List에 Search 함수를 사용한다. Circular Linked List에 Search 함수는 Employment Current Node를 만들어서 while문을 통해서 Head부터 다음 Node가 Head인 Node까지 탐색을 하는데 Search 하려는 Node를 탐색하지 못한 경우 NULL값을 return 한다. 따라서 return 값이 NULL일 경우 ERROR를 출력한다. TO가 “NE”인 경우 반대이기 때문에 2D Linked List에 Search함수를 사용한다. 2D Linked List에 Search 함수는 먼저 Region Current Node를 만들어서 Search하려는 해당 Node를 Head부터 끝까지 탐색한다. Region Node를 탐색하지 못한 경우 NULL값을 바로 return한다. 만약 Region Node를 찾은 경우 Get Down을 통해서 해당 Region Node밑을 탐색한다. 탐색 실패한 경우 NULL값을 return하고 탐색 성공한 경우 해당 Employment Node를 return한다. 먼저 탐색 실패한 경우에 ERROR를 출력하고 업데이트 할 정보가 1개라도 존재한다면 “IN\_UPDATE.txt”파일을 다시 연다. 다시 TO를 비교한다.

**“NE”인 경우**

2D Linked List의 Search 함수를 사용해서 업데이트하려는 Node를 return 받는다. return받은 Node를 2D Linked List의 Update함수의 인자로 사용한다. 2D Linked List Update 함수는 Search함수와 마찬가지로 Current Node를 생성해서 Next해가면서 Region Node를 찾고 아래를 Get Down을 통해서 탐색하면서 Node를 찾아서 Node의 위치에 따라서 연결선을 정리한다. 만약, Region Node아래 Employment Node가 한 개만 존재한다면 Delete함수를 사용해서 Region Node도 삭제한다. Update하려는 Node값을 가지고 있고, TO를 “NE”로 바꿔주고 Circular Linked List에 Insert 함수를 사용해서 Circular Linked List에 삽입한다.

**“E”인 경우**

먼저 Circular Linked List에서 Search함수를 통해서 해당 Node를 return받는다. Return 받은 Node를 인자로 넣어서 Circular Linked List에서 Search함수처럼 해당 Node를 찾고 해당 위치에 따라서 연결선을 정리해준다. Update하려는 Node를 다시 return받아서 Region Node가 존재하지 않는다면 새로운 Region Node를 동적할당 받고, 2D Linked List와 BST Insert함수를 사용해서 BST와 2D Linked List에 삽입해준다. 만약, Circular Linked List에 Node가 1개 존재하는데 그것을 Update하는 경우 메모리도 해제한다.

1. **FIND**

BST Search함수를 사용한다. BST 자료구조에 특성상 탐색하는 Node가 해당 Node보다 작으면 왼쪽 Subtree에 들어있고, 해당 Node보다 크다면 오른쪽 Subtree에 존재한다. 따라서 Region Current Node를 생성하고 탐색하는 Node와 크기비교를 통해서 탐색한다. 만약 탐색에 실패한 경우 NULL을 return해서 ERROR를 출력하고, Region Node의 탐색에 성공했다면 바로 아래 Employment Node를 return한다. Return받은 Employment Node의 아래가 존재한다면 while문을 통해서 NULL이 될 때까지 탐색해서 출력한다.

1. **DELETE**

먼저 2D Linked List Search함수를 사용해서 Delete하려는 Node가 존재하는지 확인한다. 그리고 Circular Linked List Search함수를 사용해서 Delete하려는 Node가 존재하는지 확인한다. 두개의 자료구조에서 Delete하려는 Node가 모두 존재하지 않을 경우 ERROR를 출력한다.

