

데이터 구조 실습

PROJECT 1

BST

Professor	이기훈 교수님
Department	Computer engineering
Student ID	[설계o, 실습 화] 2013722064, [설계o, 실습 목] 2013722063, [설계o, 실습 화] 2014722071
Name	김시훈(조장), 고정원, 신지영
Class	Tue 3,4 / Thu 3,4
Date	2016. 10. 07

▶ Introduction

[프로젝트 내용에 대한 설명]

1차 프로젝트는 BST 자료구조를 사용해 단어장 프로그램을 구현하는 프로젝트이다.
외워야할 단어(to memorize), 외우고 있는 단어(memorizing), 외운 단어(memorized) 세 구조로 나누어
각각이 Queue, BST, Circular LinkedList의 자료구조를 사용해서 프로그램이 동작하도록 하는 것이다.

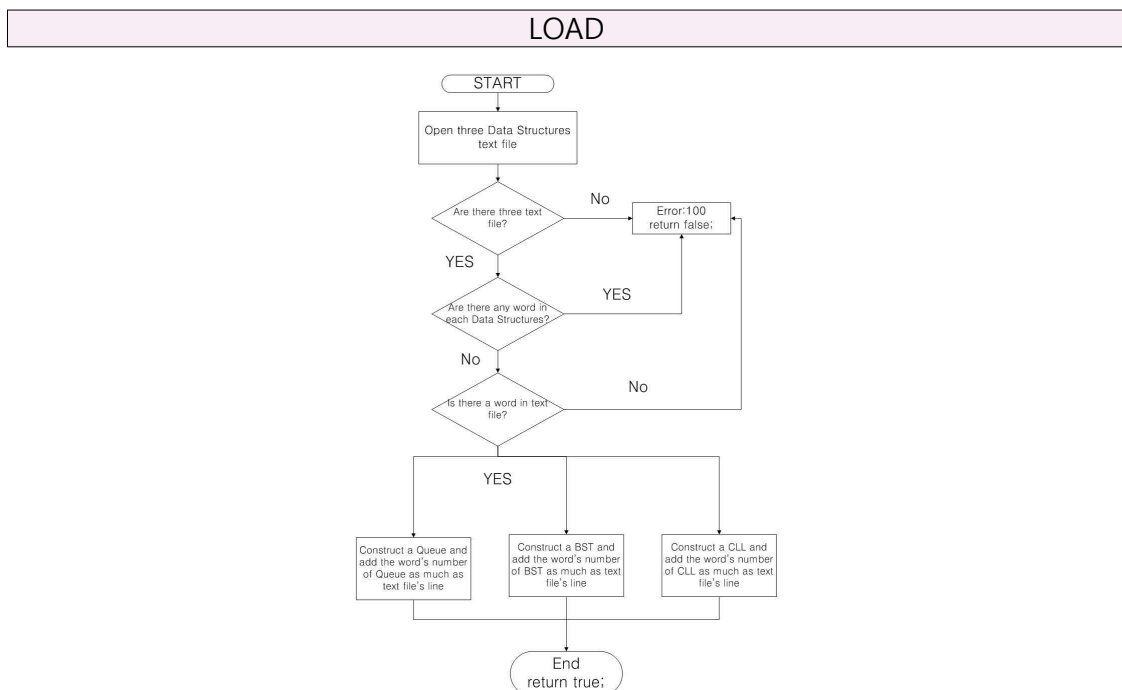
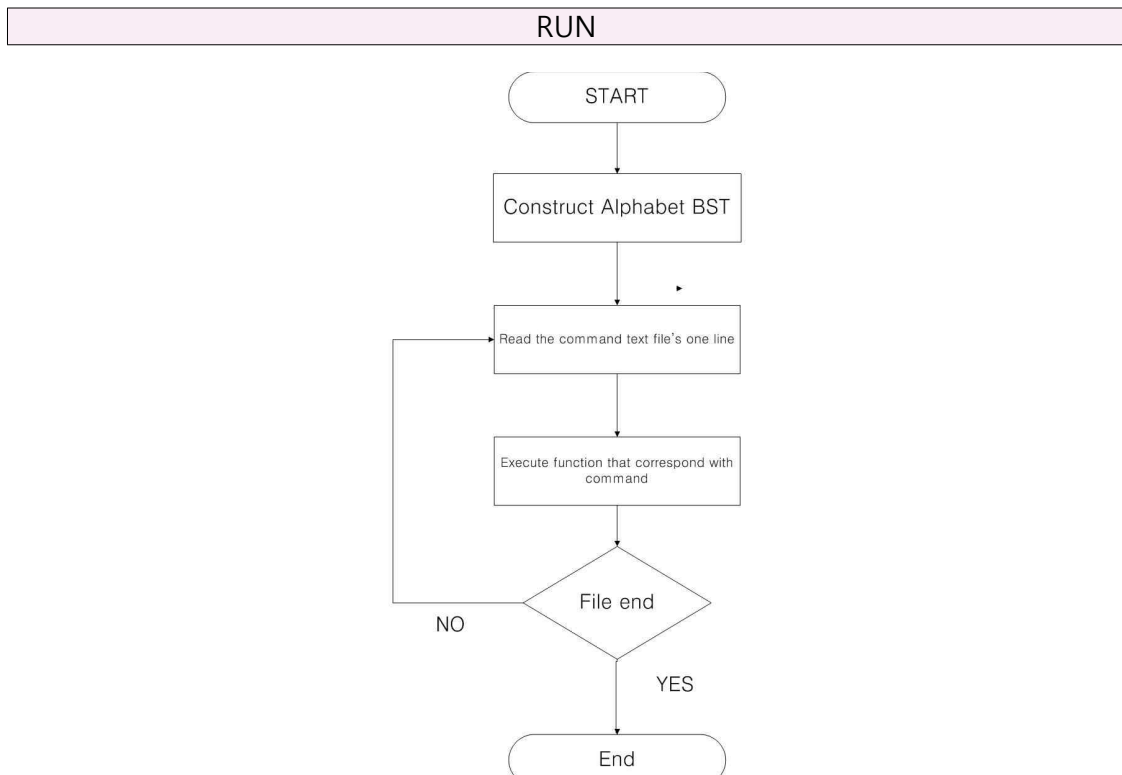
BST는 알파벳을 노드의 값으로 하는 Alphabet BST와 단어를 노드의 값으로 하는 WordBST가
별도로 구성되어있다. WordBST에서는 최대 존재할 수 있는 노드의 수가 100개이다.

각 기능을 수행하는 command(LOAD, ADD, MOVE, SAVE, TEST, SEARCH, PRINT,
UPDATE, EXIT)가 지정되어있고, command를 읽으면 그 해당 command에 따라 동작하도록
하여야 하는데, 이 command는 텍스트 파일로 입력 받아 읽어 수행하도록 한다.

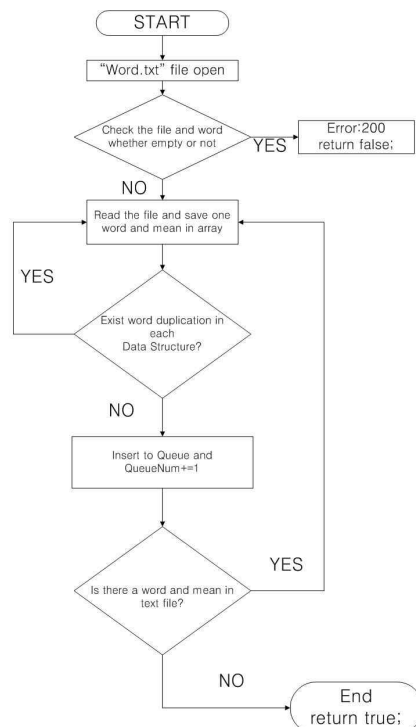
또한, 로그파일을 생성해서 그 파일에 프로그램 수행 결과가 기록되도록 한다.

