

Data Structure Lab. Project #1 REPORT

프로젝트제목: 단어장 프로그램

제출일자: 2016년 10월 07일 (금)

학 과: 컴퓨터공학과

담당교수: 이기훈교수님

학 번: 2012722016

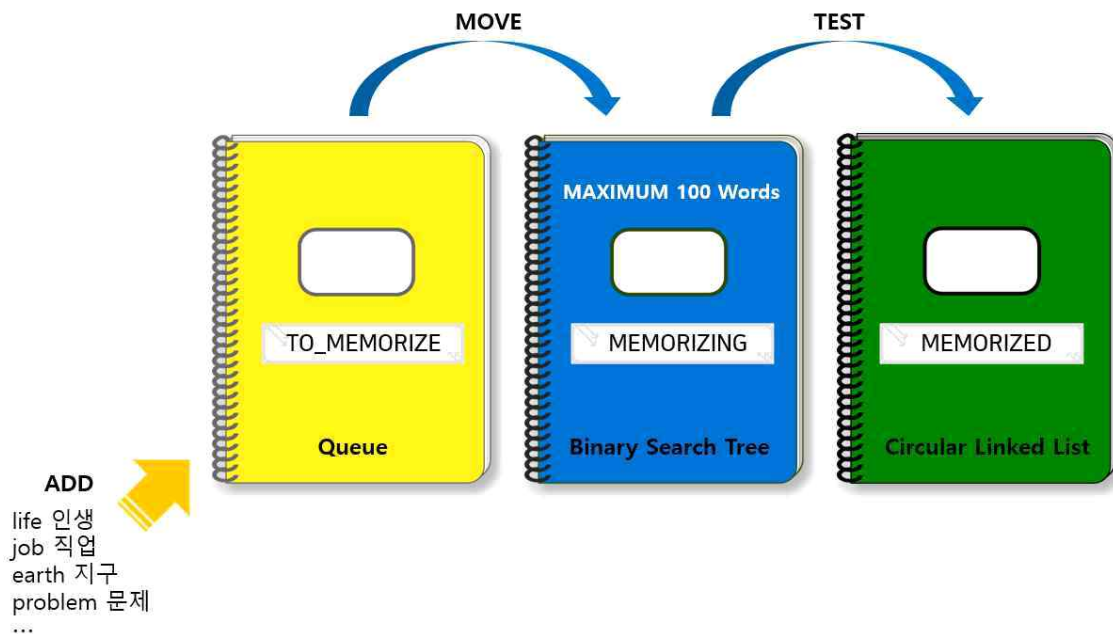
2007

성 명: 이명진

이종찬

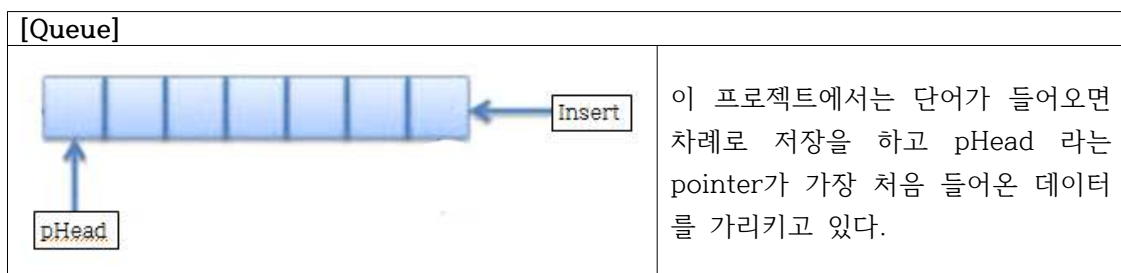
[Introduction]

이 프로젝트는 단어장 프로그램이다. 영단어와 이에대한 뜻이 저장되어 있는 word 가 텍스트 파일에 저장되어 있다.



- [To_MEMORIZE]

프로그램이 시작되면 ADD란 명령어를 통해 word 텍스트파일에 있는 모든 단어들이 To_MEMORIZE 파일로 입력이 된다. 이 경우 To_MEMORIZE의 단어들은 Queue 형태로 저장이 된다. Queue는 간략히 설명하면 아래의 그림과 같이 FIFO(First In First Out) 형태로 먼저 들어온 데이터가 가장 먼저 나가는 방식의 구조이다.

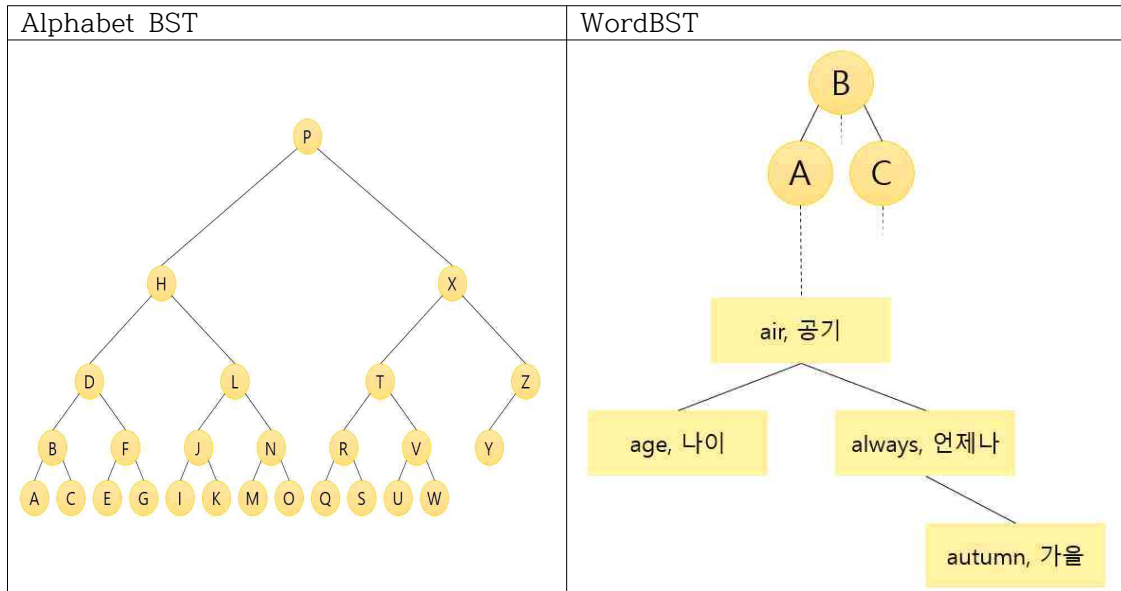


이 Queue 안에는 데이터가 pair함수를 사용하여 각각의 노드에 영어단어와 한글 뜻이 쌍으로 저장이 되어있다. 새로운 단어가 들어오면 Push를 하여 저장하고 MEMORIZING으로 단어를 옮길 때에는 Pop을 하여 단어를 뺀다.