**2D Linked List & BST Delete**

먼저 Search 함수를 통해서 DELETE하려는 Region Node 바로 밑에 있는 Employment Node를 return 받는다. While문 안에서 2D Linked List Delete함수를 사용한다. 2D Linked List Delete 함수에서는 Region Node밑에 Employment Node를 삭제하고 아래 Employment Node를 연결해주면서 삭제가 진행되었을 시 true를 return한다. 만약 삭제가 모두 진행되어서 Region Node밑에 Employment Node가 존재하지 않는다면 false를 return해서 Region Node 밑에 Employment Node를 모두 삭제한다. 또한 Region Node가 Left, Right, Next로 연결되어 있는데 2D Linked List Del Next 함수를 사용해서 Employment Node를 모두 삭제한 Region Node를 찾아서 단일로 연결되어 있던 선을 정리한다. 그 다음에 BST Delete 함수를 사용해서 Region Node를 삭제한다. 먼저 부모를 가리킬 수 있는 qNode를 생성하고, 삭제하려는 Node를 가리킬 수 있는 pNode를 생성한다. 삭제하려는 Node의 위치를 찾아야 하기 때문에 BST 특성에 따라서 pNode와 qNode의 포인터 위치를 움직인다. Delete하려는 Node의 위치를 찾았으면 각 조건을 비교하면서 Delete를 진행한다.

**먼저 Leaf Node를 지울 시,**

삭제하려는 Node의 자식들이 아예 존재하지 않기 때문에 3가지 조건을 봐주면 된다. 먼저 부모를 가리키는 Node의 위치가 NULL일 경우 Root를 삭제하는 것이기 때문에 root를 지워준다. 또, 부모 Node의 왼쪽 자식을 삭제하는 경우 부모의 왼쪽 자식의 연결을 끊고 해당 Region Node를 삭제해준다. 반대의 경우에는 오른쪽 자식의 연결을 끊고 해당 Region Node를 삭제해준다.

**자식이 1개 있는 Node를 지우는데 오른쪽 자식만 존재할 경우,**

먼저 부모 Node가 NULL일 경우 Root를 삭제하는 것이기 때문에 root의 오른쪽 Node를 root로 Set해주고 root와 오른쪽 자식의 연결을 끊어준다. 그 다음 해당 Node를 Delete한다. 또, 부모 Node의 왼쪽 자식을 삭제하는 경우 삭제하려는 Node의 오른쪽 자식을 부모 Node의 왼쪽 자식으로 연결해주고 해당 Node를 삭제한다. 반대의 경우에는 삭제하려는 Node의 오른쪽 자식으로 삭제하려는 Node의 오른쪽 자식과 연결해주고 해당 Node를 삭제한다.

**자식이 1개 있는 Node를 지우는데 왼쪽 자식만 존재할 경우,**

위의 알고리즘과 비슷하다. Root를 삭제할 경우에는 먼저 root를 삭제하려는 Node의 왼쪽으로 Set해주고 원래 root였던 Node를 삭제한다. 또, 부모 Node의 왼쪽 자식을 삭제할 때, 부모 Node의 왼쪽 자식으로 삭제하려는 Node의 왼쪽을 연결해준다. 그리고 해당 Node를 삭제한다. 반대의 경우에는 부모 Node의 오른쪽으로 삭제하려는 Node의 오른쪽과 연결하고 해당 Node를 삭제한다.

**자식이 2개 있는 Node를 지우는 경우,**

부모와 삭제하려는 Node의 위치를 가리키는 포인터를 두개 그대로 가지고 있고, 자식이 2개있는 경우 삭제하려는 Node의 왼쪽 Subtree중에서 가장 큰 Node를 찾아야 한다. 따라서 이 대체될 Node를 찾기 위해서 Prevprev, Prev, Curr을 생성한다. 처음에 Prevprev는 삭제하려는 Node를 가리키고, Prev는 삭제하려는 Node의 왼쪽 자식을, Curr은 삭제하려는 Node의 왼쪽 자식의 오른쪽 자식을 가리킨다. 왜냐하면 왼쪽 삭제하려는 Node의 왼쪽 Subtree에서 가장 큰 값을 찾기 위한 과정은 삭제하려는 Node의 왼쪽 자식의 오른쪽 자식으로 계속 내려간다면 찾을 수 있기 때문이다. 따라서 Prevprev는 prev의 위치를 쫓아가고, prev는 Curr을 쫓아가고, Curr은 오른쪽으로 계속 내려간다. 먼저 조건을 나누기전에 대체되는 Node인 Prev를 찾고, Prevprev는 Prev의 부모 Node를 가리킨다.