► Flowchart

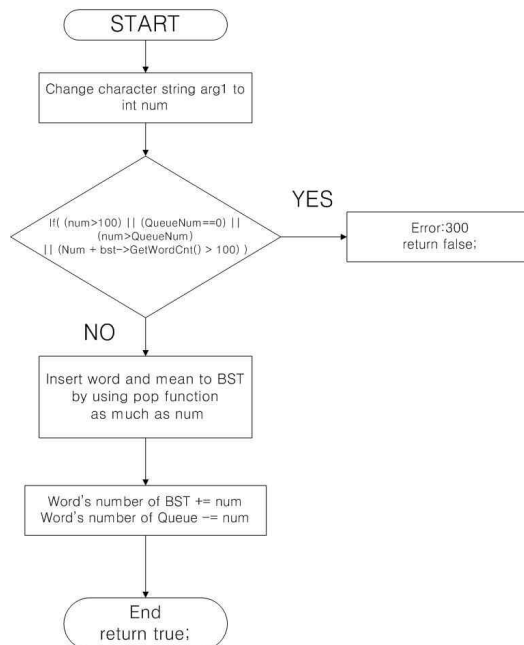
[설계한 프로젝트의 플로우차트를 그리고 설명]



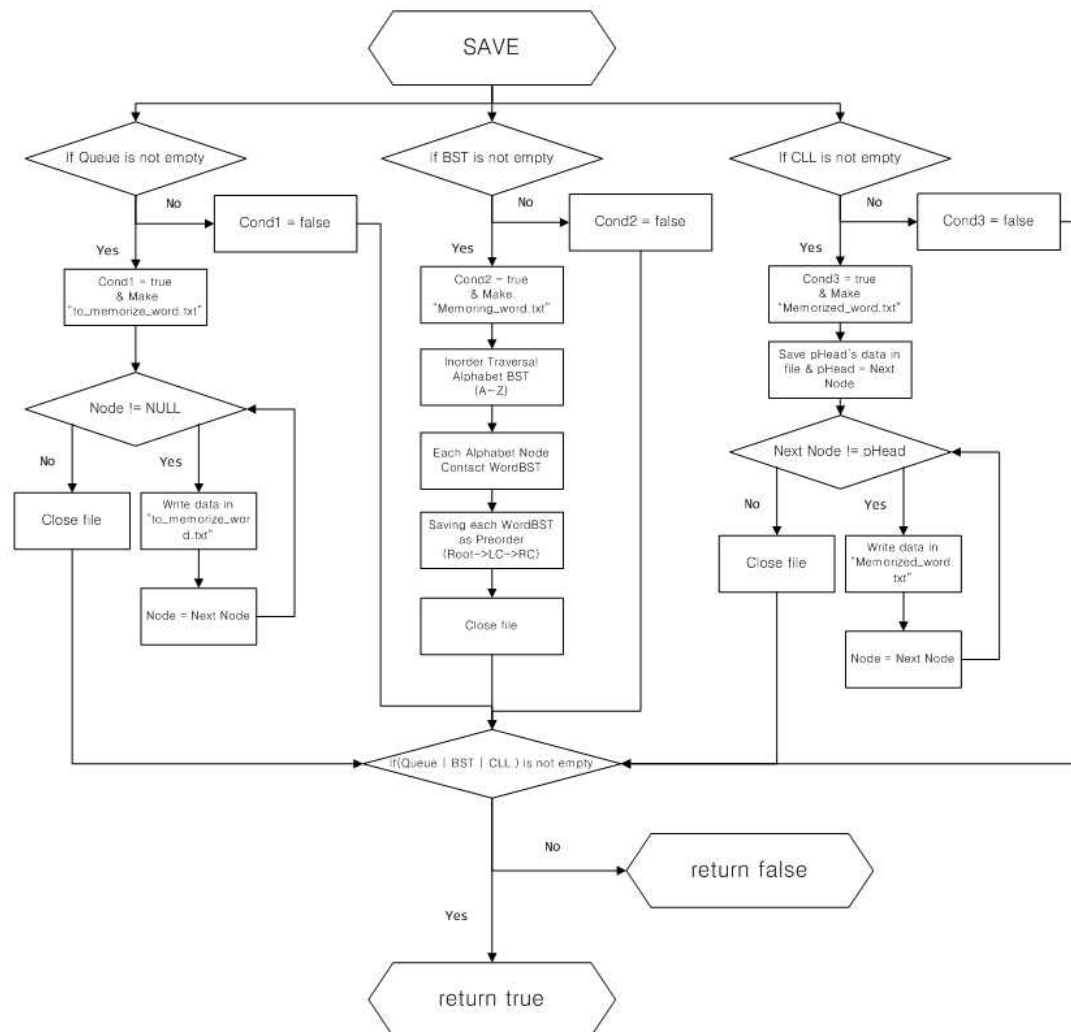
ADD



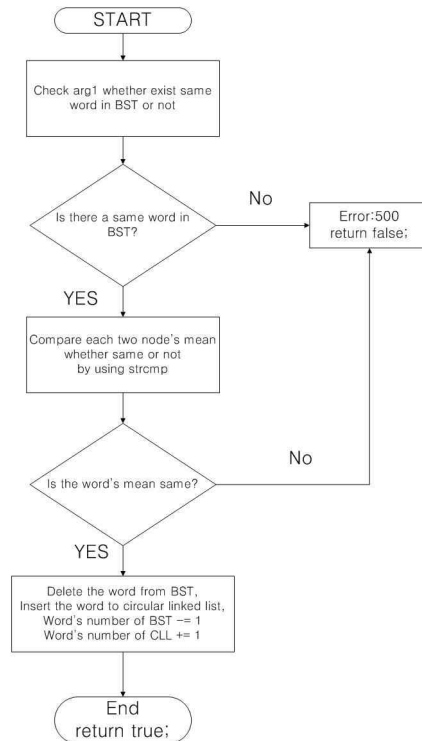
MOVE



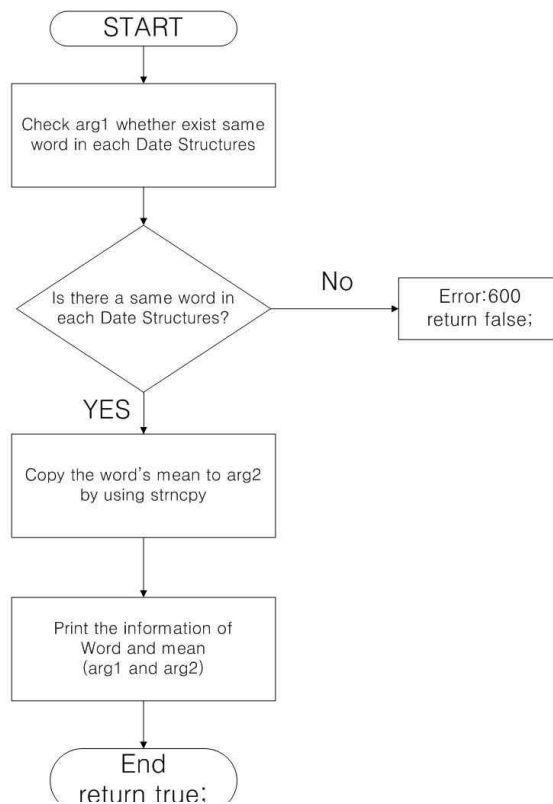
SAVE



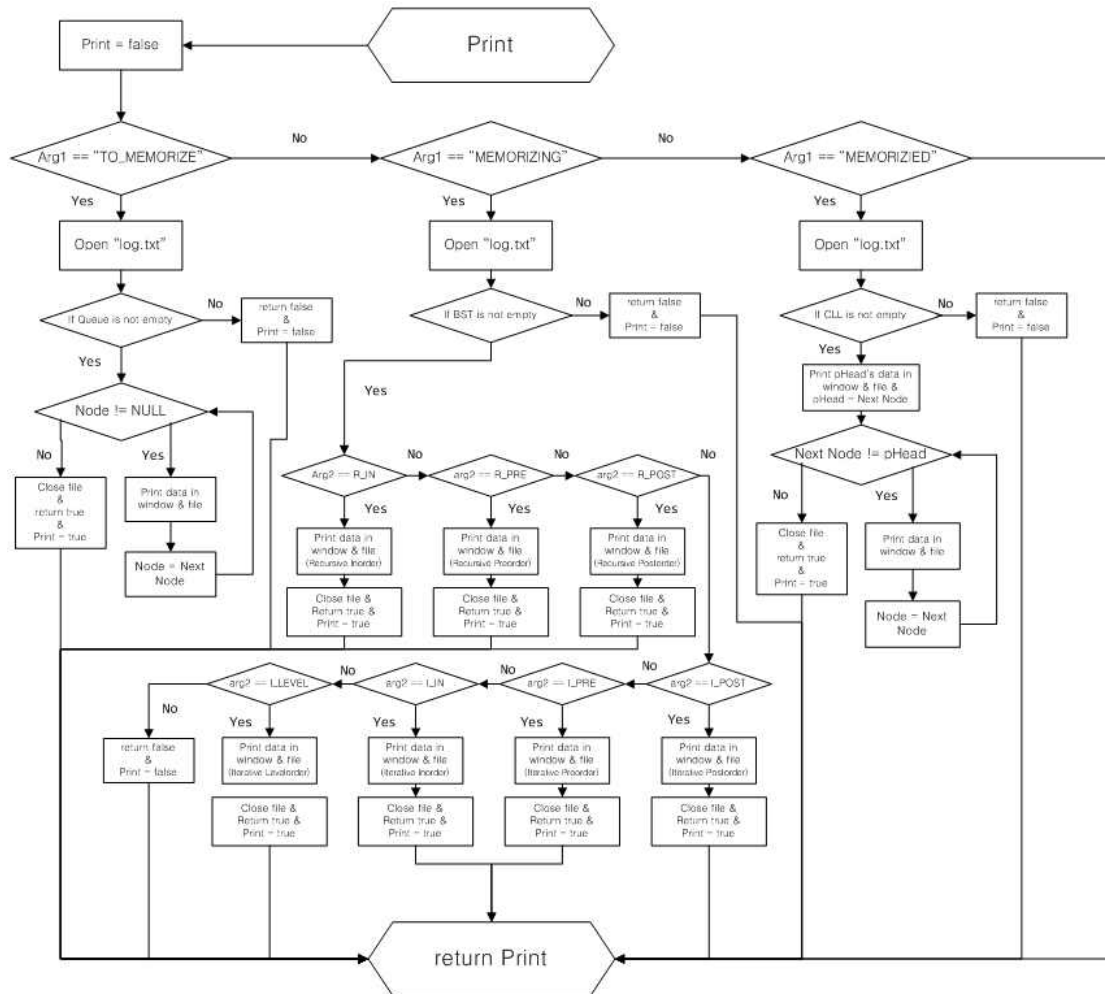
TEST



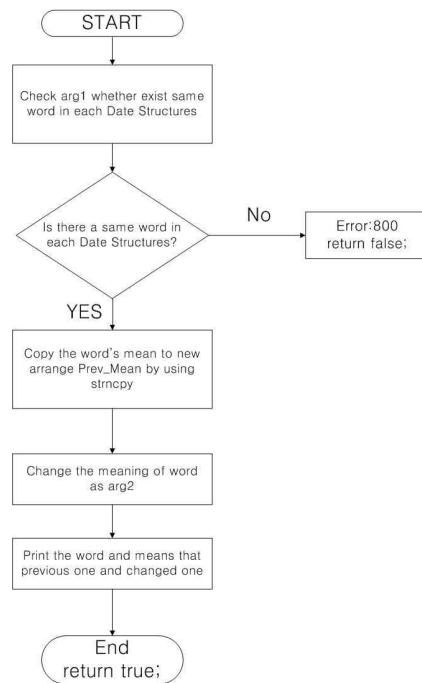
SEARCH



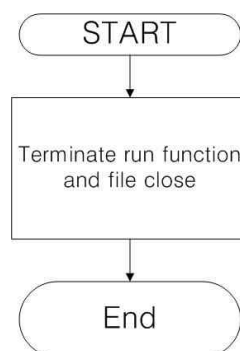
PRINT



UPDATE



EXIT



▶ Algorithm

[프로젝트에서 사용한 알고리즘의 동작을 설명]

LOAD	* command 형식 LOAD
------	-----------------------------

- 현재 저장되어있는 세 개의 자료구조에 대해 해당하는 텍스트 파일을 읽어 각 자료구조에 이전과 동일한 구조(연결순서)를 갖도록 하는 command.
- 3개의 텍스트 파일(to_memorize_word.txt, memorizing_word.txt, memorized_word.txt)이 존재하지 않거나 자료구조에 이미 데이터가 들어 있을 경우에 에러 코드 출력함.

ADD	* command 형식 ADD
-----	----------------------------

- 외워야할 단어들(to memorize)이 단어장에 기입이 되도록 하는 command.
- 가장 처음 프로그램을 시작하자마자 alphabet BST를 주어진 형식대로(P, H, X, D, L, T, Z, B, F, J, N, R, V, Y, A, C, E, G, I, K, M, O, Q, S, U, W 순으로) 구축을 해놓은 뒤에 이 command를 읽게 되면 단어와 그 단어의 뜻을 포함하는 노드를 생성하여 해당 노드를 Queue에 insert해줌.
- 단어 텍스트 파일(word.txt)이 존재하지 않거나 이 파일에 단어가 존재하지 않을 경우 에러 코드 출력함.

MOVE	* command 형식 MOVE [the number of node to move]
------	--

- 사용자가 입력한 수만큼 외워야할(to memorize) 단어장의 단어들을 외우고 있는(memorizing) 단어장에 옮겨지도록 하는 command.