- [MEMORIZING]

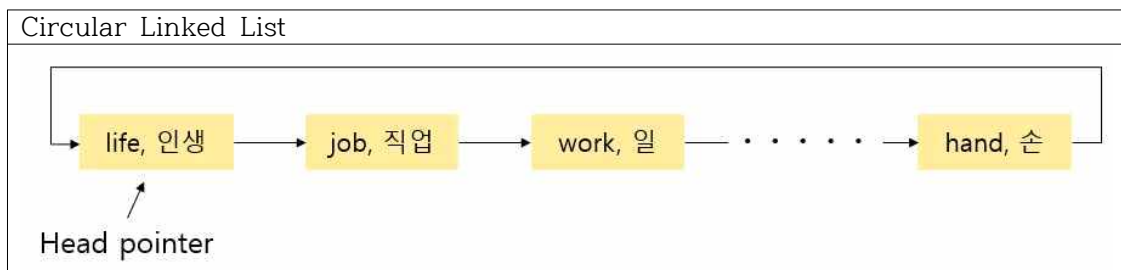
말 그대로 단어를 외우는 부분이라고 볼 수 있다. TO_MEMORIZE에서 외울

단어들을 MOVE라는 명령어를 통해 옮긴다. 이때 MEMORIZING에는 단어의 개수가 최대 100개가 존재한다. Queue에서 Pop하여 옮긴 단어들을 이 부분에서는 Binary search Tree 형태로 저장한다.



위의 그림과 같이 Tree 구조로 저장을 하기 때문에 단어들을 찾을 때(SEARCH), 단어들을 UPDATE할 때와 같은 경우 효율적으로 동작이 가능하게 된다. 프로그램이 시작되면 Alphabet BST는 생성자를 통해 미리 생성이 된다. BST를 구성할 때에는 사전적 순서가 작은 노드는 왼쪽, 큰 노드는 오른쪽 서브 트리에 위치한다. MEMORIZING에서는 단어들을 TEST 명령어를 통해 단어를 외우는 시험을 할 수 있다. TEST 한 영어단어와 한글 뜻이 일치하면 외웠다는 의미에서 MEMORIZED로 단어노드가 옮겨진다.

- [MEMORIZED]



TEST 명령어를 통해 단어를 외웠다면 MEMORIZED로 단어가 들어온다. 위의 그림과 같이 Head pointer를 통해 가장 처음 들어온 노드를 가리키고 있다. 하지만 Queue와 다른 점은 가장 마지막 노드가 가장 처음의 노드를 가리키고 있는 circular

Linked List 형태라는 점이다.

- [명령어]

1. LOAD : 기존의 단어장 정보를 불러오는 명령어, 텍스트 파일에 단어장 정보가 존재 할 경우에만 각각의 텍스트 파일을 읽어 이전과 동일한 연결순서를 갖는 자료구조형태로 저장한다. (ex : LOAD)

TO_MEMORIZE : to_memorize_word.txt

MEMORIZING : memorizing_word.txt

MEMORIZED : memorized_word.txt

2. ADD : 단어 텍스트 파일에 있는 단어 정보를 읽어오는 명령어 (ex : ADD)

단어 텍스트 파일 : word.txt

3. MOVE : 사용자가 입력한 수 만큼 TO_MEMORIZE의 단어들을 MEMORIZING으로 옮기는 명령어, 단 100개이하의 단어들이 MEMORIZING에 저장된다.

(ex : MOVE 90)

4. SAVE : 현재 자료구조에 있는 단어장의 정보를 저장하는 명령어 (ex : SAVE)

5. TEST : 단어를 외웠는지 테스트하는 명령어 (ex : TEST area 지역)

6. SEARCH : 단어의 뜻을 찾아 출력하는 명령어, 모든 자료구조에서 검색하여 존재할 경우 출력하여 준다. (ex : SEARCH area 지역)

7. PRINT : 입력한 단어장의 있는 단어들을 출력하는 명령어

- Queue : Header부터 순서대로 출력

- BST : 7개의 방법을 입력받아 출력

 - 1. Recursive pre-order

 - 2. Iterative pre-order

 - 3. Recursive in-order

 - 4. Iterative in-order

 - 5. Recursive post-order

 - 6. Iterative post-order

 - 7. Iterative level-order

- Circular Linked List : Header부터 순서대로 출력

(ex : PRINT TO_MEMORIZE , PRINT MEMORIZING I_PRE)

8. UPDATE : 단어의 뜻을 변경하는 명령어 (ex : UPDATE area 구역)

9. EXIT : 프로그램 상의 메모리를 해제하며, 프로그램을 종료하는 명령어
(ex : EXIT)

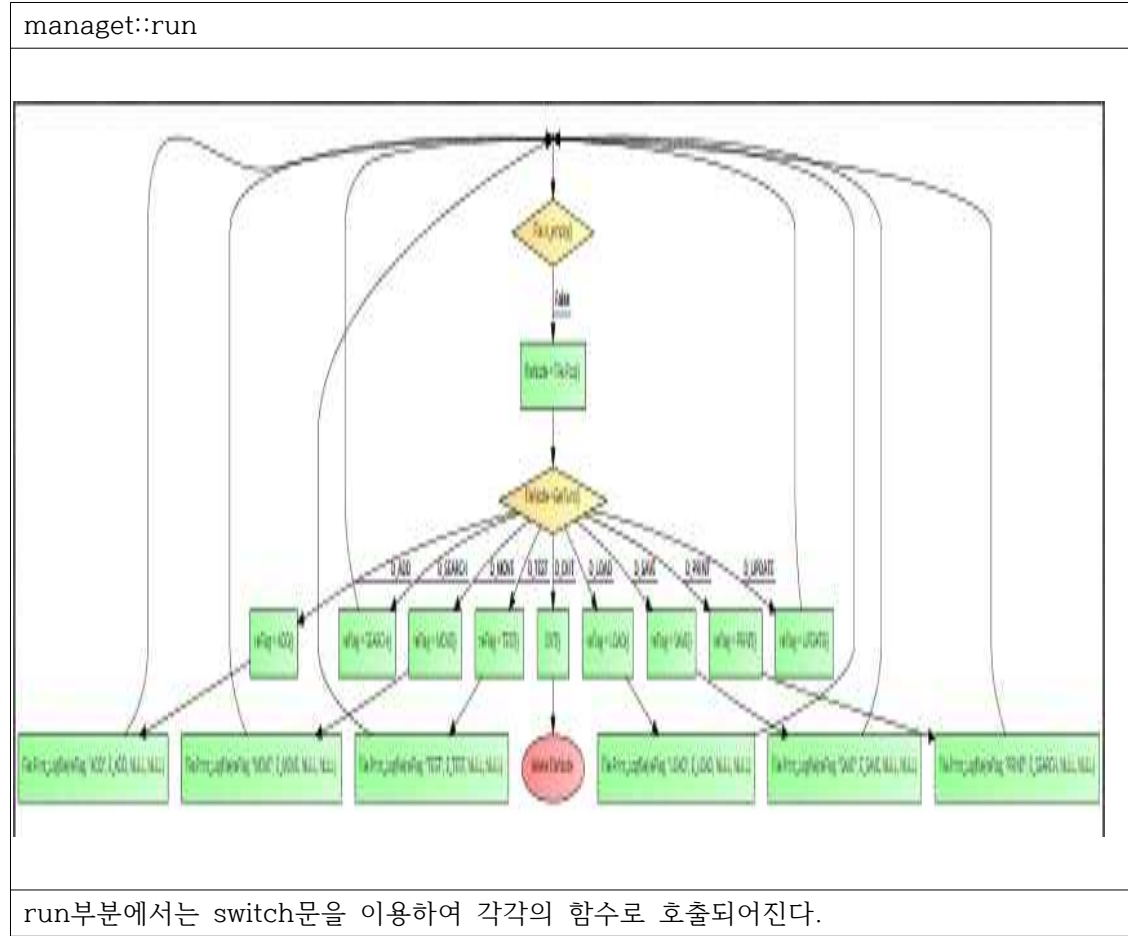
- [ERROR CODE]