**Root를 지우는 경우,**

먼저 처음에 가리키고 있는 위치에서 Prevprev가 변경되지 않고 삭제하려는 Node의 위치를 가리킨다면 삭제하려는 Node의 왼쪽 자식의 오른쪽 자식이 없는 경우이다. 따라서 Prev는 삭제하려는 Node의 바로 왼쪽 자식이 된다. 따라서 이때는 root의 바로 오른쪽 자식만 Prev의 오른쪽 자식으로 연결해주고 root를 다시 Set해주고 해당 원래 root를 Delete한다. 이 경우가 아닌 경우, Prev의 root의 왼쪽 자식과 오른쪽 자식을 연결해주고, Prev의 왼쪽 자식이 존재한다면 prevprev의 오른쪽 자식으로 연결해주고 삭제하려는 Node를 Delete해주면 된다.

**Root를 삭제하지 않는 경우,**

Root를 삭제하지 않는 경우에는 위의 root를 지우는 경우의 알고리즘과 같다. Head를 Set해주는 과정이 없고, 나머지 과정은 같다.

**Circular Linked List Delete**

해당하는 Region Code에 속하는 모든 Node들을 지워야 하기 때문에 while문을 통해서 Circular Linked List Delete 함수를 계속 호출 한다. Delete하려는 Node를 탐색해서 찾지 못하거나 해당 Node들을 모두 지운 경우 NULL을 return한다. 만약 NULL값이 return되면 Delete를 그만한다. Circular Linked List Delete 함수는 먼저 Head부터 Next를 통해서 Next Node가 Head가 되는 부분까지 삭제될 Node를 탐색한다. 찾은 경우에는 여러가지 조건에 따라서 삭제가 이루어진다. 먼저, Head정보를 삭제하고 Node가 Head Node 한 개만 존재 한다면 모든 연결을 끊어주고 Head를 return해서 삭제한다. 또, Head Node를 삭제하고 Node가 여러 개 존재할 시, Head Node와 다음 Node의 연결을 끊고, 다음 Node를 Head로 Set해준다. 그리고 Node가 더 존재할 경우, 마지막까지 탐색해서 Head와 연결한다. 그리고 삭제하려는 해당 Node를 return해서 삭제한다. 또, Head정보를 수정하지 않고 삭제하려는 Node가 마지막 Node일 경우 이전 Node와 Head를 연결해주고 Delete하려는 해당 Node를 return해서 삭제를 진행한다. 중간의 Node를 삭제하는 경우 이전 Node와 다음 Node를 연결하고 해당 Node를 return해서 삭제한다.

1. **PRINT**

PRINT함수는 먼저 BST에서 getRoot함수를 사용해서 NULL일 경우, 비어 있기 때문에 출력할 정보가 없어 ERROR를 출력한다. NULL이 아닐 경우 Stack을 사용 해서 중위순회 방식을 사용한다. 먼저 rNode를 root를 가리키게 한다. While문 내에서 rNode가 NULL일 때까지 root부터 왼쪽으로 쭉 탐색하면서 Stack에 Push한다. Stack이 비어 있지 않다면 rNode를 쌓인 Stack의 맨 위를 가리키게 하고 Pop해준다. 이 때, rNode가 NULL일 경우 while문을 탈출한다. 이 때는, 모두 탐색이 끝나서 출력이 끝난 경우이다. 그렇지 않을 경우에는 현재 rNode가 가리키고 있는 Region Node 밑에 있는 Employment Node를 Down하면서 탐색하며 모두 출력한다. 그리고 rNode를 오른쪽 자식을 가리키게 해준다. 이 방식을 통해서 계속 진행하면 Stack을 사용해서 모든 채용정보를 출력할 수 있다.

1. **EXIT**

EXIT함수는 모든 자료구조에 소멸자를 호출한다. 모든 자료구조에 소멸자 호출을 통해서 구성되어 있는 모든 자료구조들을 삭제한다.