- 외우고 있는 단어장의 자료구조는 WordBST로, 그 자료구조 안에 최대 100개의 노드밖에 가지지 못하므로 수행했을 시에 초과가 되거나 입력한 수만큼 단어가 자료구조(Queue)에 존재하지 않을 경우 에러 코드 출력함.

SAVE	* command 형식 SAVE
------	-----------------------------

- 자료구조들 안에 저장되어 있는 것들을 3개의 텍스트 파일(to_memorize_word.txt, memorizing_word.txt, memorized_word.txt)에 저장하도록 하는 command.
- memorizing_word.txt 파일에 단어를 알파벳 순으로 저장하면 LOAD하게 되면 skewed binary tree가 되므로 preorder를 이용함.
- 단어장 정보가 존재하지 않을 경우 에러 코드 출력함.

TEST	* command 형식 TEST [word] [mean]
------	---

- 단어를 외웠는지 테스트 한 후에 단어에 해당하는 뜻이 맞을 경우 그 해당 단어를 외운(memorized) 단어장으로 이동시키는 command.
- 입력한 단어가 존재하지 않거나 단어의 뜻이 알맞지 않을 경우 에러 코드 출력함.

SEARCH	* command 형식 SEARCH [word]
--------	--------------------------------------

- 입력받은 단어를 이용해 자료구조들을 검색하여 단어와 뜻을 함께 출력하는 command.
- 입력받은 단어가 존재하지 않을 경우 에러 코드 출력함.

PRINT	* command 형식 PRINT [vocabulary name] [traversal order]
-------	---

- 입력받은 단어장에 있는 단어들을 출력하는데 MEMORIZING의 경우에만 함께 입력받은 트리 순회 방법(R_PRE, I_PRE, R_IN, I_IN, R_POST, I_POST, I_LEVEL)에 따라 출력하는 command.
- 입력받은 단어장 정보가 존재하지 않을 경우 에러 코드 출력함.

UPDATE	* command 형식 UPDATE [word] [mean]
--------	---

- 입력받은 단어에 해당하는 뜻을 새로운 뜻으로 수정한 뒤에 단어와 뜻을 출력하는 command.
- 입력받은 단어가 존재하지 않거나 단어장 정보가 존재하지 않을 경우 에러 코드 출력함.