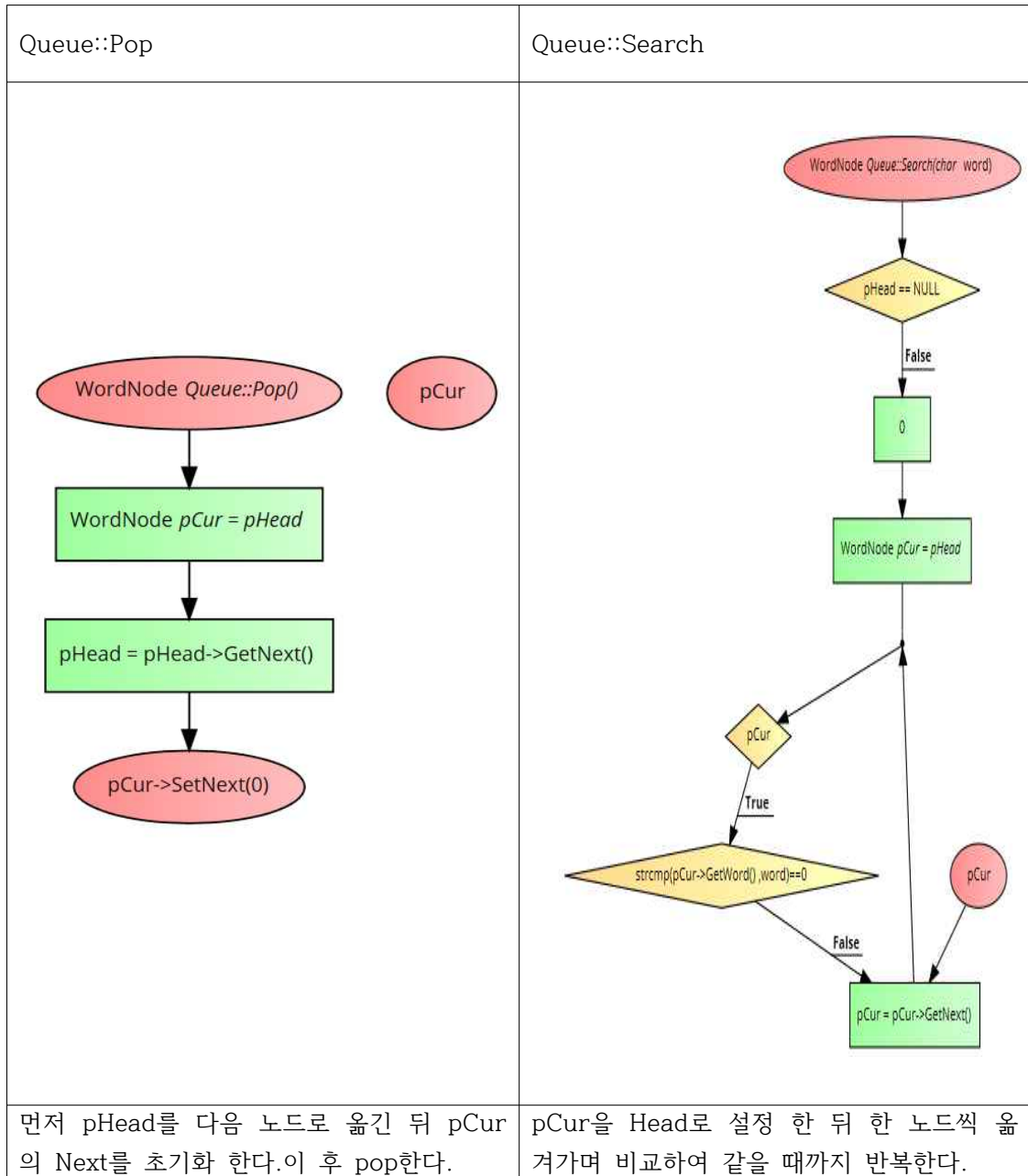
동작	에러 코드
LOAD	100
ADD	200
MOVE	300
SAVE	400
TEST	500
SEARCH	600
PRINT	700
UPDATE	800

- * 모든 명령어는 반드시 대문자로 입력하여야한다.
- * TEST, SEARCH, UPDATE 시 단어의 대소문자 구별하지 않는다.
- * 명령어에 인자가 모자라거나 필요이상의 입력받을 경우 Error가 나온다.
- * 모든 단어장에는 중복된 단어가 존재 하지 않는다.