**Circular Linked List 소멸자**

먼저 Head가 NULL이 아닌 경우 진행한다. Head가 NULL인 경우, 모든 Node가 존재하지 않기 때문에 삭제할 필요가 없기 때문이다. 먼저 Node가 1개만 존재한다면 Head를 삭제하고 끝낸다. 하지만, Node가 여러 개 존재할 경우 while문을 통해서 한 개씩 모두 삭제한다. Prevprev를 Head로 설정하고 Prev를 Head의 Next로 처음 설정을 한다. 먼저 마지막 Node를 탐색하면서 마지막 Node를 찾아서 삭제하고 이전 Node와 Head를 연결해준다. 맨 뒤를 탐색해서 연결선을 정리하면서 모든 Node들을 삭제한다.

**2D Linked List 소멸자**

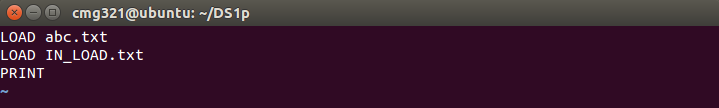
이 때도 Head가 NULL이 아닌 경우 진행한다. NULL인 경우 Node가 존재하지 않기 때문이다. While문 2개를 사용해서 Region Node는 Next로 가면서 탐색하고, 각 Region Node들 밑의 Employment Node들을 모두 삭제한다.

**BST 소멸자**

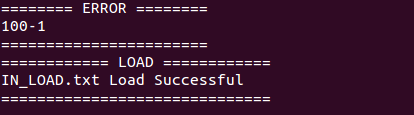
BST 소멸자도 마찬가지로 Head가 NULL이 아닌 경우 진행한다. NULL인 경우 Node가 존재하지 않기 때문이다. Head가 NULL이 아닌 경우 처음에 인자로 root를 넣어줘서 Remove함수를 통해 진행한다. 재귀함수 호출을 통해서 제일 밑에 자식부터 Delete해주면서 모든 Node들을 Delete해준다.

**Result Screen**

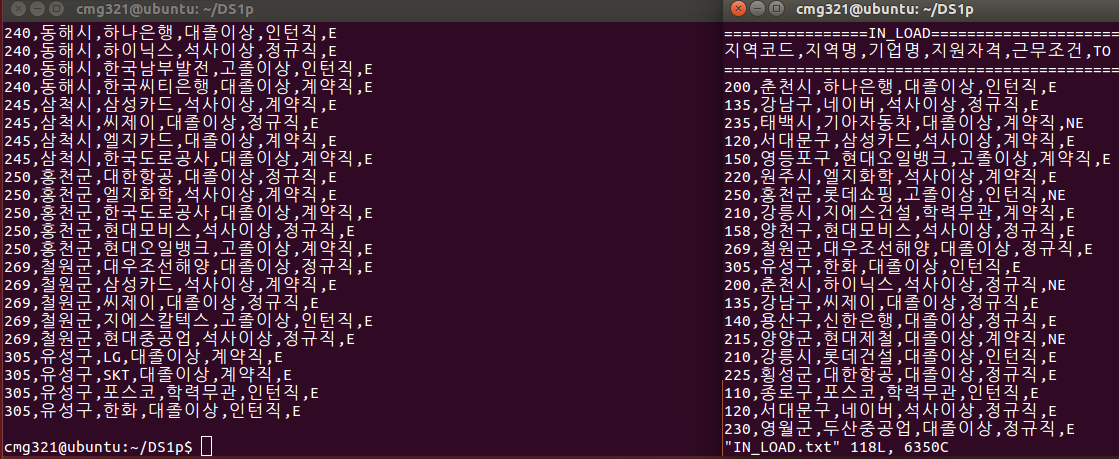
**LOAD, PRINT**



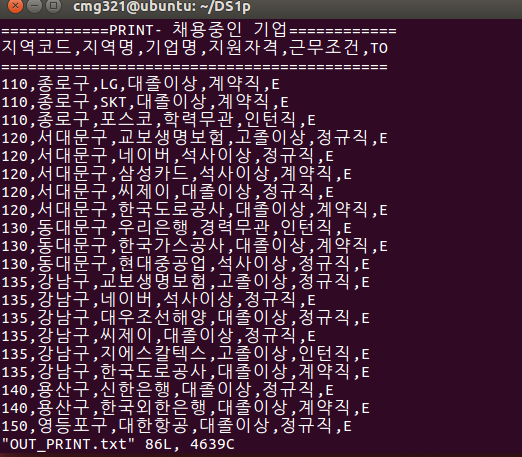
“IN\_Command.txt”파일에 저장되어 있는 Command 명령어이다.