EXIT	* command 형식 EXIT
------	-----------------------------

- 프로그램 상의 메모리를 모두 해제시키고 프로그램을 종료시키는 command.

main.cc
<pre> int main() { call run function of manager class. } </pre>
Manager.cc
<pre> void Manager::run() { Alphabet BST->Insert() files open. read one line and save token for command and parameter. // compare token with command if LOAD { if parameter is exceeded, Print Error. else call LOAD() function. } else if ADD { if parameter is lack, Print Error. else if parameter is exceeded, Print Error. else call ADD() function. } else if MOVE { if parameter is lack, Print Error. else if parameter is exceeded, Print Error. else copy parameter, call MOVE() function. } else if SAVE { if parameter is exceeded, Print Error. else call SAVE() function. } else if TEST { if parameter is lack, Print Error. else if parameter is exceeded, Print Error. else copy parameter, call TEST() function. } else if SEARCH { if parameter is lack, Print Error. else if parameter is exceeded, Print Error. else copy parameter, call SEARCH() function. } else if PRINT { if parameter is lack, Print Error. </pre>

```

        else {
            if first parameter == TO_MEMORIZE {
                if parameter is exceeded, Print Error.
                else copy parameter, call PRINT() function.
            }
            else if first parameter == MEMORIZING {
                if parameter is lack, Print Error.
                else if parameter is exceeded, Print Error.
                else copy parameter, call PRINT() function.
            }
            else if first parameter == MEMORIZED {
                if parameter is exceeded, Print Error.
                else copy parameter, call PRINT() function.
            }
        }
    }
}

else if UPDATE {
    if parameter is lack, Print Error.
    else if parameter is exceeded, Print Error.
    else copy parameter, call UPDATE() function.
}

else if EXIT {
    break;
}

else command not correct.
files close.
}

bool Manager::LOAD() {
    files open.
    if files not exist, Print Error. return false;
    if each of data structure is not empty, Print Error.

    // set data in each data structure reading three text file
    while(Not end of Queue file) Push word and mean in Queue and counting.
    while(Not end of BST file) Push word and mean in BST and counting.
    while(Not end of CLL file) Push word and mean in Circular LinkedList and
counting.
    files close.
    return true;
}

```

```

bool Manager::ADD() {
    files open and if not exist file, Print Error.
    else {
        read one line.
        if empty file, Print Error. return false;
        else {
            token for word and mean.
            // exist : check_dup=1, not exist : check_dup=0
            check whether word exist or not in three data structure.
            if all of check_dup is 0, set word and mean in Queue.

            while not end of file {
                read one line.
                token for word and mean.
                check whether word exist or not in three data structure.
                // exist : check_dup=1, not exist : check_dup=0
                if all of check_dup is 0, set word and mean in Queue.
            }
            return true;
        }
    }
    files close.
}

bool Manager::MOVE() {
    file open using subsequent writing.
    // arg1 : the number of node to move
    change parameter of string type into integer type.

    if parameter is larger than 100, Print Error. return false;
    if Queue is empty, Print Error. return false;
    if node to move(parameter) + node in Word BST if larger than 100, Print
Error. return false;
    if parameter is larger than node of Queue, Print Error. return false;

    move node into BST.
    file close.
    return true;
}

bool Manager::SAVE() {
    open file("log.txt")

```

```

bool cond1, cond2, cond3

cond1 = execute queue's Save() function
cond2 = execute bst's Save() function
cond3 = execute cll's Save() function

if (cond1 || cond2 || cond3) {
    print success on window & file
    close file
    return true
}
else {
    print error on window & file
    fout.close();
    return false
}
}

bool Manager::TEST() {
    file open using subsequent writing.

    // arg1 : word, arg2 : mean
    if word(arg1) is not exist in BST, Print Error. return false;
    else {
        if word's mean is not correct, Print Error. return false;
    }
    // word's mean(arg2) is correct (test pass)
    Delete the node in BST and set count - 1.
    Insert in Circular LinkedList and set count + 1.
    file close.
    return true;
}

bool Manager::SEARCH() {
    file open using subsequent writing.

    // arg1 : word
    // exist : search=1, not exist : search=0
    check whether word exist or not in three data structure.

    if at least one of search is 1, Print word and mean of the node. return
true;

```

```

else Print Error. return false;
file close.
}
Bool Manager::PRINT() {
    bool print = false
    open file("log.txt")

    // arg1 : data structure name, arg2 : order
    if arg1 is "TO_MEMORIZE" {
        if(Queue doesn't have any word) {
            print error on window & file
            file close.
            return false;
        }
        else Execute Queue's Print function
    }
    else if(arg1 == "MEMORIZING") {
        if(BST doesn't have any word) {
            print error on window & file.
            file close.
            return false;
        }
        else Execute BST's Print function.
    }
    else if(arg1 == "MEMORIZED") {
        if(CLL doesn't have any word) {
            print error on window & file
            file close.
            return false;
        }
        else Execute CLL's Print function.
    }

    if(print == false) {
        print error on window & file.
        file close.
        return false;
    }
    else {
        file close.
    }
}

```

```

        return print;
    }
}

bool Manager::UPDATE() {
    file open using subsequent writing.

    // arg1 : word, arg2 : mean
    // exist : search=1, not exist : search=0
    check whether word exist or not in three data structure.

    if at least one of search is 1, Change old mean of the node into new
    mean(arg2).
    else Print Error. return false;
    file close.
}

```

Queue.cc

```

void Queue::Push(WordNode* node) {
    // p : current node
    p = pHead.
    if Queue is empty, set the input node to pHead.
    else {
        for(move to final node)
            final node connect to the input node.
    }
    return;
}

WordNode* Queue::Pop() {
    if Queue is empty, return NULL;
    else {
        set pCur to pHead.
        copy word and mean to temp node.
        hand over pHead to next node.
        delete pCur; return temp;
    }
}

WordNode* Queue::Search(char* word) {
    // temp : current node, arr1 : word of current node, arr2 : word to
    search
    temp = pHead;
    if Queue is empty, return NULL;
    for(as much as length of word) {

```



```

        change capital letter(word to search) into small letter.
    }
    while(1) {
        initialize array.
        change capital letter(word of current node) into small letter.

        if arr1 == arr2, return temp.
        else move to next node.

        if temp == NULL, return NULL; // search fail
    }
}

bool Queue::Print() {
    if Queue is not empty {
        open file("log.txt")
        temp = pHead

        for( ; temp ; temp = temp->GetNext()) {
            print temp's Word, Mean on window and file.
        }
        file close.
        return true;
    }
    else return false;
}

bool Queue::Save() {
    open file("to_memorize_word.txt")
    if (pHead != '\0') {
        WordNode* Cur = pHead;

        for (; Cur != NULL ; Cur = next node)
            print Cur's data on file
        close file
        return true
    }
    else {
        close file
        return false
    }
}
}

```

AlphabetBST.cc

```

void AlphabetBST::Insert(AlphabetNode* node) {
    // p : current node, pp : parent node of current node
    p = root;
    while(p) {
        copy p to pp.
        if input node alphabet < current node alphabet, move to left child.
        if input node alphabet > current node alphabet, move to right child.
    }
    if BST is not empty {
        if input node alphabet < parent node alphabet, set input node to left
child of parent node.
        else set input node to right child.
    }
    else set input node to root.
}
AlphabetNode* AlphabetBST::Search(char alphabet) {
    currentNode = root

    while(currentNode) {
        if 'alphabet' is small letter {
            if currentNode alphabet > capital letter of 'alphabet', move to left
child.

            else if currentNode alphabet < capital letter of 'alphabet', move
to right child.
            else return currentNode;
        }
        else {
            if currentNode alphabet > 'alphabet', move to left child.
            else if currentNode alphabet < 'alphabet', move to right child.
            else return currentNode;
        }
    }
    return NULL;
}
bool AlphabetBST::Print(char* order) {
    if(BST doesn't have any word) {
        if(order == "R_PRE") {
            execute R_PRE function.
            return true;
        }
    }
}