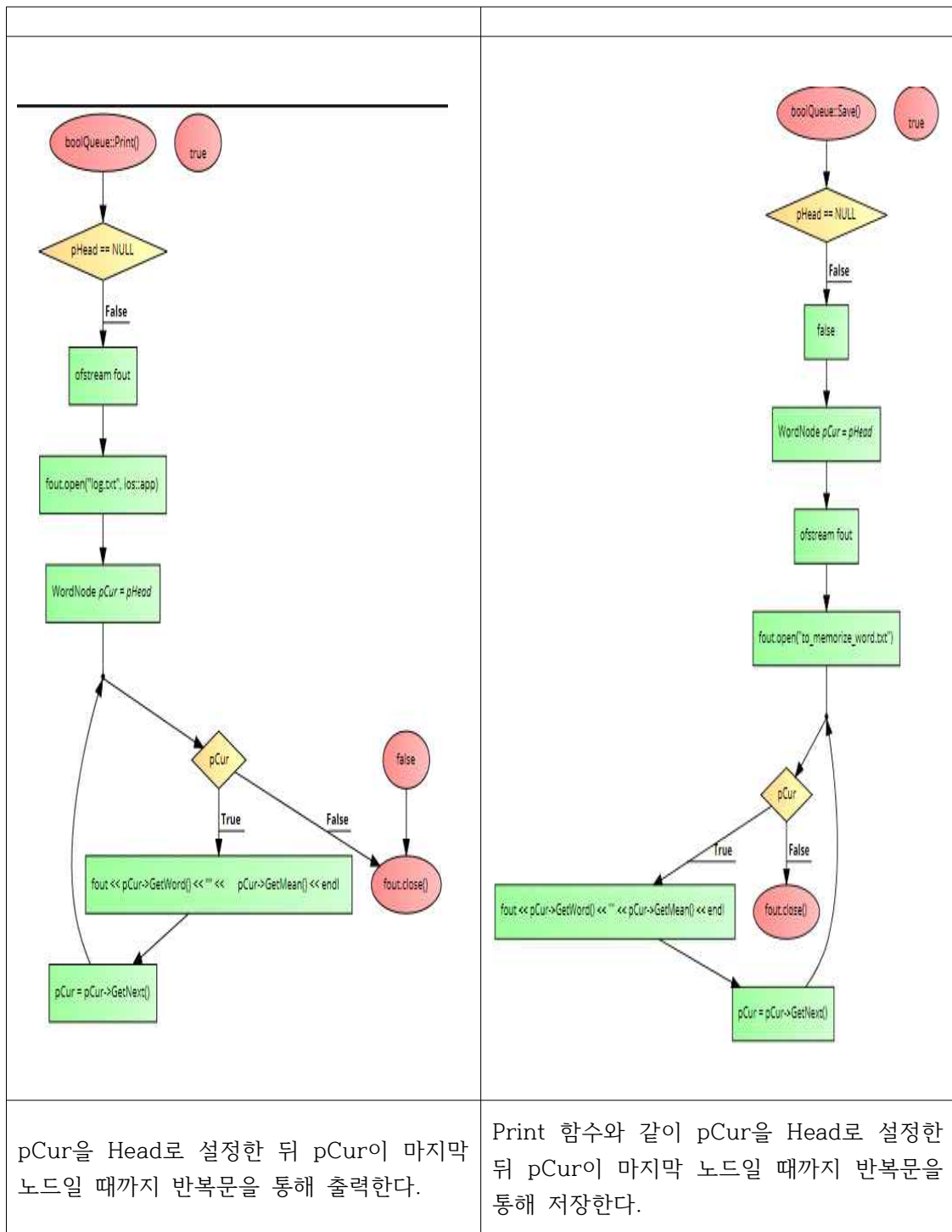
LOAD	3개의 텍스트 파일이 존재하지 않을 경우 자료구조에 이미 데이터가 들어가 있을 경우
ADD	단어 텍스트 파일이 존재하지 않을 경우
MOVE	TO_MEMORIZE에 단어가 없을 경우 입력받은 수와 MEMORIZING의 단어 수의 합이 100이 넘어갈 경우 입력한 단어 수 만큼 단어가 존재하지 않을 경우
SAVE	단어장 정보가 존재하지 않을 경우
TEST	입력한 단어가 MEMORIZING에 존재하지 않을 경우
SEARCH	입력한 단어가 존재하지 않을 경우
PRINT	입력한 단어장 정보가 존재하지 않을 경우 MEMORIZING의 경우 출력 순서를 입력하지 않았을 경우
UPDATE	단어가 존재하지 않을 경우 단어장 정보가 존재하지 않을 경우

[Flowchart]

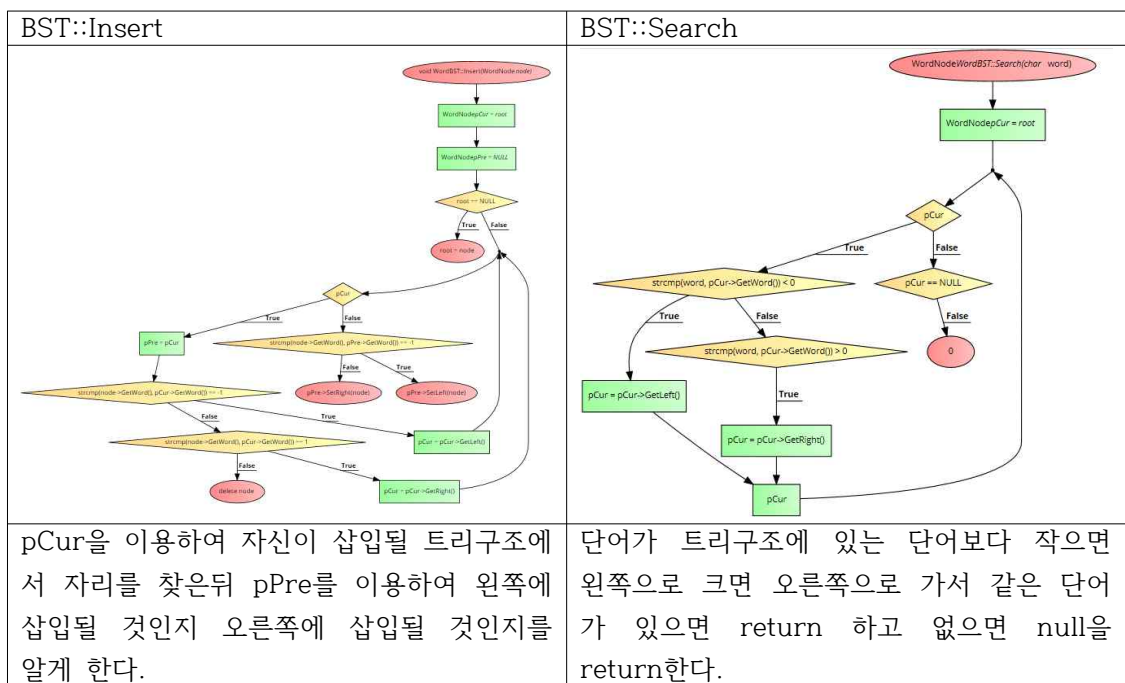
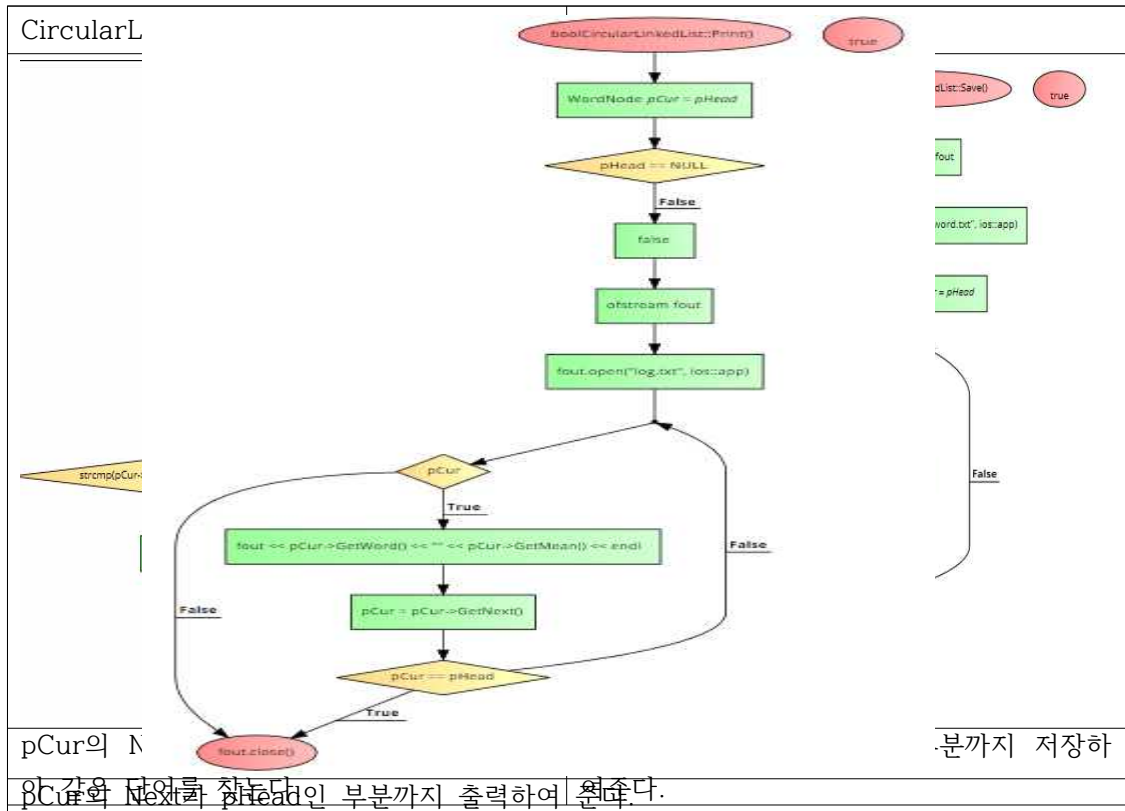




