



데이터 구조 설계 및 실습 프로젝트 보고서

2015722052 윤좌홍

2015722049 이승호

2015722094 홍종현

introduction

이 프로그램은 세 개의 자료구조를 활용하여 단어들과 그 뜻을 저장하는 프로그램입니다. 이때 프로그램의 기능으로 Queue 자료구조에서 빼낸 데이터를 Binary Search Tree에 insert하는 것, Binary Search Tree에서 빼낸 데이터를 Circular LinkedList에 insert하는 것, 각각의 자료구조에 들어있는 데이터를 수정하거나 저장하는 것, 저장된 것을 다시 불러오는 것 등이 있습니다.

프로그램의 동작 명령어로는 다음과 같습니다.

LOAD

기존의 단어장 정보를 불러오는 명령어로, 텍스트 파일에 단어장 정보가 존재할 경우 텍스트 파일을 읽어 (to_memorize_word.txt, memorizing.txt, memorized.txt) 이전과 동일한 연결 순서대로 다시 저장합니다. 만약 3개의 텍스트파일이 모두 존재하지 않거나 자료구조에 이미 데이터가 들어가 있는 경우에 알맞은 에러 코드를 출력합니다.

ADD

단어 텍스트 파일에 있는 단어 정보를 읽어오는 명령어로, 단어 텍스트 파일에 존재하는 단어들을 Queue(to_memorize)에 모두 저장합니다. 단어 텍스트파일이 존재하지 않거나 파일에 단어가 존재하지 않을 경우 알맞은 에러코드를 출력합니다.

MOVE

사용자가 입력한 수 만큼 TO_MEORIZE의 단어들을 MEMORIZING으로 옮기는 명령어로, 1~100 사이의 정수를 입력받고 입력받은 수와 BST(MEMORIZING)의 단어 수의 합이 100을 넘어가지 않도록 합니다. Queue(TO_MEMORIZE)에 단어가 없는 경우, 입력한 수 만큼 단어가 존재하지 않을 경우 등이 발생하면 에러 코드를 출력합니다.

SAVE

현재 단어장 정보를 저장하는 명령어로 각각 Queue, BST, Circular LinkedList에 저장된 단어들을 to_memorize, memorizing, memorized 텍스트 파일에 저장합니다. memorizing 텍스트파일에 단어를 알파벳 순으로 저장하면 LOAD할 때, BST의 child node가 한쪽으로 쏠려 skewed binary tree가 되기 때문에 pre - ordering으로 저장하도록 합니다. 이때, 자료구조에 저장된 데이터가 존재하지 않을 경우 에러코드를 출력합니다.

TEST

단어를 외웠는지 테스트하는 명령어로, memorizing에 입력한 단어가 있는지 찾아 뜻이 맞는지 확인하고, 맞을 경우 해당 단어를 memorizing에서 memorized로 이동시킵니다. 입력한 단어가 memorizing에 존재하지 않거나 단어의 뜻이 틀릴 경우 에러코드를 출력합니다.

SEARCH

단어의 뜻을 찾아 출력하는 명령어로, to-memorize, memorizing, memorized에 입력한 단어가 존재하는 경우, 영어 단어와 한글 뜻을 출력해줍니다. 만약 입력한 단어가 3개의 자료구조에 모두 없을 경우, 에러코드를 출력합니다.

PRINT

입력한 단어장에 있는 단어들을 출력하는 명령어로, to-memorize를 입력할 경우, Queue에 있는 단어들을 순차적으로 전부 출력합니다. memorizing의 경우 추가로 R-PRE, I-PRE, R-IN, I-IN, R-POST, I-POST, I-LEVEL을 입력받아 각각의 순서대로 단어들을 전부 출력합니다. MEMORIZED를 입력할 경우 circularLinkedList에 저장된 단어들을 전부 출력합니다. 이때, 입력한 단어장 정보가 존재하지 않을 경우 에러코드를 출력합니다.

UPDATE

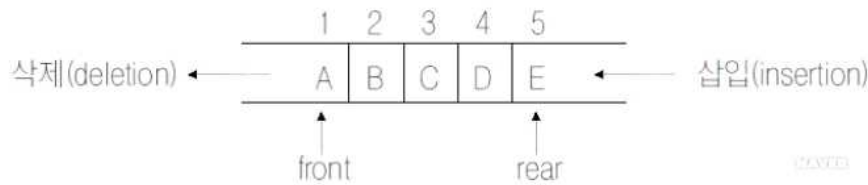
단어의 뜻을 변경하는 명령어로 to-memorize, memorizing, memorized에 입력한 단어가 존재할 경우 단어의 한글 뜻을 입력한 뜻으로 변경한 뒤 결과를 출력해줍니다. 만약 단어가 존재하지 않거나 단어장 정보가 존재하지 않을 경우 알맞은 에러코드를 출력합니다.

EXIT

프로그램 상의 메모리를 모두 해제하며 프로그램을 종료합니다.

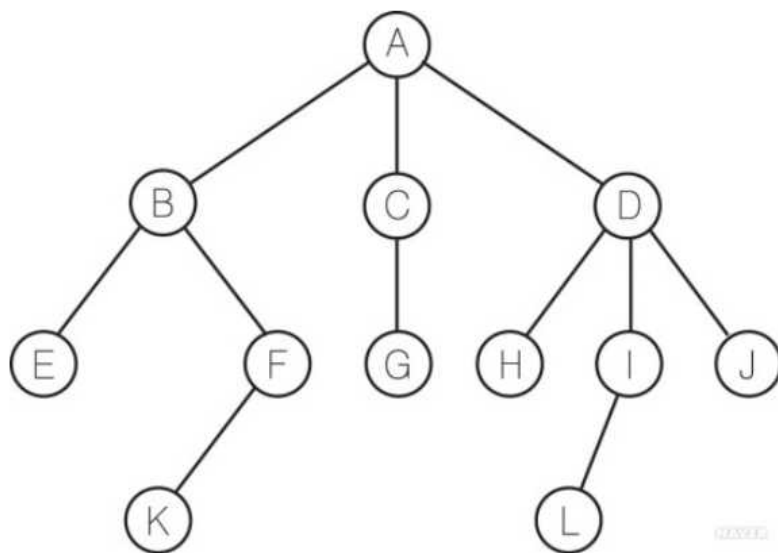
단어와 의미를 저장할 각각의 자료구조의 특성은 다음과 같습니다.

Queue



Queue는 데이터를 맨 뒤에서부터 집어넣고, 넣은 순서부터 빠져나오는 구조를 가지고 있습니다.(first in first out) 위의 사진에서는 큐의 앞이 front이고 맨 뒤가 rear라는 포인터가 가리키고 있습니다. 각각의 역할은 front의 경우 큐의 먼저 들어온 순서대로 데이터를 반환하기 위해 항상 맨 앞을 가리키고 있습니다. 따라서 데이터를 반환할 때, return front로 쉽게 포인터나 데이터를 반환할 수 있도록 도와줍니다. rear의 경우 삽입을 좀 더 편하게 해주는 포인터로써, 큐의 하위 계층적 구조인 연결리스트(Linked List)의 삽입과 비슷합니다. 둘 다 삽입연산의 경우엔 리스트의 맨 뒤까지 가서 삽입을 해야 하므로 시간이 리스트의 길이 n 만큼 걸리고 $O(n)$ 만큼의 시간을 필요로 하게 됩니다. 따라서 이것을 개선하기 위해 만들어진 것이 리스트의 맨 뒤를 가리키는 rear(tail)포인터입니다. rear포인터의 다음에 새로 들어갈 데이터를 삽입하고 rear를 다시 맨 뒤를 가리키도록 초기화 해줌으로써 동작합니다.