먼저 처음 명령어에서 “IN\_LOAD.txt”파일을 읽지 않았기 때문에 ERROR 코드 100-1을 출력한 것을 확인할 수 있다. 그리고 다음 명령어에서 “IN\_LOAD.txt”파일을 읽어서 성공 메시지를 출력한 것을 확인할 수 있다.

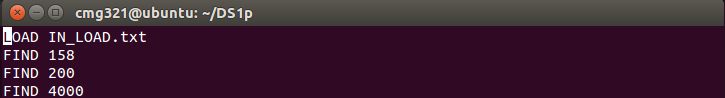


그리고 다음 명령어 “PRINT”를 통해서 채용중인 기업들을 Region Code 순으로 해서, 오름차순으로 정렬해 모두 출력한 것을 확인할 수 있다.

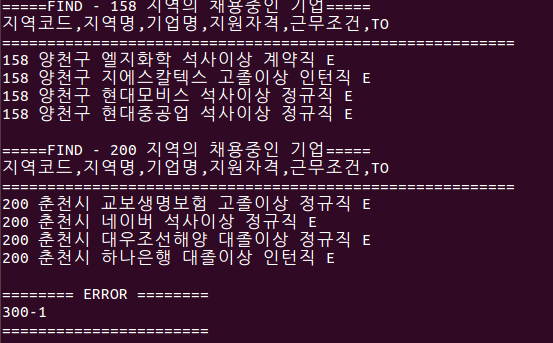


또한 모든 정보들이 “OUT\_PRINT.txt”에 저장된 것을 확인할 수 있다.

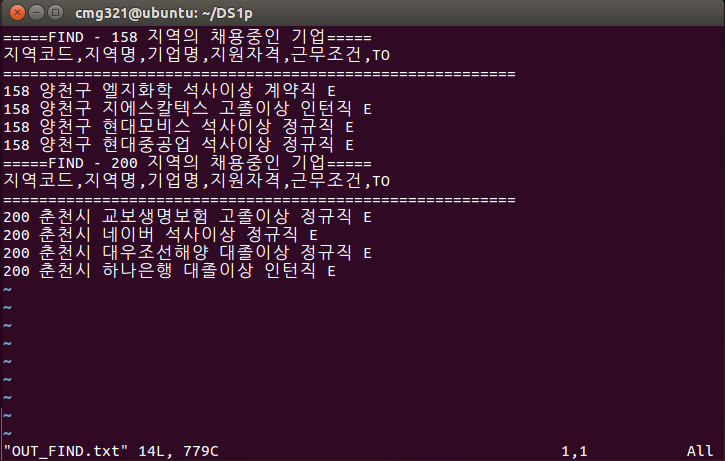
**FIND**



“IN\_Command.txt”파일에 저장되어 있는 Command 명령어이다.

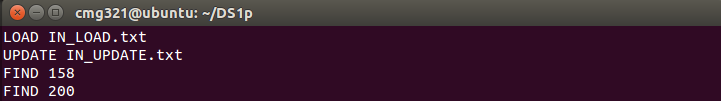


Region Code가 158인 지역과, 200인 지역의 모든 채용정보를 출력한 것을 확인할 수 있다. 또한 해당하는 Region Code를 가지고 있는 지역이 없기 때문에 ERROR를 출력한 것을 확인할 수 있다.

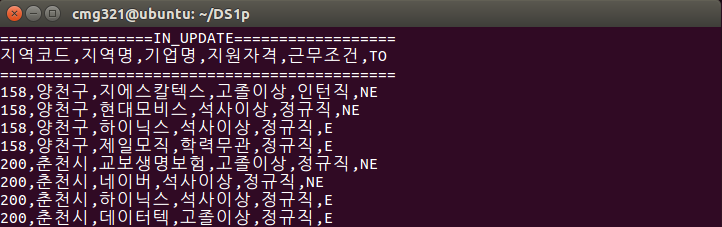


찾은 모든 정보들이 “OUT\_FIND.txt”에 저장된 것을 확인할 수 있다.