<p>Queue::~Queue</p> <pre> graph TD Start([::Queue()//delete queue]) --> Cond1{pHead == NULL} Cond1 -- False --> Assign1[WordNode pCur = pHead] Assign1 --> Assign2[WordNode delpCur = pCur] Assign2 --> Cond2{pCur->GetNext()} Cond2 -- True --> Assign3[delpCur = pCur] Assign3 --> Assign4[pCur = pCur->GetNext()] Assign4 --> Cond2 Cond2 -- False --> End1([delete pCur]) </pre> <p>pCur과 delpCur을 이용하여 Head부터 노드 한 개씩 차례로 지운다.</p>	<p>Queue::Push</p> <pre> graph TD Start([void Queue::Push(WordNode node)// LOAD, ADD]) --> Cond1{pHead == NULL} Cond1 -- True --> Assign1[pHead = node] Note1[when node insert at first] -.-> Assign1 Cond1 -- False --> Assign2[WordNode pCur = pHead] Assign2 --> Cond2{pCur->GetNext()} Note2[make a pCur pointer] -.-> Cond2 Cond2 -- True --> Assign3[pCur = pCur->GetNext()] Assign3 --> Cond2 Cond2 -- False --> End1([pCur->SetNext(node)]) Note3[pCur pointer that is last insert Node] -.-> End1 </pre> <p>반복을 통해 pCur의 Next가 없는 노드를 찾는다. 이때 Next가 존재하지 않는다면 set을 이용하여 push를 해준다.</p>
<p>Queue::Print</p>	<p>Queue::Save</p>