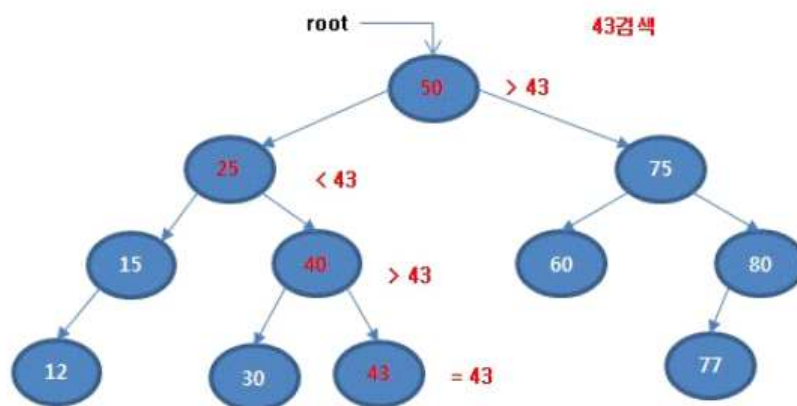
Tree



이진 탐색 트리의 경우 위의 트리에서 파생된 개념입니다. 설명에 들어가기에 앞서 트리의 기

본 구조부터 살펴보면, 트리는 가장 위의 하나가 root노드로, 뿌리에서부터 시작하는 나무와 비슷하여 tree라고 이름 붙여졌습니다. root에서 가지처럼 뻗어나온 것이 branch, 각각의 노드들을 element라고 하며, root로부터 가장 멀리있는 자식노드가 하나도 없는 노드들을 leaf 노드라고 합니다. 한 노드마다 자식을 가질 수 있는 개수는 정해져 있지 않으며 트리를 탐색 (Traversal)하기 위해선 배열을 이용하여 트리의 모든 노드들을 배열에 저장하여 접근하는 방법과 root노드부터 탐색하여 branch를 따라 모든 leaf까지 순차적으로 내려가는 트리 순회방법(Tree Traversal)이 있습니다. 배열을 이용한 방법의 경우 트리의 자식노드들을 Level로 구간화하여 배열에 저장하기 때문에 한 Level에 빈 노드들이 많을 경우 메모리 낭비가 심하다는 단점이 있습니다. 따라서 root노드를 기준으로 Tree Traversal하는 방법을 사용하여 프로젝트를 진행할 것입니다.

-Binary Search Tree

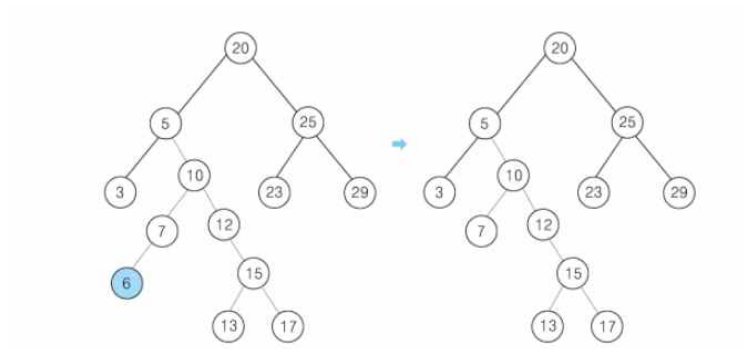


Tree의 종류 중에서 Binary Search Tree라는 것이 존재합니다. Binary Search Tree란 Tree에서 키 값을 기준으로 부모노드보다 자식노드의 값이 작으면 왼쪽에 위치하고, 크다면 오른쪽에 위치하게 되는 Tree구조입니다. 이를 응용하면 그냥 Tree에서의 삽입보다 간편해질 뿐더러 탐색 또한 시간이 훨씬 더 단축됩니다. 예를들어, 위의 사진과 같은 경우에서 43을 검색한다고 했을 때, 다음과 같은 과정을 거칩니다. root노드의 키 값 50과 비교하여 작으면 왼쪽 노드로 들어간다 -> 대상 노드의 키 값 25와 비교하여 크면 오른쪽으로 들어간다 -> 대상 노드의 키 값 40과 비교하여 크면 오른쪽으로 들어간다, 이때 오른쪽은 비어있는 곳이므로 43이 위치할 곳은 40의 오른쪽 자식노드가 되는 것입니다.

탐색의 경우에도 삽입과 비슷한 과정을 거칩니다. 가장 먼저 root노드를 기준으로 한 키 값이 root보다 작으면 왼쪽, 크다면 오른쪽으로 들어간다 -> 이 과정을 찾고자 하는 키 값이 나올 때까지 or 더 이상 찾을 노드가 없을때까지 재귀적으로 반복하게 됩니다.

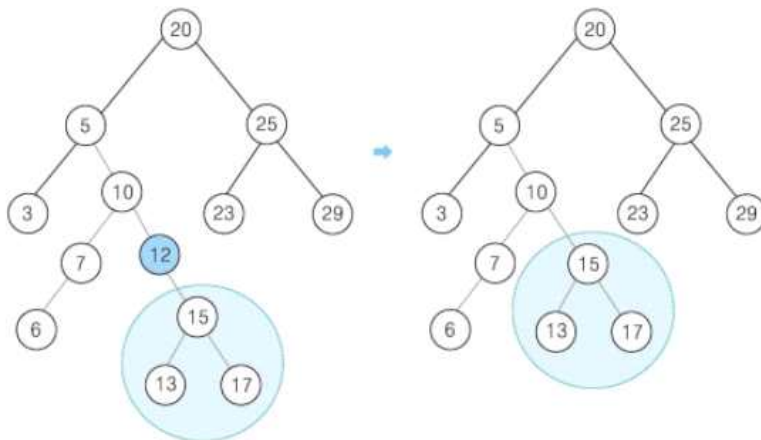
삭제 연산의 경우엔 조금 복잡합니다. 삭제의 경우 case가 3가지로 나뉘게 됩니다.