```

```

        else if(order == "I_PRE") {
            execute I_PRE function.
            return true;
        }
        else if(order == "R_IN") {
            execute R_IN function.
            return true;
        }
        else if(order == "I_IN") {
            execute I_IN function.
            return true;
        }
        else if(order == "R_POST") {
            execute R_POST function.
            return true;
        }
        else if(order == "I_POST") {
            execute I_POST function.
            return true;
        }
        else if(order == "I_LEVEL") {
            execute I_LEVEL function.
            return true;
        }
        else return false;
    }
    else return false;
}

int AlphabetBST::GetWordCnt() {
    // WordCnt : the number of BST
    return WordCnt;
}

void AlphabetBST::SetWordCnt(int word_cnt) {
    WordCnt = word_cnt;
}

void AlphabetBST::R_IN(AlphabetNode* root, char * order) {
    AlphabetNode* CurrentNode = root
    if (CurrentNode) {
        recursive inorder.
        execute CurrentNode's WordBST's Print function.
    }
}

```

```

    }
}

void AlphabetBST::R_PRE(AlphabetNode * root, char * order) {
    AlphabetNode* CurrentNode = root
    if (CurrentNode) {
        recursive preorder.
        execute CurrentNode's WordBST's Print function
    }
}

void AlphabetBST::R_POST(AlphabetNode * root, char * order) {
    AlphabetNode* CurrentNode = root
    if (CurrentNode) {
        recursive postorder
        execute CurrentNode's WordBST's Print function
    }
}

bool AlphabetBST::I_IN(char * order) {
    if (root == NULL)
        return false
    else {
        Declare Array of AlphabetNode*
        Declare int type variable cnt , top and initialize to 0

        AlphabetNode* CurrentNode = root
        while (1) {
            while (CurrentNode) {
                arr[cnt] = CurrentNode
                CurrentNode = left leaf node
                increase cnt
                increase top
            }

            if (cnt == 0)
                return true
            decrease top
            initialize CurrentNode to arr[top]
            initialize arr[cnt] to NULL
            decrease cnt
            execute CurrentNode's WordBST's Print function
            CurrentNode = right left node
        }
    }
}

```

```

    }
}
}
bool AlphabetBST::I_PRE(char * order) {
    if (root == NULL)
        return false
    else {
        Declare Array of AlphabetNode*
        Declare int type variable cnt , top and initialize to 0

        AlphabetNode* CurrentNode = root

        while (1) {
            while (CurrentNode) {
                initialize arr[cnt] to CurrentNode
                execute CurrentNode's WordBST's Print function
                CurrentNode = CurrentNode's left leaf node
                increase cnt
                increase top
            }
            if (cnt == 0)
                return true
            decrease top
            initialize CurrentNode to arr[top];
            initialize arr[cnt] to NULL
            decrease cnt
            CurrentNode = CurrentNode's right leaf node
        }
    }
}

bool AlphabetBST::I_POST(char * order) {
    if (root == NULL)
        return false
    else {
        Declare Array of AlphabetNode*
        Declare int type variable cnt and initialize to 0
        AlphabetNode* CurrentNode = root

        while (1) {

```

```

        while (CurrentNode exist) {
            if (Cur's right child exist) {
                arr[cnt] = Cur's right child
                increase cnt
            }
            arr[cnt] = CurrentNode
            CurrentNode = Cur's left leaf node
            increase cnt
        }
        if (cnt == 0)
            return true
        decrease cnt
        CurrentNode = arr[cnt]
        arr[cnt] = NULL
        decrease cnt

        if (CurrentNode have right child & arr[cnt] == right child) {
            arr[cnt] = CurrentNode
            CurrentNode = Cur's right leaf node
            increase cnt
        }
        else {
            execute Cur's WordBST's Print function
            CurrentNode = NULL
            increase cnt
        }
    }
}

bool AlphabetBST::I_LEVEL(char * order) {
    Declare Array of AlphabetNode*
    Declare int type variable cnt, frt and initialize to 0
    AlphabetNode* CurrentNode = root
    if (root) {
        while (CurrentNode) {
            execute Cur's WordBST's Print function
            if (Cur's left child exist) {
                arr[cnt] = Cur's left leaf node
                increase cnt
            }
        }
    }
}

```

```

        if (Cur's right child exist) {
            arr[cnt] = Cur's right leaf node
            increase cnt
        }
        if (frt == cnt)
            return true
        CurrentNode = arr[frt]
        arr[frt] = NULL
        increase frt
    }
}
return false
}
bool AlphabetBST::Save() {
    if(BST doesn't have any word)
        return false
    else {
        open file ("memorizing_word.txt")
        close file
        execute inSave(root)
        return true
    }
}
bool AlphabetBST::inSave(AlphabetNode * root) {
    AlphabetNode* Cur = root
    if (Cur) {
        recursive inorder
        execute Cur's WordBST's Save()
    }
}
void AlphabetBST::Clear(AlphabetNode * root) {
    CurrentNode = root
    if AlphabetBST is not empty, call Clear(leftchild & rightchild) function and
    delete CurrentNode;
}

```

WordBST.cc

```

void Insert(WordNode * node) {
    // p : current node, pp : parent node of current node
    change capital letter(new node) into small letter.
}

```

```

while(p) {
    copy p to pp node.
    change capital letter(current node) into small letter.

    if current node word > new node word, move to left child.
    else if current node word < new node word, move to right child.
}

if BST is not empty {
    initialize array.
    change capital letter into small letter.

    if parent node word > new node word, set new node to left child.
    else set new node to right child.
}
}

WordNode * WordBST::Delete(char * word) {

    while (Cur) {
        if (find target)
            Loop escape
        if ( word < Cur's data ) {
            Udata parent node & Cur -> left leaf node
        }
        else {
            Udata parent node & Cur -> right leaf node
        }
    }

    if (BST has only root Node) {
        copy data to temp node
        delete root node
    }
    else if (target is leaf node) {
        copy data to temp node
        if (Cur is Parent's left child) {
            set parent's left child to NULL
            delete current node
        }
        else {

```



```

        set parent's right child to NULL
        delete current node
    }
}

else if (target has left child) {
    copy data to temp node
    declare left sub tree's root node tCur

    if (tCur has two child) {
        while (tCur has right child) {
            Udata tCur's parent & tCur -> right leaf node
        }

        copy tCur's data to Cur

        if (tCur is leaf Node) {
            set tparent's right child to NULL
            delete tCur
        }
        else {
            set tparentNode's rightChild to tCur's leftChild
            delete tCur;
        }
    }

    else if (Cur is root) {
        set root to Cur's LeftNode
        delete current node
    }
    else if (Cur is parentNode's left child) {
        set parentNode's LeftNode to Cur's LeftNode
        delete current node
    }
    else {
        set parentNode's rightChild to Cur's leftChild
        delete current node
    }
}

```

```

else if (target has right child) {
    copy data to temp node
    declare right sub tree's root node tCur

    copy cur's data to temp node

    if (tCur has two child) {
        while (tCur has left child) {
            Update tCur's parent & tCur -> left leaf node
        }

        copy tCur's data to Cur

        if (tCur is leaf Node) {
            set tparentNode's rightChild to NULL
            delete tCur
        }
        else {
            set tparentNode's leftChild to tCur's RightChild
            delete tCur
        }
    }

    else if (Cur is root) {
        set root to Cur's right child
        delete Cur
    }
    else if (Cur is parentNode's left child) {
        set parentNode's left child to Cur's right child
        delete Cur
    }
    else {
        set parentNode's right child to Cur's left child
        delete Cur
    }
}

else {
    declare left sub tree's root node tCur
    copy tCur's data to temp node

```

```

        while (tCur has right child) {
            Updata tCur's parent & tCur -> right leaf node
        }

        copy tCur's data to Cur

        if (tCur is leaf Node) {
            if (tCur is parent's left child)
                set tparentNode's left child to NULL
            else
                set tparentNode's right child to NULL
            delete tCur
        }
        else {
            if (tCur is parent's right child)
                set tparentNode's right child to tCur's left child
            else
                set tparentNode's left child to tCur's left child
            delete tCur
        }
    }
    return temp node
}

WordNode* Search(char * word) {
    currentNode = root
    if BST is empty, return NULL;

    change capital letter(word to search) into small letter.

    while(currentNode) {
        initialize array.
        change capital letter(current node) into small letter.

        if current node word > word to search, move to left child.
        else if current node word < word to search, move to right child.
        else return currentNode;
    }
    return NULL; // search fail
}