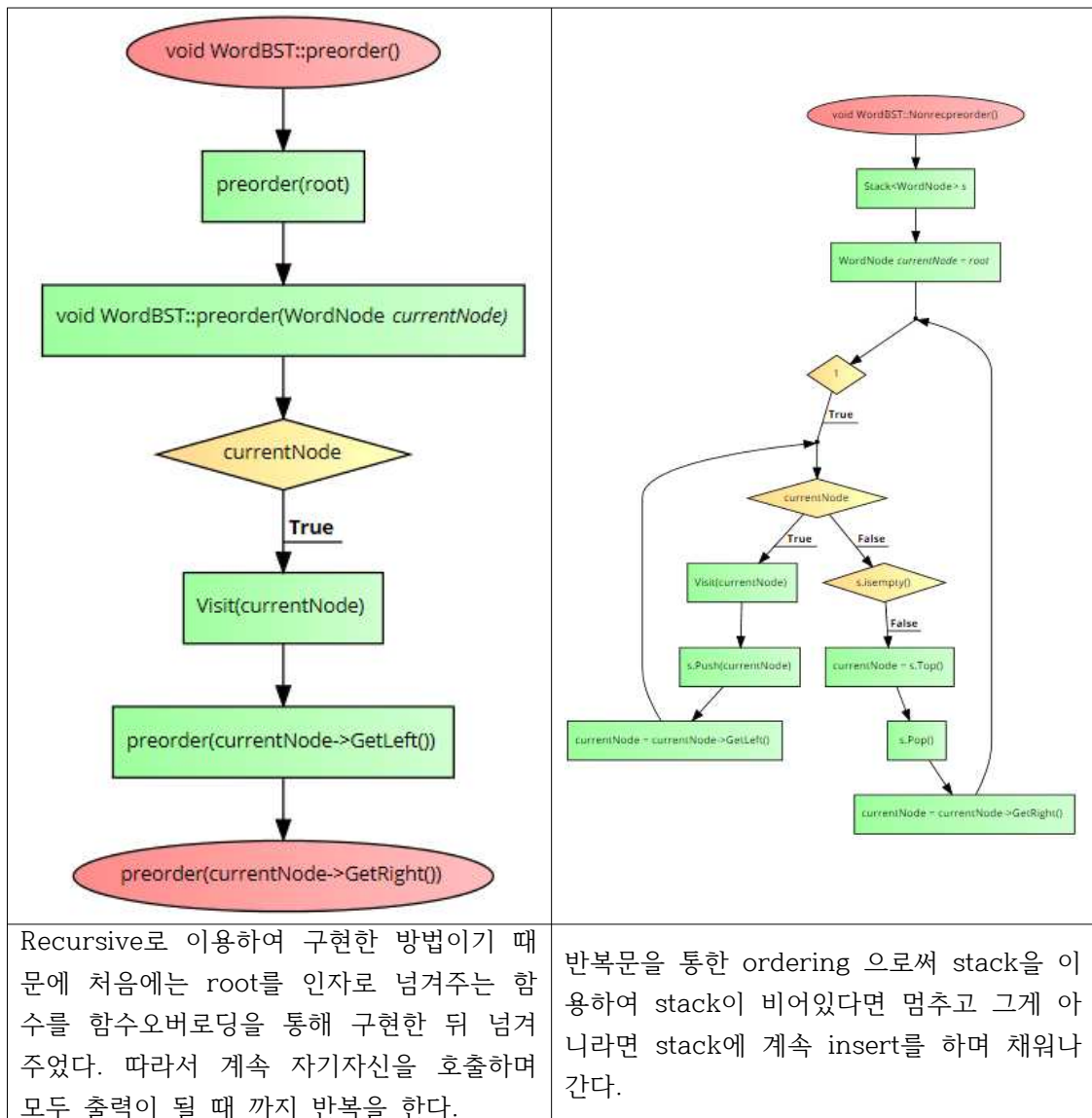


CircularLinkedList:: Print

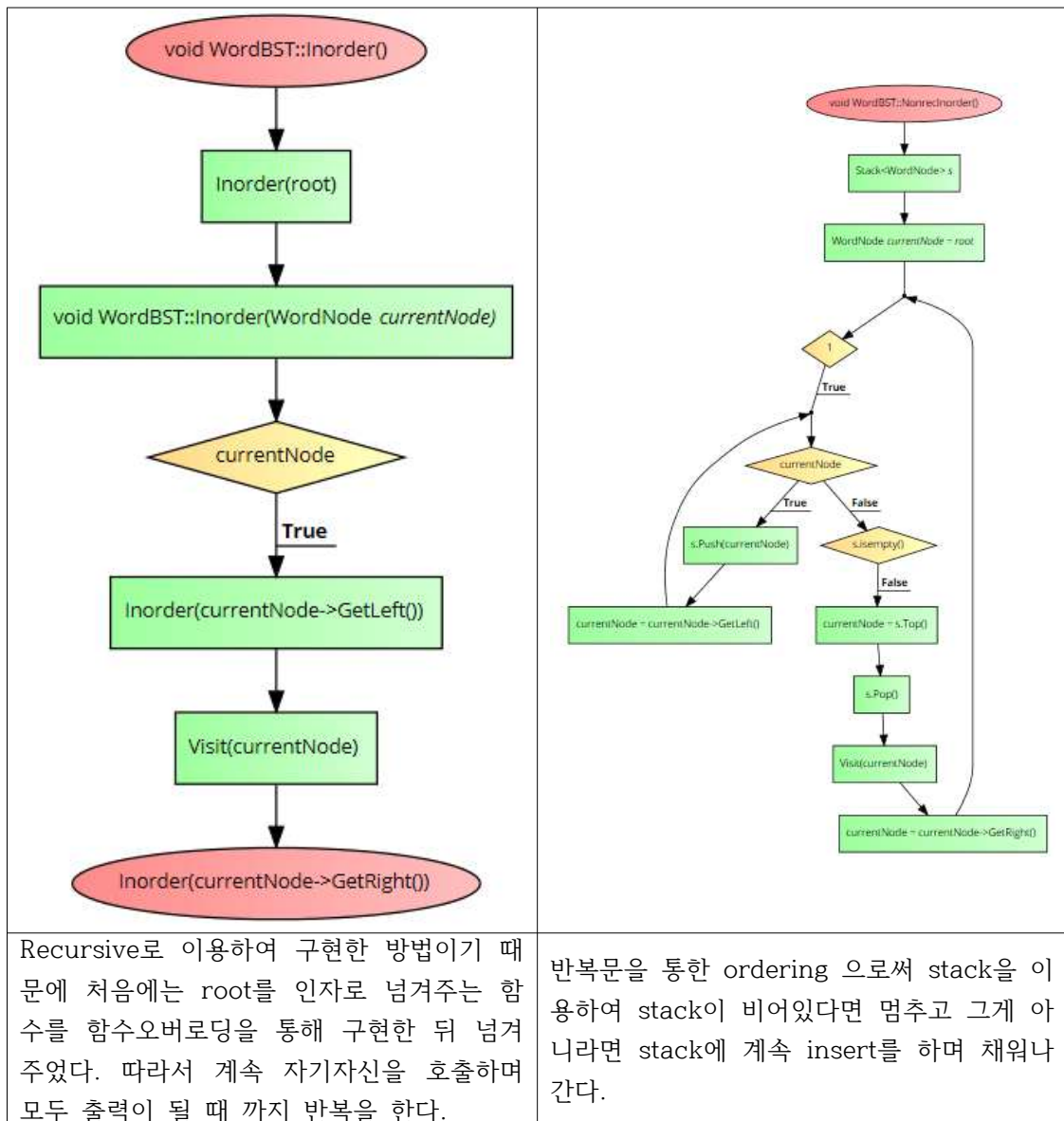


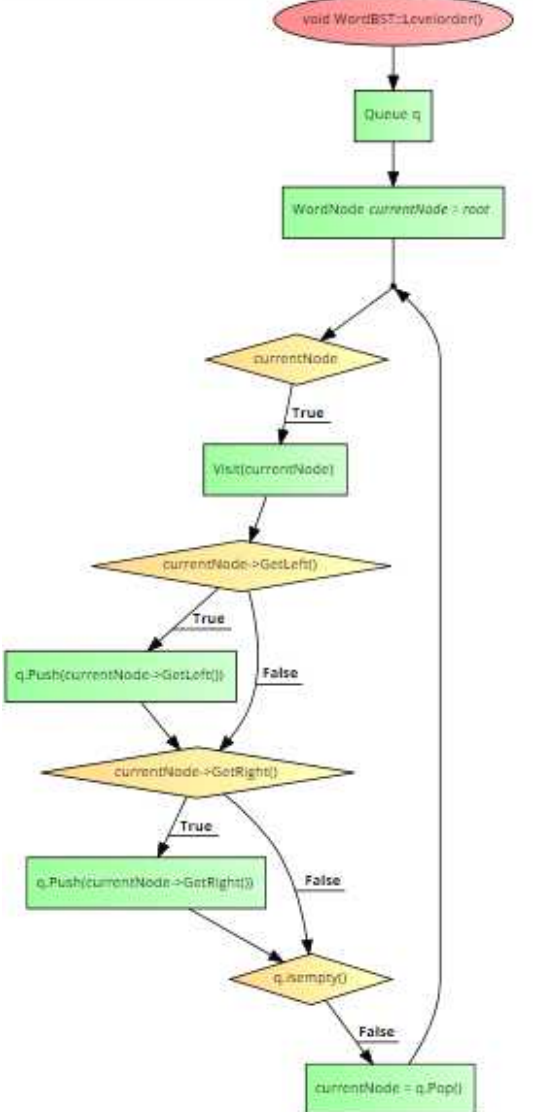
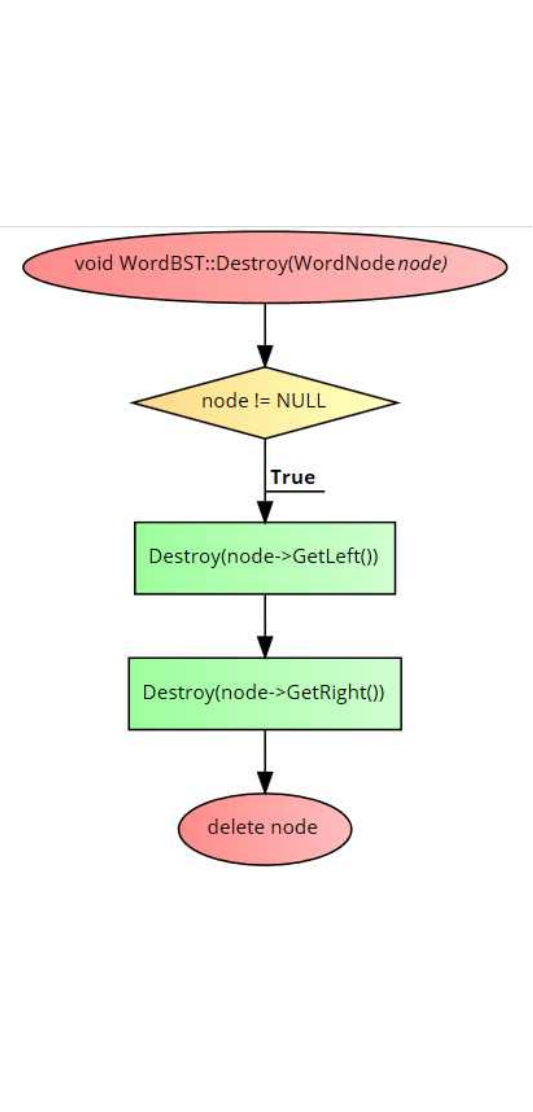
CircularLinkedList:: ~CircularLinkedList	CircularLinkedList :: Insert
<pre> graph TD Start([CircularLinkedList::~~CircularLinkedList()]) --> Init[WordNode pCur = pHead] Init --> Tail[WordNode tail] Tail --> LoopStart(()) LoopStart --> Cond1{pCur->GetNext() == pHead} Cond1 -- True --> Next[WordNode pCur = pCur->GetNext()] Next --> LoopStart Cond1 -- False --> TailSet[tail = pCur] TailSet --> LoopStart LoopStart --> Cond2{pCur != tail} Cond2 -- True --> DelHead[delete pHead] DelHead --> ReInit[WordNode pCur = pHead] ReInit --> LoopStart Cond2 -- False --> DelTail[delete tail] DelTail --> ReInit </pre>	<pre> graph TD Start([void CircularLinkedList::Insert(WordNode node)]) --> Cond1{pHead == NULL} Cond1 -- True --> SetHead[pHead = node] SetHead --> InitCur[WordNode pCur = pHead] Cond1 -- False --> InitCur InitCur --> LoopStart(()) LoopStart --> Cond2{pCur->GetNext() == pHead} Cond2 -- True --> SetNext[pCur->SetNext(node)] SetNext --> SetNextTail[node->SetNext(pHead)] SetNextTail --> LoopStart Cond2 -- False --> NextCur[pCur = pCur->GetNext()] NextCur --> LoopStart </pre>
<p>pCur의 Next가 head와 같아질 때 pCur은 마지막노드이다. 마지막노드를 찾으면 마지막노드까지 삭제시킨다.</p>	<p>반복을 통해 pCur의 Next가 head인 노드를 찾는다. 이때 pCur은 마지막 노드이므로 마지막 노드와 연결하므로써 insert한다.</p>

BST::Save



<div data-bbox="245 286 775 1294"> <pre> graph TD Start([void WordBST::postorder()]) --> Call1[postorder(root)] Call1 --> FuncDef[void WordBST::postorder(WordNode currentNode)] FuncDef --> Cond1{currentNode} Cond1 -- True --> Call2[postorder(currentNode->GetLeft())] Call2 --> Call3[postorder(currentNode->GetRight())] Call3 --> End([Visit(currentNode)]) </pre> </div> <div data-bbox="245 1305 796 1505"> <p>Recursive로 이용하여 구현한 방법이기 때문에 처음에는 root를 인자로 넘겨주는 함수를 함수오버로딩을 통해 구현한 뒤 넘겨주었다. 따라서 계속 자기자신을 호출하며 모두 출력이 될 때 까지 반복을 한다.