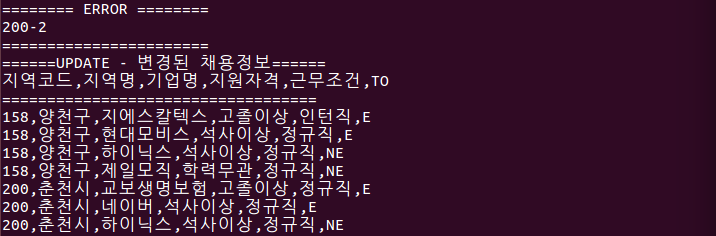
**UPDATE**



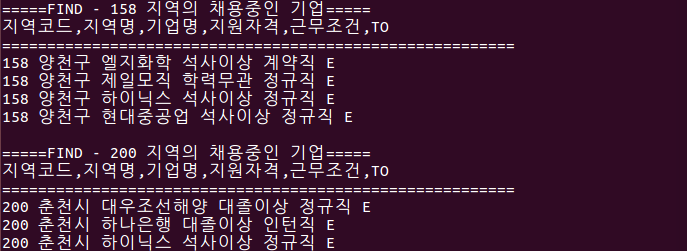
“IN\_Command.txt”파일에 저장되어 있는 Command 명령어이다.



“IN\_UPDATE.txt”파일에 저장되어 있는 Update하려는 목록이다.

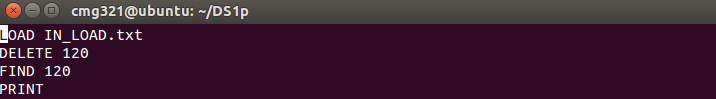


이전에 FIND에서 확인했을 때, “200, 춘천시, 데이터텍”이라는 채용중인 기업에 존재하지 않기 때문에 ERROR를 출력한 것을 확인할 수 있다. 나머지는 존재하는 Data이기 때문에 모두 출력한 것을 확인할 수 있다.

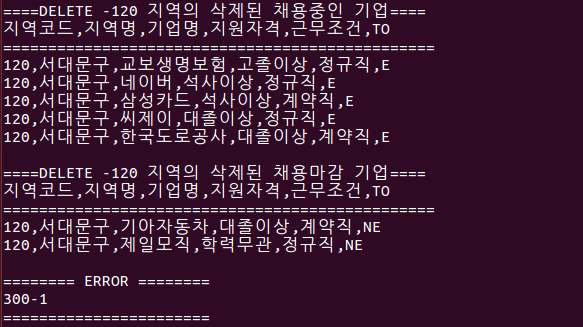


FIND 결과 “158, 지에스” , “158, 현대모비스”가 NE로 수정되었고, “158, 제일모직”, “158, 하이닉스”가 E로 수정되었고 “200, 교보”, “200, 네이버”가 NE로 수정되었고, “200, 하이닉스”가 E로 수정되었기 때문에 다음과 같은 결과를 얻을 수 있다.