1. 자식노드가 0개인 경우



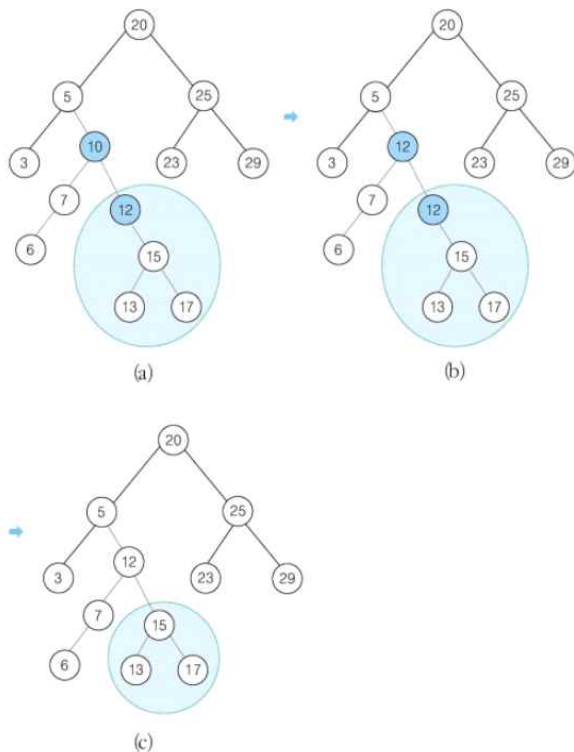
위의 사진과 같은 트리에서 6을 제거하고자 할 경우 7의 왼쪽 자식과의 연결을 끊으면 (NULL) 자식노드가 0개인 경우의 삭제가 완료됩니다.

2. 자식 노드가 1개인 경우



위의 사진의 경우 자식노드가 1개인 경우의 삭제를 보여주는 것으로 부모노드의 연결을 삭제노드의 자식노드에게로 연결시켜 연결관계를 끊는 것을 확인할 수 있습니다.

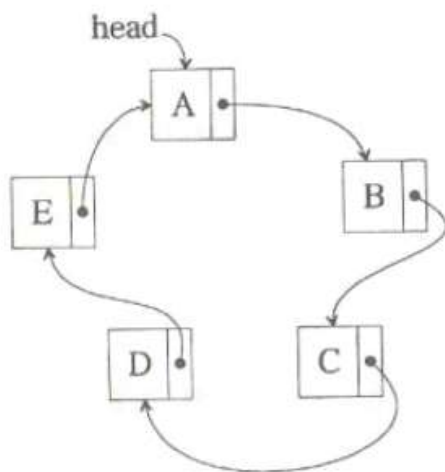
3. 자식 노드가 2개인 경우



자식노드가 2개인 경우의 삭제는 조금 특이합니다. 먼저 삭제하고자 하는 노드의 자식노드를 삭제노드의 자리에 위치시킵니다. 그런 다음 부모 노드의 포인터로 그 자리를 가리키도록 합니다. 이렇게 한 이유는 Binary Search의 특성이 무너지지 않도록 해야하기 때문입니다. 그냥 바로 삭제만 할 경우 값의 대소관계가 맞지않아 사용하는 의미를 저해하게 되므로 노드를 삭제할때는 반드시 위의 case 3개를 맞춰주어야 합니다.

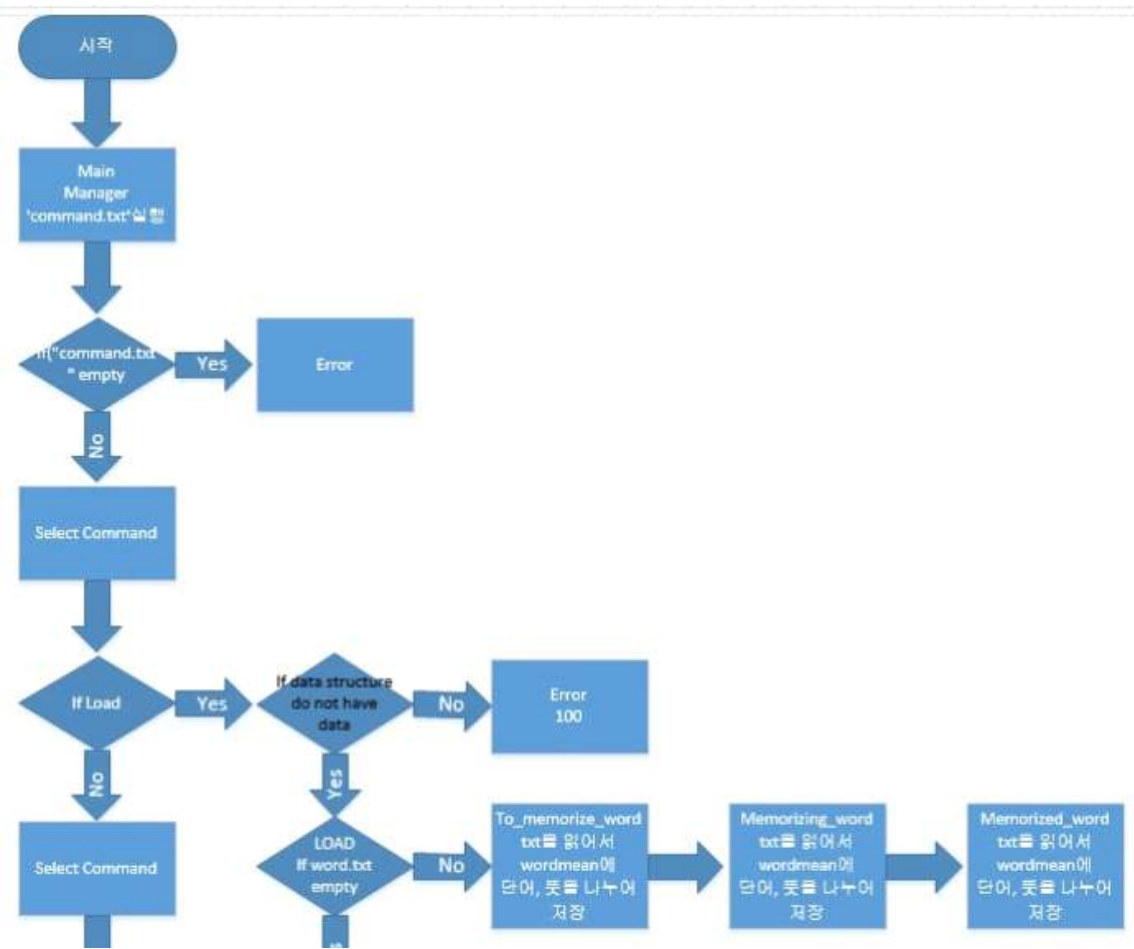
Tree의 특성으로 트리의 순회또한 재귀적인 방법이 가장 보편화 된 방법입니다. 그러나 재귀 함수를 이용한 방법은 메모리를 많이 소모하기 때문에 트리의 크기가 커질 경우 자칫 스택오버플로우(Stack Overflow)가 발생할 수도 있습니다. 따라서 반복적인 루틴으로 탐색하는 것이 좋은 방법입니다.