</p> </div>	<div data-bbox="804 286 1347 1294"> <pre> graph TD Start([void WordBST::nonrecpostorder()]) --> Init[Stack<WordNode> s] Init --> Cond1{1} Cond1 -- True --> Cond2{root} Cond2 -- True --> Push1[s.Push(root)] Push1 --> Prev[WordNode prev = NULL] Prev --> Cond3{s.empty()} Cond3 -- True --> Cond4{s.empty()} Cond4 -- True --> End([Visit(currentNode)]) Cond3 -- False --> Pop1[WordNode cur = s.top()] Pop1 --> Cond5[prev == prev->GetLeft() == cur prev->GetRight() == cur] Cond5 -- True --> Cond6[cur->GetLeft() == prev] Cond6 -- True --> Cond7[cur->GetRight() == prev] Cond7 -- True --> Push2[s.Push(cur->GetLeft())] Cond7 -- False --> Push3[s.Push(cur->GetRight())] Cond6 -- False --> Visit1[Visit(cur)] Cond7 -- True --> Visit2[Visit(cur)] Visit1 --> Pop2[s.Pop()] Visit2 --> Pop2 Pop2 --> Prev[prev = cur] Prev --> Cond3 </pre> </div> <div data-bbox="804 1305 1355 1505"> <p>반복문을 통한 ordering 으로서 stack을 이용하여 stack이 비어있다면 멈추고 그게 아니라면 stack에 계속 insert를 하며 채워나간다.</p> </div>
<div data-bbox="245 1912 775 1968"> <pre> graph LR Start([BST::inorder]) </pre> </div>	<div data-bbox="804 1912 1347 1968"> <pre> graph LR Start([BST::nonrecinorder]) </pre> </div>



BST::levelorder	BST::~~BST()
 <pre> graph TD Start([void WordBST::Levelorder()]) --> Queue[Queue q] Queue --> Root[WordNode currentNode = root] Root --> IsCurrentNode[currentNode] IsCurrentNode -- True --> Visit[Visit(currentNode)] Visit --> IsLeft[currentNode->GetLeft()] IsLeft -- True --> PushLeft[q.Push(currentNode->GetLeft())] IsLeft -- False --> IsRight[currentNode->GetRight()] IsRight -- True --> PushRight[q.Push(currentNode->GetRight())] IsRight -- False --> IsQueueEmpty[q.empty()] IsQueueEmpty -- False --> Pop[currentNode = q.Pop()] Pop --> IsCurrentNode IsQueueEmpty -- True --> End([]) </pre>	 <pre> graph TD Start([void WordBST::Destroy(WordNode node)]) --> IsNodeNull{node != NULL} IsNodeNull -- True --> DestroyLeft[Destroy(node->GetLeft())] DestroyLeft --> DestroyRight[Destroy(node->GetRight())] DestroyRight --> DeleteNode([delete node]) </pre>
<p>level order는 queue를 이용하여 설계한다. queue가 비어있다면 멈추고 비어있지 않을 때 까지 반복한다. 이때 level 순이기 때문에 부모노드, 왼쪽, 오른쪽 자식 순으로 출력된다.</p>	<p>recursive 형식으로 소멸자를 구현하였다. post order 방식이므로 마지막에 부모노드를 삭제한다.</p>