```

```
bool WordBST::Print(char* order) {
    if(BST exist) {
        open file("log.txt")
        if(order == "R_PRE") {
            cond = execute R_PRE function
            close file
            return cond
        }
        else if(order == "I_PRE") {
            cond = execute I_PRE function
            close file
            return cond
        }
        else if(order == "R_IN") {
            cond = execute R_IN function
            close file
            return cond
        }
        else if(order == "I_IN") {
            cond = execute I_IN function
            close file
            return cond
        }
        else if(order == "R_POST") {
            cond = execute R_POST function
            close file
            return cond
        }
        else if(order == "I_POST") {
            cond = execute I_POST function
            close file
            return cond
        }
        else if(order == "I_LEVEL") {
            cond = execute I_LEVEL function
            close file
            return cond
        }
        else
            return false
    }
```

```

    }
    else
        return false
}
bool WordBST::R_IN() {
    if (root != NULL) {
        open file("log.txt")
        execute R_IN function
        close file
        return true
    }
    else
        return false
}

void WordBST::R_IN(WordNode * root, ofstream &fout) {
    WordNode* CurrentNode = root

    if (CurrentNode) {
        recursive inorder
        print Cur's data on window & file
    }
}

bool WordBST::R_PRE() {
    if (root != NULL) {
        open file("log.txt")
        execute R_PRE function
        close file
        return true
    }
    else
        return false
}

void WordBST::R_PRE(WordNode * root, ofstream &fout) {
    WordNode* CurrentNode = root

    if (CurrentNode) {

```

```

        recursive preorder
        print Cur's data on window & file
    }
}

bool WordBST::R_POST() {
    if (root != NULL) {
        open file("log.txt")
        execute R_POST function
        close file
        return true
    }
    else
        return false
}

void WordBST::R_POST(WordNode * root, ofstream &fout) {
    WordNode* CurrentNode = root

    if (CurrentNode) {
        recursive postorder
        print Cur's data on window & file
    }
}

bool WordBST::I_IN() {
    if (root == NULL)
        return false
    else
    {
        Declare Array of WordNode*
        Declare int type variable cnt , top and initialize to 0

        WordNode* CurrentNode = root

        open file("log.txt")
        while (1) {
            while (CurrentNode) {

```

```

        arr[cnt] = CurrentNode
        CurrentNode = left leaf node
        increase cnt
        increase top
    }

    if (cnt == 0) {
        close file
        return true
    }
    decrease top
    initialize CurrentNode to arr[top]
    initialize arr[cnt] to NULL
    decrease cnt
    print Cur's data on window & file
    CurretNode = right left node
}

}

}

bool WordBST::I_PRE() {
    if (root == NULL)
        return false
    else {
        Declare Array of WordNode*
        Declare int type variable cnt , top and initialize to 0

        WordNode* CurrentNode = root

        open file("log.txt")
        while (1) {
            while (CurrentNode) {
                initialize arr[cnt] to CurrentNode
                print Cur's data on window & file
                CurrentNode = CurrentNode's left leaf node
                increase cnt
                increase top
            }

```

```

        if (cnt == 0) {
            close file
            return true
        }
        decrease top
        initialize CurrentNode to arr[top];
        initialize arr[cnt] to NULL
        decrease cnt
        CurrentNode = CurrentNode's right leaf node
    }
}

bool WordBST::I_POST() {

    if (root == NULL)
        return false
    else {
        Declare Array of AlphabetNode*
        Declare int type variable cnt and initialize to 0
        WordNode* CurrentNode = root
        open file("log.txt")
        while (1) {
            while (CurrentNode exist) {
                if (Cur's right child exist) {
                    arr[cnt] = Cur's right child
                    increase cnt
                }
                arr[cnt] = CurrentNode
                CurrentNode = Cur's left leaf node
                increase cnt
            }

            if (cnt == 0) {
                close file
                return true
            }
            decrease cnt
            CurrentNode = arr[cnt]
            arr[cnt] = NULL
        }
    }
}

```



```

        decrease cnt

        if (CurrentNode have right child & arr[cnt] == right
child) {

            arr[cnt] = CurrentNode
            CurrentNode = Cur's right leaf node
            increase cnt
        }
        else {
            print Cur's data on window & file
            CurrentNode = NULL
            increase cnt
        }
    }
}

bool WordBST::I_LEVEL() {
    Declare Array of WordNode*
    Declare int type variable cnt, frt and initialize to 0
    WordNode* CurrentNode = root
    if (root) {
        open file("log.txt")
        while (CurrentNode) {
            print Cur's data on window & file
            if (Cur's left child exist) {
                arr[cnt] = Cur's left leaf node
                increase cnt
            }
            if (Cur's right child exist) {
                arr[cnt] = Cur's right leaf node
                increase cnt
            }
        }
        if (frt == cnt) {
            close file
            return true
        }
        CurrentNode = arr[frt]
        arr[frt] == NULL
    }
}

```

```

        increase frt
    }
}
return false
}
bool WordBST::Save() {
    open file("memorizing_word.txt")
    if (root) {
        return execute preSave(root, ingfout)
    }
    else {
        close file
        return false
    }
}

bool WordBST::preSave(WordNode * CurrentNode, ofstream & ingfout) {
    WordNode* temp = CurrentNode
    if (temp) {
        recursive preorder
        print Cur's data on file
    }
    return true
}

```

CircularLinkedList.cc

```

void CircularLinkedList::Insert(WordNode * node) {
    // temp : current node
    temp = pHead
    if CLL is empty {
        node connect to oneself.
        set the node to pHead.
        return:
    }
    move to previous node of pHead.
    set new node to next node of temp.
    set pHead to next node of new node.
}

WordNode * CircularLinkedList::Search(char * word) {
    // temp : current node
    temp = pHead
    if CLL is empty, return NULL:
}

```



```

change capital letter(word to search) into small letter.
while(1) {
    initialize array.
    change capital letter(current node) into small letter.

    if current node word == word to search, return temp;
    else move to next node.

    if temp == pHead, return NULL; // search fail
}
}

bool CircularLinkedList::Print() {
    if CLL is not empty {
        open file("log.txt")
        temp = pHead

        print temp's Word, Mean on window and file.
        temp = next Node


        for( ; temp != pHead ; temp = temp->GetNext()) {
            print temp's Word, Mean on window and file.
        }
        file close.
        return true;
    }
    else return false;
}

bool CircularLinkedList::Save() {
    open file("memorized_word.txt")

    if (pHead) {
        WordNode* Cur = pHead;
        print Cur's data on file

        for (; Cur != pHead; Cur = next node)
            print Cur's data on file
        close file
        return true
    }
}

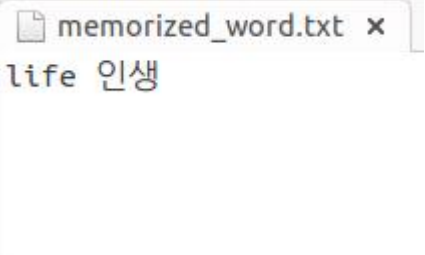
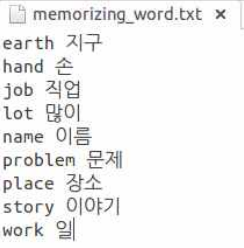
```



```
    else {  
        close file  
        return false  
    }  
}
```




▶ Result Screen

[모든 명령어에 대해 결과화면을 캡처하고 동작을 설명]

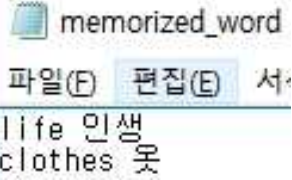
Command 1	
1	
<pre> LOAD ADD SEARCH problem MOVE 10 MOVE 91 SEARCH earth TEST life 인생 PRINT TO_MEMORIZED SAVE EXIT </pre>	첫 번째 상황의 "command.txt"파일입니다.
2	
<pre> == == == ERROR == == == 100 == == == ADD == == == Success == == == SEARCH == == == problem 문제 == == == MOVE == == == Success == == == ERROR == == == 300 == == == SEARCH == == == earth 지구 == == == TEST == == == PASS == == == PRINT == == == make 만나다 different 다른 important 중요한 ...(중간생략) captain 선장 bucket 양동이 cage 새장 kite 연 miracle 기적 == == == SAVE == == == Success </pre>	위의 커맨드를 입력하였을 때의 결과화면입니다. LOAD는 SAVE된 파일이 없기에 에러를 출력하게 됩니다. SEARCH의 경우는 problem이라는 단어가 자료구조에 존재하므로 성공적으로 나오며 MOVE의 경우는 처음엔 10개를 입력하여 성공이지만, 다음에오는 91을 더하면 101이 되어 두 번째는 에러를 출력합니다. 그리고 TEST 또한 MOVE로 MEMORIZING에 알맞은 단어가 이동하였으므로 MEMORIZED로 이동이 가능합니다.
3	
	SAVE를 한 뒤 memorized_word.txt의 파일 내용입니다.
4	
	SAVE를 한 뒤 memorizing_word.txt의 파일 내용입니다.