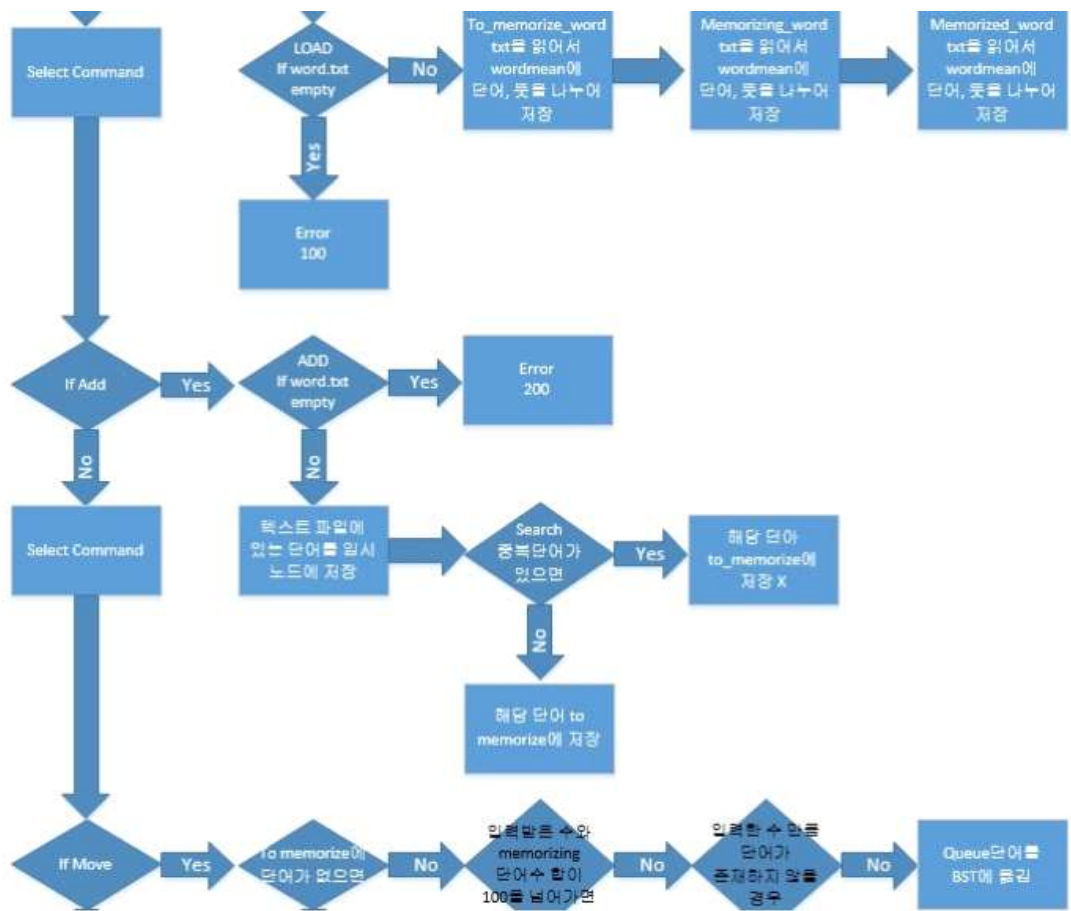
Circular Linked List

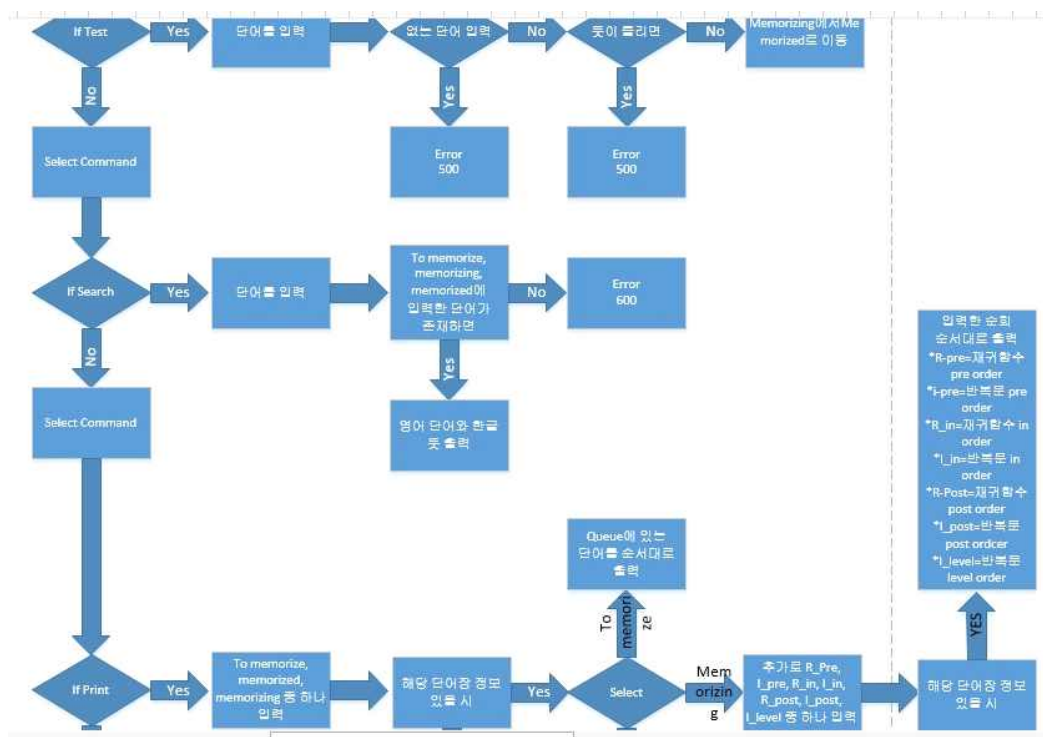
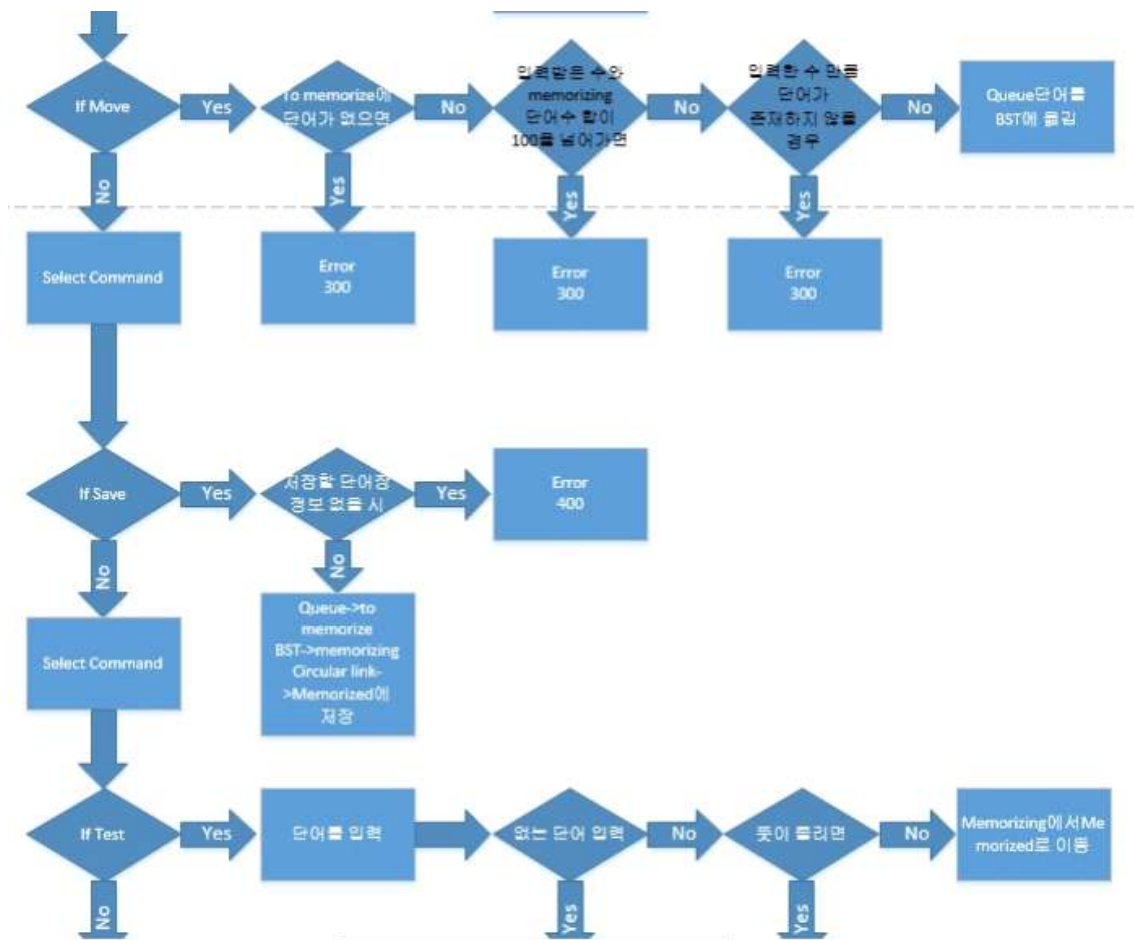