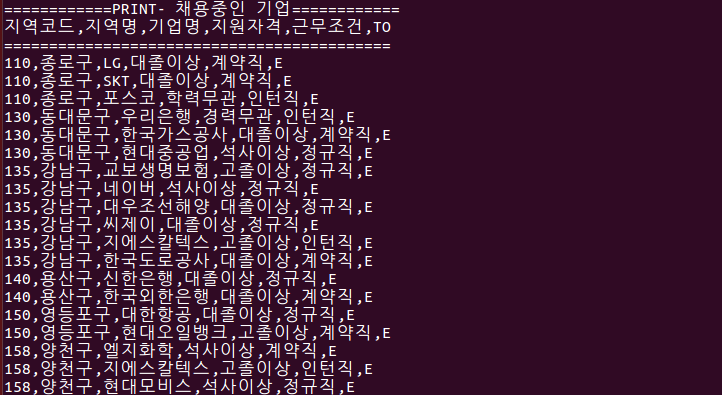
**DELETE**



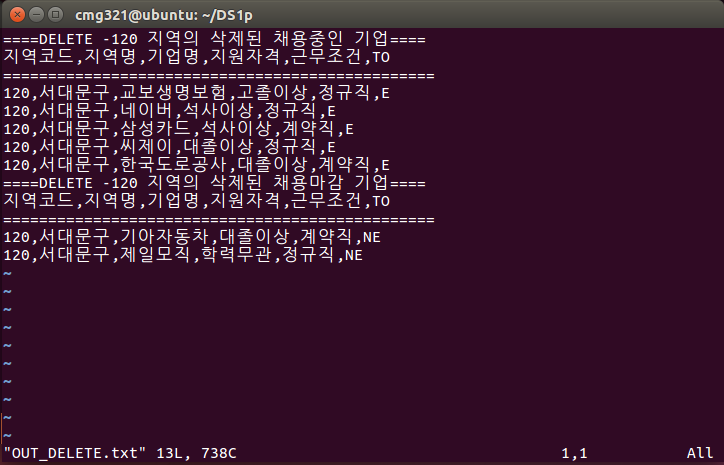
“IN\_Command.txt”파일에 저장되어 있는 Command 명령어이다.



DELETE – 120을 통해서 채용중인 기업과 채용마감 기업에서 삭제된 목록들을 출력한 것을 확인할 수 있다. 또, FIND – 120을 했을 경우 120지역의 모든 Node가 존재하지 않으므로 ERROR를 출력한 것을 확인할 수 있다.



PRINT 명령어 결과 120 Region Code를 가지고 있는 Node가 없는 것을 확인할 수 있다.



삭제된 목록이 OUT\_DELETE.txt 에 저장된 것을 확인할 수 있다.

**Consideration & Conclusion**

먼저 지난학기 객체지향프로그래밍 설계 Project를 진행하는데 있어서 내용이 비슷해서 틀을 잡기 쉬웠다. 먼저 Circular Linked List와 Singly Linked List로 구성해서 2D Linked List로 구성하는 자료구조는 이미 했던 것이기 때문에 크게 어렵지 않았다. BST는 이번 데이터구조 강의와 설계 과제에서 진행했던 부분이 큰 도움이 되었다. BST 자료구조를 구성하는데 필요한 개념과 실제 코드로 작성해보았기 때문에 3개의 자료구조를 다루는 것은 크게 어렵지 않았다. 하지만 설계과제에서 진행했던 DELETE와 같이 기능을 구현하고 검증해봤는데 잘 되지 않았다. 왜냐하면 설계 과제에서는 Key값을 그냥 대체해 버리지만 대체되는 Node의 딸린 자식들을 처리하지 못하는 부분에서 다르다는 것을 알고 수정할 수 있었다. 그리고 UPDATE 기능을 구현하는데 있어서는 예외 조건이 많아서 좀 까다로웠다. 그리고 리눅스에서 Compile하는데 있어서 어려움이 많았다. 먼저 Visual Studio에서 작성해서 리눅스로 옮겨서 make했는데 Segmentation fault 오류로 정상적으로 작동되지 않았다. 그래서 eclipse에서 디버그를 해보려고 실습 강의자료 그대로 실행법을 따라서 진행했는데 파일 위치를 찾을 수 없다는 오류 때문에 디버그도 쉽게 되지 않았다. 따라서 Command를 읽는 부분에서 수정을 좀 해서 다시 디버그 했더니 성공할 수 있었다. 그런데 LOAD에서 다시 오류가 나는 것을 확인할 수 있었다. 디버그 결과 File의 끝에 도달했는데도 탈출을 못하는 것이 문제였다. 따라서 LOAD data를 불러오는 배열을 while문 끝에 초기화해주고 시작지점에서 strlen함수를 사용해서 길이가 0일 경우 탈출할 수 있게 하니까 성공했다. Visual에서는 while(fin.eof)로 진행하면 파일 끝에서 잘 멈추는데 리눅스에서 Data를 많이 불러 올 경우 파일의 끝을 왜 못 찾는지는 아직도 잘 모르겠다. 이 점에 대해서는 공부가 더 필요할 것 같다. 이전 학기에 진행했던 Project와 데이터구조 설계 및 실습 수업이 도움이 많이 되어서 쉽게 진행할 수 있었습니다. 감사합니다.