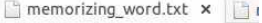
5	
<pre> log.txt x command.txt x memorized_ == == == ERROR == == == 100 == == == ADD == == == Success == == == SEARCH == == == problem 문제 == == == MOVE == == == Success == == == ERROR == == == 300 == == == SEARCH == == == earth 지구 == == == TEST == == == PASS == == == PRINT == == == make 만들다 different 다른 important 중요한 person 사람 clothes 옷 ...(중간생략) spirit 정신 route 길 pole 막대기 rubber 고무 root 뿌리 structure 구조 vote 투표 patient 환자 quick 빠른 captain 선장 bucket 양동이 cage 새장 kite 연 miracle 기적 == == == SAVE == == == Success == == == </pre>	<p>위의 커맨드를 입력하였을 때의 "log.txt"의 상태입니다. 위의 결과화면과 동일한 결과가 나올 수 있습니다.</p>

Command 2	
1	
<pre> command.txt x LOAD MOVE 15 UPDATE area 구역 TEST area 구역 PRINT MEMORIZED SAVE EXIT </pre>	<p>두 번째 상황의 "command.txt"파일입니다.</p>
2	
<pre> == == == LOAD == == == Success == == == == == == MOVE == == == Success == == == == == == UPDATE == == == area 지역->구역 == == == == == == ERROR == == == 500 == == == == == == PRINT == == == life 인생 == == == == == == SAVE == == == Success == == == </pre>	<p>위의 커맨드를 입력하였을 때의 결과화면입니다. LOAD는 첫 번째 상황에서 SAVE했던 각 자료구조의 파일을 읽어와서 그대로 구현할 수 있으므로 성공이 출력됩니다. 그 다음 15개의 단어를 MOVE로 이동하는데 성공한 뒤 자료구조 상에 있는 area라는 단어가 존재하여 구역이라는 뜻으로 뜻을 바꾸는데 성공합니다. 하지만 TEST는 MEMORIZING의 단어를 옮기는 명령어이기 때문에 area라는 단어를 TEST할 순 없습니다.</p>

3	<div data-bbox="309 309 743 353">  memorized_word.txt x </div> <div data-bbox="300 360 475 405">life 인생</div>	<p>SAVE를 한 뒤 memorized_word.txt의 파일 내용입니다.</p>
4	<div data-bbox="341 618 695 663">  memorizing_word.txt x </div> <div data-bbox="336 667 592 1167"> activity 활동 clothes 옷 different 다른 earth 지구 example 예 fire 불 fun 재미 hand 손 important 중요한 job 직업 lot 많이 letter 편지 listen 듣다 make 만들다 </div>	<p>SAVE를 한 뒤 memorizing_word.txt의 파일 내용입니다.</p>
5	<div data-bbox="261 1258 762 1303">  to_memorize_word.txt x </div> <div data-bbox="245 1310 539 1906"> learn 배우다 each 각각 same 같은 bird 새 trip 여행 vacation 휴가 summer 여름 course 강좌 spring 봄 autumn 가을 winter 겨울 space 공간 </div> <div data-bbox="245 1944 376 1973">...(중간생략)</div>	<p>SAVE를 한 뒤 to_memorize_word.txt의 파일 내용입니다.</p>

<pre> == == == == LOAD == == == == Success == == == == == == == == == == == == == == MOVE == == == == Success == == == == == == == == == == == == == == UPDATE == == == == area 지역->구역 == == == == == == == == == == == == == == ERROR == == == == 500 == == == == == == == == == == == == == == PRINT == == == == life 인생 == == == == == == == == == == == == == == SAVE == == == == Success == == == == == == == == == == </pre>	<p>위의 커맨드를 입력하였을 때의 "log.txt"의 상태입니다. 위의 결과화면과 동일한 결과가 나옴을 알 수 있습니다.</p>
---	--

Command 3		
1		
<pre>LOAD MOVE 15 TEST clothes 옷 PRINT MEMORIZING R_IN PRINT MEMORIZING I_IN SAVE EXIT </pre>		세 번째 상황의 "command.txt"파일입니다.
2		
<pre>== == == == LOAD == == == == Success == == == == MOVE == == == == Success == == == == TEST == == == == PASS == == == == PRINT == == == == activity 활동 autumn 가을 bird 새 course 강좌 different 다른 each 각각 earth 지구 example 예 fire 불 fun 재미 hand 손 important 중요한 job 직업 learn 배우다 ...(중간생략) listen 듣기 lot 많이 make 만들다 movie 영화 name 이름 newspaper 신문 paper 종이 part 부분 person 사람 place 장소 plan 계획 plant 식물 problem 문제 same 같은 space 공간 spring 봄 story 이야기 street 거리 summer 여름 trip 여행 vacation 휴가 winter 겨울 work 일 == == == == SAVE == == == == Success == == == ==</pre>		위의 커맨드를 입력하였을 때의 결과화면입니다. LOAD는 두 번째 상황에서 SAVE했던 각 자료구조의 파일을 읽어와서 그대로 구현할 수 있으므로 성공이 출력됩니다. 그 다음 15개의 단어를 MOVE로 이동하는데 성공한 뒤 MEMORIZING에 clothes라는 단어가 존재하고 명령어 상의 뜻과도 일치하기에 MEMORIZED의 자료구조로 이동할 수 있습니다. 그리고 두 가지 인오더 방법으로 MEMORIZING의 단어를 출력합니다.
3		
		SAVE를 한 뒤 memorized_word.txt의 파일 내용입니다.

<div>4</div> <div>  <pre> activity 활동 autumn 가을 bird 새 course 강좌 different 다른 earth 지구 each 각각 example 예 fire 불 fun 재미 hand 손 important 중요한 job 직업 lot 많이 letter 편지 learn 배우다 listen 듣다 make 만들다 movie 영화 name 이름 </pre> </div>	<div>SAVE를 한 뒤 memorizing_word.txt의 파일 내용입니다.</div>
<div>5</div> <div> <pre> == == == LOAD == == == Success == == == MOVE == == == Success == == == TEST == == == PASS == == == PRINT == == == activity 활동 autumn 가을 bird 새 course 강좌 different 다른 each 각각 earth 지구 example 예 fire 불 fun 재미 hand 손 important 중요한 job 직업 learn 배우다 letter 편지 listen 듣다 lot 많이 ...(중간생략) paper 종이 part 부분 person 사람 place 장소 plan 계획 plant 식물 problem 문제 same 같은 space 공간 spring 봄 story 이야기 street 거리 summer 여름 trip 여행 vacation 휴가 winter 겨울 work 일 == == == SAVE == == == Success </pre> </div>	<div>위의 커맨드를 입력하였을 때의 "log.txt"의 상태입니다. 위의 결과화면과 동일한 결과가 나옴을 알 수 있습니다.</div>