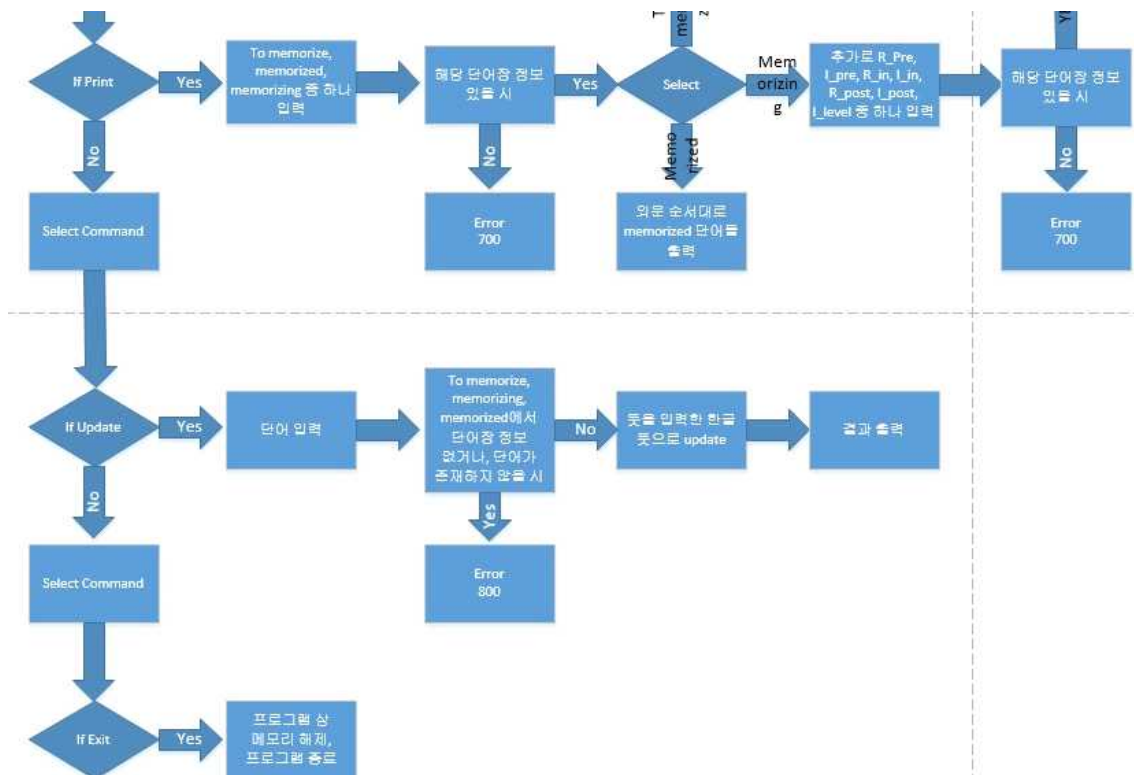
원형 연결리스트의 경우 일반적인 연결리스트와 크게 차이는 안나지만 마지막 노드가 다시 헤더를 가리킨다는 특징이 있습니다. 이렇게 될 경우 삽입연산을 할 때 새로운 노드는 항상 헤더를 가리키게 됩니다. 따라서 리스트의 끝이 항상 헤더를 가리키는 자료구조라고 볼 수 있습니다.

FlowChar:









Algorithm:

Recursive-Pre-ordering

recursive방법은 함수안에서 자기 자신(함수)를 또 다시 호출하는 방식입니다. 트리 자체가 재귀적인 구조이므로 pre ordering을 했을 때, 트리를 전부 순회할 수 있습니다. 동작은 다음과 같습니다.

- root노드부터 출력을 한뒤, 왼쪽 자식노드가 없을때까지 출력합니다.
- 왼쪽 자식노드가 더 이상 존재하지 않을 때, 오른쪽 서브트리를 방문하여 다시 루트노드부터 출력합니다.
- 이 과정을 더 이상 이동할 노드가 존재하지 않을때까지 반복합니다.

Recursive-In-ordering

In-order방식은 이진 탐색트리에서 값을 순서대로 출력해주는 역할을 합니다.

왜냐하면 이진 탐색트리에서 이미 root노드를 기준으로 값이 구간화되기 때문입니다.

따라서 이를 이용하여 왼쪽부터 노드들의 순서대로 출력하면 값이 정렬되게 때문입니다.

트리순회 과정은 다음과 같습니다.

- root노드를 기준으로 왼쪽자식노드가 없을때까지 들어갑니다.

(왼쪽 서브트리를 출력하기 위함)

왼-쪽 자식노드가 더 이상 없다면, 자기자신을 출력하고 다시 루트노드로 돌아옵니다.

- 루트 노드가 현재 노드이므로 현재 노드를 출력하고 오른쪽 노드로 들어갑니다.

오른쪽 서브트리를 출력하기 위함)

Recursive-Post-ordering

Post order방식은 이진 탐색트리에서 root노드가 가장 나중에 출력되는 순회방법입니다.

앞서 설명했던 Inorder방식과는 다르게 왼쪽 서브트리를 출력한 후 root로 다시 돌아오고, 오른쪽 서브트리로 가서 출력을 하는 방식입니다.

트리순회 과정은 다음과 같습니다.

- 왼쪽 자식노드가 더 이상 없을때까지 들어간다

- 왼쪽 자식노드가 없다면, 자기 자신을 출력하고, root노드로 돌아와서 오른쪽 자식노드를 출력한다.

- 마지막으로 root 노드를 출력한다.

- 이러한 작은 routin을 계속 반복하여 재귀적으로 출력합니다.

Iterator-Pre-ordering

앞서 설명했듯이 재귀적으로 트리를 순회하는 것은 메모리자원을 많이 잡아먹는다는 단점이 있습니다. 따라서 이것을 해결하기 위해 Stack이라는 클래스를 만들어 방문할 노드들(부모노드)을 기억하고 순서에 맞게 출력할 것입니다. 과정은 다음과 같습니다.

Pre-ordering으로 출력하는 것은 재귀와 같은 방식입니다. 그러나 접근하는 방식은 조금 다릅니다.

- 먼저 root노드를 스택에 쌓은 뒤, pop을 하여 출력하고 오른쪽 자식노드와 왼쪽자식노드 순서로 스택에 push합니다. (이때 오른쪽 자식노드가 반드시 먼저 들어가야 합니다. Stack은 Last in First out의 구조로 이루어져 있어 먼저 들어간 것이 가장 나중에 나오기 때문입니다.)

- Stack에 노드를 pop하여 데이터를 출력한 뒤, 다시 오른쪽 노드와 왼쪽노드를 순서대로 스택에 push합니다. (노드들을 넣을 때 NULL인 부분은 if문으로 걸러지도록 하여 연결이 끊긴 부분으로 들어가지 않도록 하였습니다.)

- 이 과정을 Stack이 다 비어있을 때까지 반복합니다.

Iterator-In-ordering

recursive-In-ordering과 출력 결과는 같습니다. Inordering또한 Preordering과 마찬가지로 Stack클래스를 사용하여 부모 노드를 기억하고 순회하는 방식을 사용합니다.

과정은 다음과 같습니다.

- 왼쪽 자식 노드가 없을때까지 들어갑니다.

- 왼쪽 자식노드가 존재하지 않는다면, 노드를 출력하고 Pop을 합니다.(이때 pop한 노드는 부모 노드입니다.)

- pop한 노드를 출력한 뒤, 오른쪽 자식노드로 들어갑니다.

- 이 과정을 Stack이 비어있을 때까지 반복합니다.

Iterator-Post-ordering

recursive-post-ordering과 출력 결과는 같습니다. 부모 노드를 기억하기 위해 Stack클래스를 사용합니다.

과정은 다음과 같습니다.

- root노드를 먼저 push한 뒤, 왼쪽 자식노드들이 없을때까지 전부 push합니다.

- 왼쪽 자식노드가 존재하지 않으면, 노드를 출력하고 Stack을 pop합니다.

- pop한 노드를 기준으로 오른쪽 자식노드로 들어가 다시 왼쪽 자식노드가 존재하지 않을때까지 push합니다.

- 만약 오른쪽 자식노드가 존재하지 않는다면 pop을 하여 노드를 출력합니다.(서브트리의 루트노드)

Iterator-Level-ordering

Level ordering은 노드들의 높이를 구간화하여 레벨로 나눈다음, 각각의 레벨의 노드들을 1부터 순서대로 출력하는 것입니다. 이때 주의할 점은 다른 Iterator 방식과는 다르게 Queue를 사용한다는 것입니다. 기억순서가 마지막으로 방문한 노드를 사용하는 것이 아니라 먼저 들어간 노드를 사용하는 것이기 때문입니다.

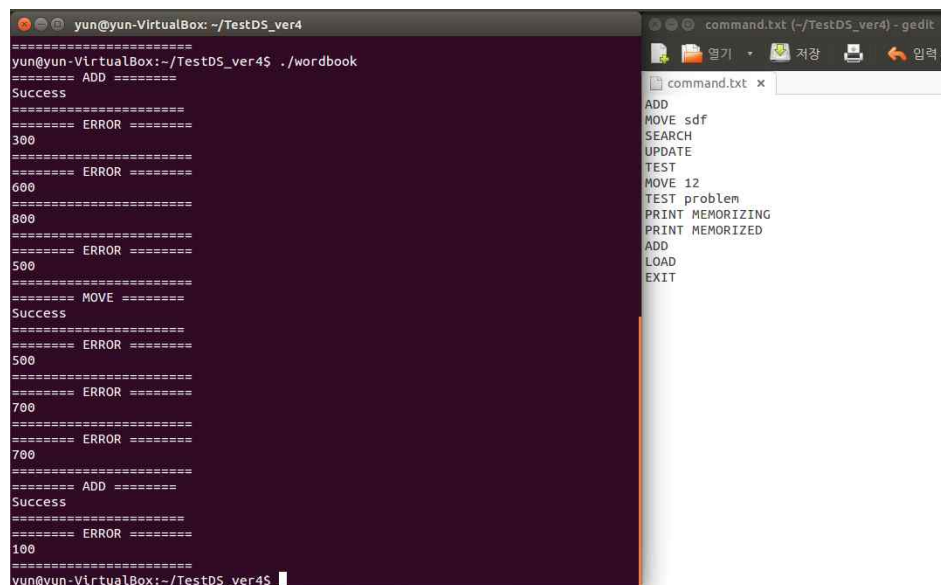
과정은 다음과 같습니다.

-root노드를 push합니다. 그리고 pop하여 데이터를 출력한 후, 왼쪽자식과 오른쪽 자식노드를 push합니다.

-이 과정을 Queue가 비어있을때까지 반복합니다.

이렇게 되면 Queue는 더 이상 남아있지 않을때까지 push와 pop을 반복할 것이고 그 과정에서 기억순서가 Level에 맞춰서 출력되게 됩니다.

Result Screen



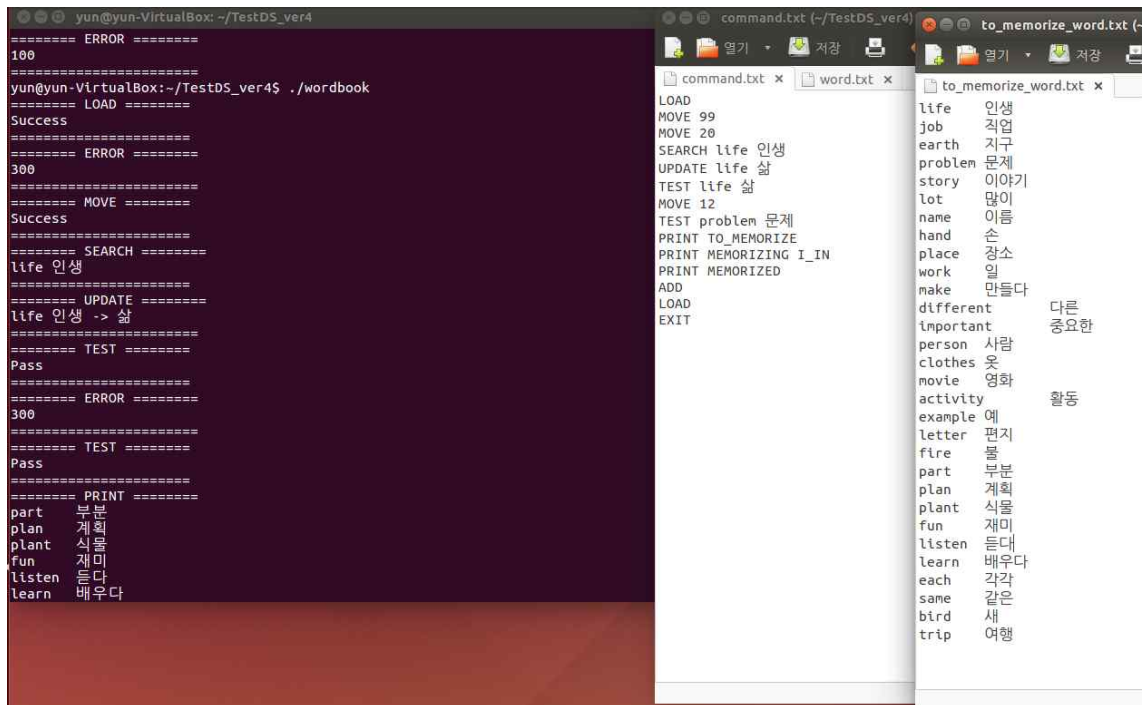
The screenshot shows two windows. The left window is a terminal with the following output:

```
yun@yun-VirtualBox: ~/TestDS_ver4
=====
yun@yun-VirtualBox:~/TestDS_ver4$ ./wordbook
===== ADD =====
Success
===== ERROR =====
300
===== ERROR =====
600
===== ERROR =====
800
===== ERROR =====
500
===== MOVE =====
Success
===== ERROR =====
500
===== ERROR =====
700
===== ERROR =====
700
===== ADD =====
Success
===== ERROR =====
100
=====
yun@yun-VirtualBox:~/TestDS_ver4$
```