Command 4		
1		
<pre> ADD MOVE 30 TEST job 직업 TEST work 일 TEST place 장소 TEST movie 영화 PRINT MEMORIZED SAVE EXIT </pre>		네 번째 상황의 "command.txt"파일입니다.
2		
<pre> == == == == ERROR == == == == 100 == == == == == == == == ADD == == == == Success == == == == == == == == MOVE == == == == Success == == == == == == == == TEST == == == == PASS == == == == == == == == TEST == == == == PASS == == == == == == == == TEST == == == == PASS == == == == == == == == TEST == == == == PASS == == == == == == == == PRINT == == == == job 직업 work 일 place 장소 movie 영화 == == == == == == == == SAVE == == == == Success == == == == </pre>		위의 커맨드를 입력하였을 때의 결과화면입니다. LOAD는 SAVE된 파일이 없기에 에러를 출력하게 됩니다. ADD는 파일이 생성되어있고 파일안에 단어도 있으므로 성공적으로 Queue에 이동합니다. 또한 Queue에 단어도 있고 BST안의 단어도 100개 이하이므로 30개의 단어가 BST로 이동이 가능합니다. 그 후 TEST로 BST상의 단어들을 CLL로 이동시킨 뒤 CLL의 단어들을 출력해줍니다.
3		
<pre> memorized_word.txt x job 직업 work 일 place 장소 movie 영화 </pre>		SAVE를 한 뒤 memorized_word.txt의 파일 내용입니다.

4	<pre> memorizing_word.txt x activity 활동 autumn 가을 bird 새 clothes 옷 clean 깨끗한 course 강좌 different 다른 earth 지구 each 각각 example 예 enjoy 즐기다 fire 불 face 얼굴 fun 재미 hand 손 important 중요한 life 인생 letter 편지 learn 배우다 lot 많이 listen 듣다 </pre>	<p>SAVE를 한 뒤 memorizing_word.txt의 파일 내용입니다.</p>
5	<pre> to_memorize_word.txt x famous 유명한 special 특별한 just 단지 nature 자연 restaurant 식당 group 집단 habit 습관 culture 문화 information 정보 advertisement 광고 science 과학 gene 유전자 war 전쟁 store 가게 sound 소리 fly 날다 easy 쉬운 poor 가난한 fast 빨리 back 뒤 always 언제나 history 역사 state 상태 soldier 군인 </pre> <p>...(중간생략)</p>	<p>SAVE를 한 뒤 to_memorize_word.txt의 파일 내용입니다.</p>
6	<pre> == == == == ERROR == == == == 100 == == == == == == == == == == == == == == == ADD == == == == Success == == == == == == == == == == == == == == == MOVE == == == == Success == == == == == == == == == == == == == == == TEST == == == == PASS == == == == == == == == == == == == == == == TEST == == == == PASS == == == == == == == == == == == == == == == TEST == == == == PASS == == == == == == == == == == == == == == == PRINT == == == == job 직업 work 일 place 장소 movie 영화 == == == == == == == == == == == == == == == SAVE == == == == Success == == == == == == == == == == == </pre>	<p>위의 커맨드를 입력하였을 때의 "log.txt"의 상태입니다. 위의 결과화면과 동일한 결과가 나옴을 알 수 있습니다.</p>

Command 5	
1	
<div> <div>log.txt x</div> <div>command.txt x</div> </div> <pre> ADD MOVE 25 PRINT MEMORIZING I_LEVEL EXIT </pre>	다섯 번째 상황의 "command.txt"파일입니다.
2	
<pre> == == == == ADD == == == == Success == == == == == == == == == == == == == == == == == == == MOVE == == == == Success == == == == == == == == == == == == == == == == == == == PRINT == == == == problem 문제 place 장소 person 사람 plan 계획 part 부분 plant 식물 hand 손 different 다른 life 인생 letter 편지 lot 많이 listen 듣다 fire 불 fun 재미 job 직업 name 이름 activity 활동 clothes 옷 earth 지구 example 예 important 중요한 make 만들다 movie 영화 story 이야기 work 일 == == == == == == == == == == == == == == == </pre>	위의 커맨드를 입력하였을 때의 결과화면입니다. ADD는 파일이 생성되어있고 파일안에 단어도 있으므로 성공적으로 Queue에 이동합니다. 또한 Queue에 단어도 있고 BST안의 단어도 100개 이하이므로 25개의 단어가 BST로 이동이 가능합니다. 그 후 BST의 단어들을 레벨오더로 출력해줍니다.

3

```
log.txt x command.txt x
== == == ADD == == ==
Success
== == == == == == == == ==
== == == MOVE == == ==
Success
== == == == == == == == ==
== == == PRINT == == ==
problem 문제
place 장소
person 사람
plan 계획
part 부분
plant 식물
hand 손
different 다른
life 인생
letter 편지
lot 많이
listen 듣다
fire 불
fun 재미
job 직업
name 이름
activity 활동
clothes 옷
earth 지구
example 예
```

위의 커맨드를 입력하였을 때의 "log.txt"의 상태
입니다. 위의 결과화면과 동일한 결과가 나옴을
알 수 있습니다.

► Consideration

[팀원별로 고찰 작성]

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자신의 점수 (10)
김시훈	Manager.cc에서 TEST, SAVE, SEARCH, EXIT WordBST.cc에서 Search, R_POST, I_POST(AlphabetBST.cc에서도) AlphabetBST.cc의 Get/SetWordCnt 각 자료구조 클래스의 Save, Search	<u>10</u>
고찰	<p>맨 처음에 세 가지의 자료구조를 가지고 프로젝트를 한다고 들었을 때, 저는 실로 난감하지 않을 수 없었습니다. 하지만 저를 믿어주는 팀원과 1학기 때 배운 C++능력으로 이번 프로젝트를 잘 헤쳐나간 것 같습니다. 특히 리눅스라는 새로운 OS를 이용해서 해야 하는 특별함 때문에 많이 어려울 것 같았으나 팀원들이 많이 도와줘서 재미있었습니다. 앞으로는 지금배운 것을 토대로 새로운 개념들에 익숙해지고 연습해서 더 나은 실력을 가지도록 해야겠습니다.</p>	

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자신의 점수 (10)
고정원	Manager.cc에서 UPDATE, LOAD, PRINT WordBST.cc에서 Delete, R_PRE, I_PRE(AlphabetBST.cc에서도) AlphabetNode.cc 전부 각 자료구조 클래스의 Insert(or Push)	<u>10</u>
고찰	<p>프로그램을 팀으로 만들어 보는 것은 이번이 처음이었는데 혼자서 만드는 것보다 좀 더 퀄리티 높은 프로그램이 만들어 지는 것 같았다. 서로 함수가 만들어 질 때마다 피드백을 해서 잘못된 부분들을 고쳐주어서 생각하지 못한 예외처리들도 쉽게 잡을 수 있었고, 프로그램을 다 만든 후 예외처리 하는 부분에서도 여러 명이 하니 오류들을 쉽게 고칠 수 있었다. 특히 마지막에 갑자기 segmentation fault 가 나서 에러를 잡기 힘들었는데, 소멸자 부분의 예외처리를 생각해 주는 것을 바로 떠올리기 어려웠는데, 이 부분을 해결하는 것이 이번 프로젝트에서 가장 힘든 일 이었다.</p>	

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자신의 점수 (10)
신지영	Manager.cc에서 run, ADD, MOVE WordBST에서 Insert, Print, R_IN, I_IN, I_LEVEL(AlphabetBST.cc에서도) WordNode.cc 전부 각 자료구조 클래스의 Print Queue.cc에서 Pop	<u>10</u>
고찰	<p>이번 과제는 bst를 중심으로 프로그램을 구현하는 프로젝트였는데, 팀원들과 적절하게 함수 구현을 배분하여 나름 빠르게 과제를 수행할 수 있었다.</p> <p>그리고 결과화면은 잘 나왔었는데 마지막에 segmentation fault가 떠서 확인해보니 소멸자의 문제였다.</p> <p>전공 수업 중에서는 팀 프로젝트가 처음이라 어색했는데 결속력도 좋았고, 팀워크가 좋아서 에러 문제가 발생하여도 금방 해결할 수 있었다.</p>	