The right window is a text editor titled 'command.txt (-/TestDS_ver4) - gedit' containing the following commands:

```
ADD
MOVE sdf
SEARCH
UPDATE
TEST
MOVE 12
TEST problem
PRINT MEMORIZING
PRINT MEMORIZED
ADD
LOAD
EXIT
```

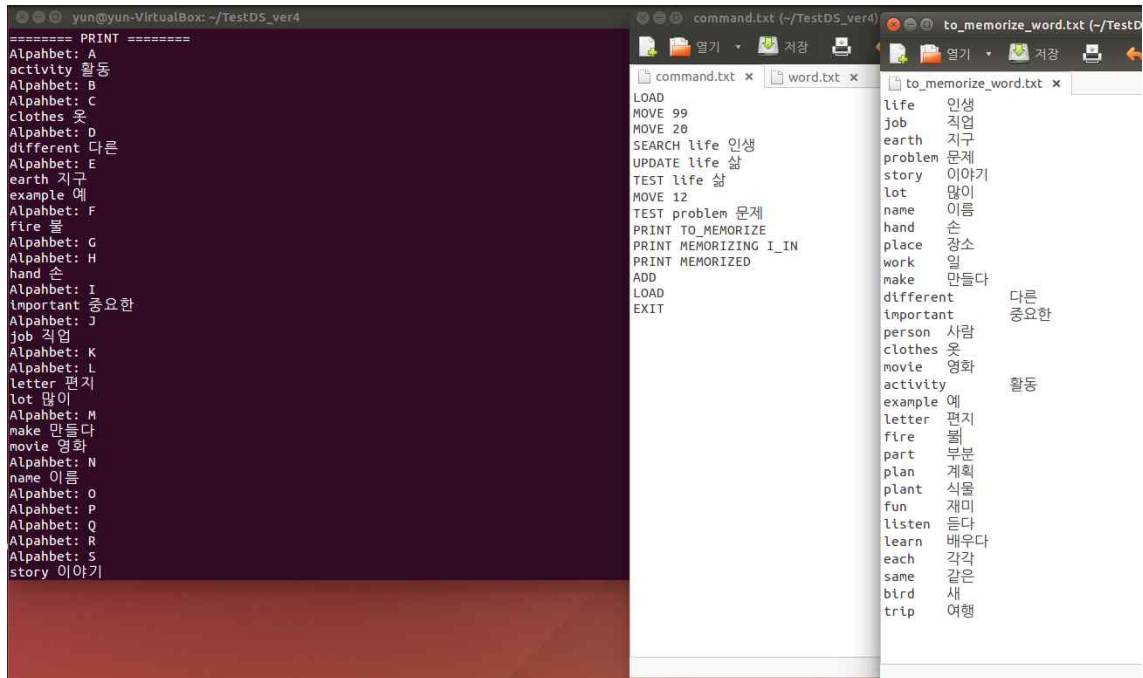
사진에서 확인할 수 있듯이 각각의 명령어에 대해 입력받는 command.txt에서 에러케이스의 문자열을 입력받았을 경우 에러코드를 출력하는 화면입니다. run함수에서 1차적으로 명령어가 잘못됐는지를 확인하고, Manager클래스의 각 멤버함수 내부에서 2차적으로 명령어의 옵션을 판단합니다. 이때 판단하는 것은 읽은 문자열의 첫 번째가 alphabet값인지 아닌지, 숫자인지 아닌지를 판단합니다.



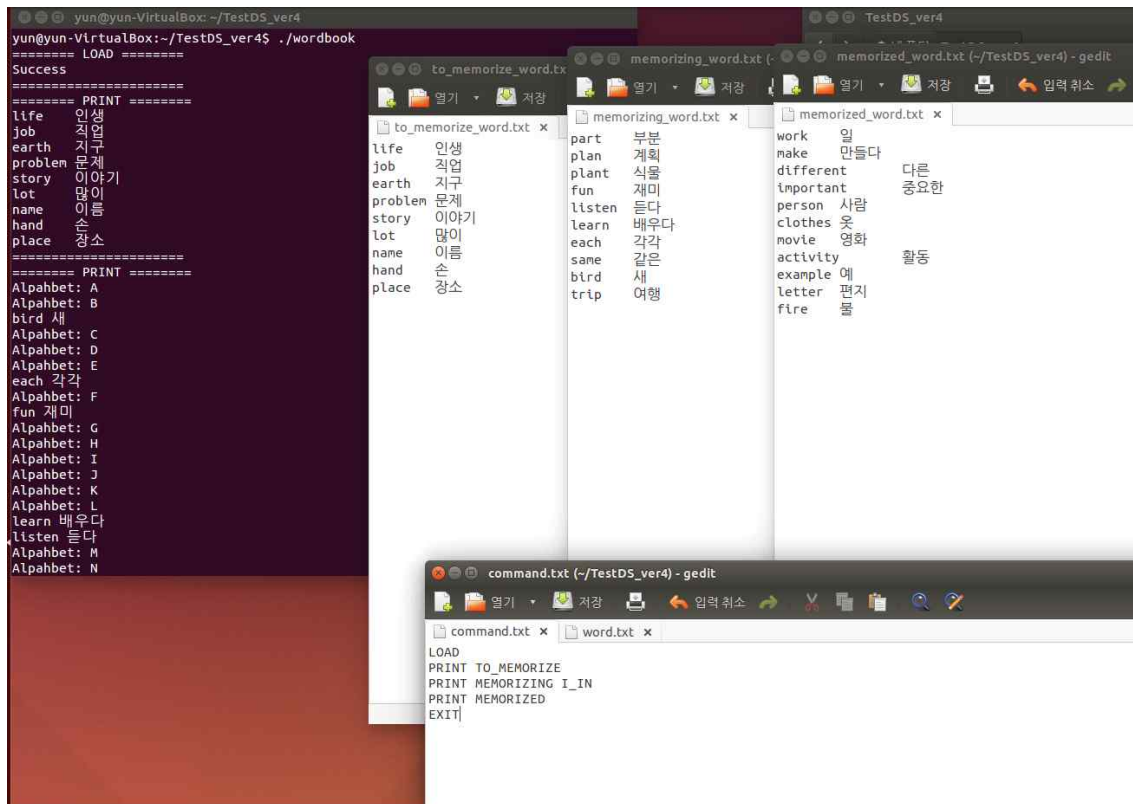
위의 사진은 MOVE와 LOAD의 에러케이스와 출력부분을 나타낸 것입니다. 또한 search와 update, test가 잘되었는지를 확인할 수 있습니다.

memorizing에서 노드의 개수가 99개를 입력받은 것은 성공했지만 20개를 입력받은 것은 실패한 화면을 보여줍니다. 이는 100개를 넘어가는 입력을 받으면 에러를 출력하도록 한 것임을 확인할 수 있습니다.

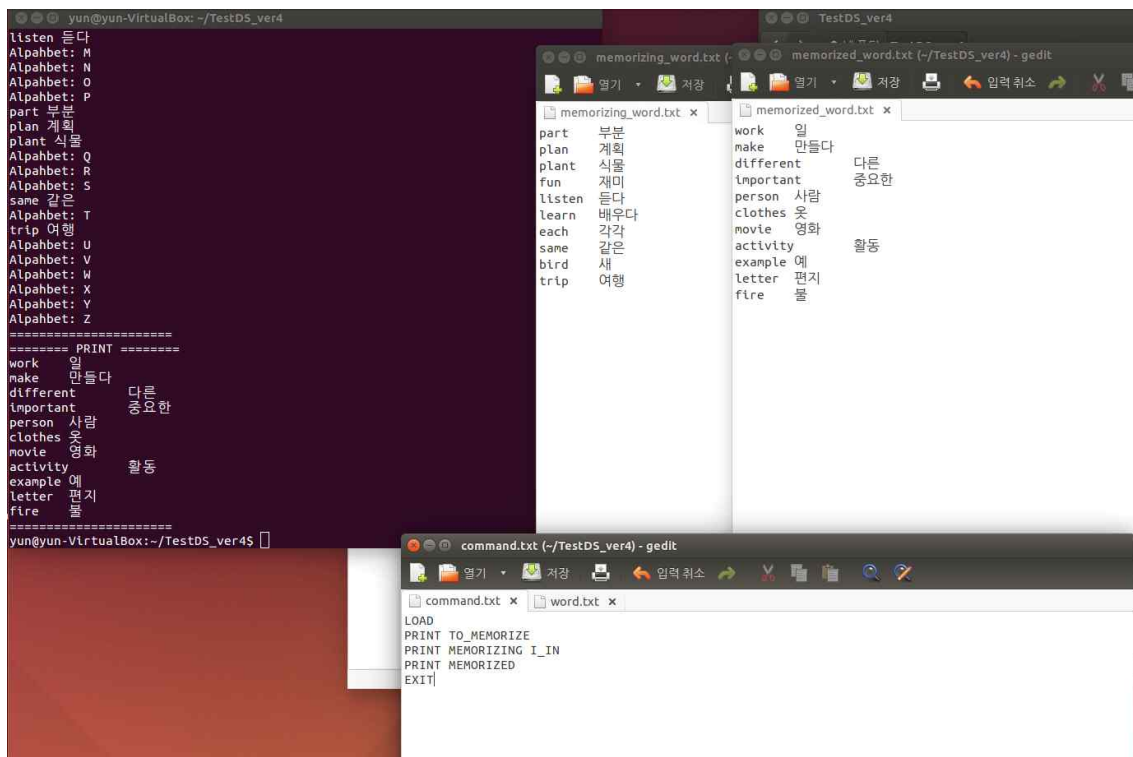
print의 출력화면은 다음 사진에 이어서 설명합니다.



memorizing을 Iterator-In-ordering으로 출력한 것을 나타내는 화면입니다. 알파벳노드의 알파벳을 먼저 출력한 뒤 해당 알파벳노드에 있는 WordNode를 전부 출력한 것을 확인할 수



LOAD명령어로 각각의 자료구조마다 노드들을 insert한 뒤, 전부 출력하는 것을 확인할 수 있는 화면입니다.



위의 출력부분을 전부 나타내는 사진 중, MEMORIZED를 나타내는 화면입니다. LOAD명령어

로 노드들이 온전하게 각각의 자료구조에 insert되어 출력된 것을 확인할 수 있습니다.

Consideration

이승호 : 이번 프로젝트는 굉장히 흥미로운 과제였다. 1학년 1학기, 2학기, 그리고 2학년 1학기를 거치면서 배웠던 모든 것들의 총 집합 같은 느낌이었다. 물론 새로 알게 된 사실들-특히queue나 stack, vector관련해서-배운 것들도 있었지만, 대부분은지난 학기에 배웠던 것들이면서, 순회회로를 짜거나 하는데linked list로 짜서 순서대로 출력하거나, 정해진 순회대로 출력하거나, 아니면 알파벳 순서로 출력하는 등 다양한 방식으로 출력하게 하기 위해 우리가 배웠던 모든 지식을 총동원해서 써야 했기 때문이다. search의 경우는 queue, bst, cll 모두 확인해서 같은 단어가 있는지 확인하고, 있다면 영단어와 뜻을 모두 출력해야 했기에 저번 1학기에서 메모장 사전 검색 과제와 굉장히 유사한 점이 많아서 그때의 경험이 많이 도움 됐다.

홍종현 : 이번에 insert와 searh를 맡으면서 이진트리의 입력과 탐색에 대해서 알아보는 시간을 가졌습니다. tree에서의 전위순회와 중위순회 후위순회 등 다양한 상황에 대해서도 고려해보았습니다.

윤좌홍 : BST를 순회하면서 메모리를 유한하게 생각하여 좀 더 메모리의 낭비 없이 효율적으로 프로그래밍하는 것에 대해 생각해보게 되는 계기가 되었습니다. 특히 메모리 릭(leak)이 발생하는지 확인하기 위해 _CrtSetReportMode() 라는 함수를 사용하여 검사한 결과 delete를 맞게 다 해주었는데도 leak이 감지된다는 것을 확인하고 이유를 알기 위해 검색해보았습니다. 그 결과 STL의 경우 leak검사에 감지될 수 있다는 것을 알게되어 더 많은 것을 배우게 된 계기가 된 것 같습니다.

이번 프로젝트에서 힘들었던 점은 팀원의 역할을 나누기가 힘들었습니다. 각자의 역할분배를 하려 하는데 다른 팀원들이 프로그래밍 실력이 부족하여 쉽사리 역할을 맡길 수 없었습니다. 또한 결과물이 제대로 나오지 않아 결국 구현을 한 명이 다 맡아서 했기 때문에 힘든 것 같습니다.

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자기 점수(10)
이승호	보고서 작성 일부분	5
홍종현	x	5
윤좌홍	그 외의 부분	9.5