[Algorithm]

<PSEUDO CODE>

QUEUE	code & explanation
<pre> <Push> if(pHead == NULL) pHead = node else pCur = pHead while(pCur->GetNext have a node) pCur = pCur->GetNext pCur->SetNext </pre>	<p>pHead가 비어 있다는 말은 처음으로 Queue에 노드가 push가 되었다는 뜻이다. 따라서 처음 들어온 노드를 pHead로 설정하여 주고 처음 들어온 노드가 아니라면 마지막 노드를 찾아 마지막노드에 들어온 노드를 연결하여 준다</p>
<pre> <Pop> if(pHead == NULL) return pCur = pHead pHead = pHead->Getnext pCur -> setNext(0) return pCur </pre>	<p>pHead가 NULL이면 Queue에 노드가 비어 있다는 뜻이므로 다시 돌아간다. 비어있지 않다면 pHead를 현재 pHead가 가리키고 있는 Next node로 설정하여 주고 pop할 노드의 next를 초기화 해준 뒤 pop할 노드를 return 한다.</p>
<pre> <Search> if(pHead == NULL) return pCur = pHead while(if pCur have word) { compare word of pCur and search word if(pCur have same word) return pCur pCur = pCur->GetNext } </pre>	<p>pHead가 NULL이면 Queue에 노드가 비어 있다는 뜻이므로 다시 돌아간다. 비어있지 않다면 비교를 시작한다. pCur에 pHead를 대입하고 pHead부터 차례로 비교하며 찾는 단어가 존재한다면 그 단어를 return 하여 준다.</p>
<pre> <Print> fileopen "log.txt" if(pHead == NULL) {fileclose and return false} else pCur = pHead while(if pCur have word) { write word and mean of pCur pCur = pCur -> GetNext } fileclose </pre>	<p>먼저 log.txt. 파일을 열어준다 하지만 pHead가 NULL이면 파일을 닫고 실패했다는 false를 return 해준다. 그게 아니라면 pCur에 pHead를 대입하여주고 pCur이 존재할 때까지 하나씩 파일에 써 준다. 그리고 마지막에 file을 close해준다.</p>
<pre> <Save> </pre>	<p>먼저 pHead가 NULL인지 검사한다. NULL</p>

<pre> if(pHead == NULL) return false pCur = pHead fileopen "to_memorize_word.txt" while(if pCur have word) { write word and mean of pCur pCur = pCur -> GetNext } fileclose </pre>	<p>이러면 데이터가 없는 것이므로 false를 return 하고 그게 아니라면 to_memorize_word 텍스트파일을 열어 print에서의 방법과 같이 하나씩 옮겨주며 저장을 하여준다. 마지막엔 파일을 닫아준다.</p>
---	---

Circular Linked List	code & explanation
<pre> <Insert> if(pHead == NULL) pHead = node else pCur = pHead while(pCur->GetNext have a node) pCur = pCur->GetNext pCur->SetNext(node) node->SetNext(pHead) </pre>	<p>pHead가 비어 있다는 말은 처음으로 cll에 노드가 Insert가 되었다는 뜻이다. 따라서 처음 들어온 노드를 pHead로 설정하여 주고 처음 들어온 노드가 아니라면 마지막 노드를 찾아 마지막노드에 들어온 노드를 연결하여 준다. 또 마지막노드가 Head를 가리키게 해 준다.</p>
<pre> <Search> if(pHead == NULL) return pCur = pHead while(if pCur have word) { compare word of pCur and search word if(pCur have same word) break else pCur = pCur->GetNext if(pCur == pHead) return 0 } return pCur </pre>	<p>pHead가 NULL이면 cll에 노드가 비어있다는 뜻이므로 다시 돌아간다. 비어있지 않으면 비교를 시작한다. pCur에 pHead를 대입하고 pHead부터 차례로 비교하며 찾는 단어가 존재한다면 멈춘다. 만약 pHead와 pCur이 같다면 이미 한번 돌아갔다는 뜻이므로 return 0를 해준다. 반복문을 나왔다면 찾는 단어가 pCur에 저장이 되었으므로 pCur을 return 한다.</p>
<pre> <Print> if(pHead == NULL) {return false} fileopen "log.txt" pCur = pHead while(if pCur have word) { write word and mean of pCur pCur = pCur -> GetNext if(pCur == pHead) break; } fileclose </pre>	<p>pHead가 NULL이면 실패했다는 false를 return 해준다. 그게 아니라면 먼저 log.txt. 파일을 열어준다. pCur에 pHead를 대입하여주고 pCur이 존재할 때까지 하나씩 파일에 써준다. pCur과 pHead가 같다면 이미 한번씩 다 출력을 해준것이므로 반복문을 빠져나온 뒤 file을 close해준다.</p>
<pre> if(pHead == NULL) return false pCur = pHead </pre>	<p>먼저 pHead가 NULL인지 검사한다. NULL이라면 데이터가 없는 것이므로 false를</p>

<pre> fileopen "memorized_word.txt" while(if pCur have word) { write word and mean of pCur pCur = pCur -> GetNext if(pCur == pHead) break } fileclose </pre>	<p>return 하고 그게 아니라면 memorized_word 텍스트파일을 열어 print에서의 방법과 같이 하나씩 옮겨주며 저장을 하여준다.역시 pCur과 pHead가 같을 땐 break를 하여 반복문을 빠져나온 뒤 파일을 닫아준다.</p>
---	--

STACK	code & explanation
<p><Top></p> <pre> if(is empty?) return return stack[top] </pre>	<p>stack에서 가장 나중에 push된 그리고 제일 먼저 pop을 해야 하는 부분을 가리키게 만드는 함수이다. stack이 비어있다면 return을 해주고 비어있지 않다면 top이란 부분은 단어가 push될 때마다 1증가시켜준 부분이므로 stack[top]은 제일 나중에 push된 부분을 가리킨다. 따라서 stack[top]을 return 한다.</p>
<p><Push></p> <pre> if (top == capacity - 1) { T* stack; stack = new T[capacity * 2] for (int i = 0; i < capacity; i++) stack1[i] = stack[i] capacity *= 2 delete[]stack stack = stack1 } stack[++top] = x; </pre>	<p>stack은 queue와 달리 alphabetBST와 WordBST부분에서 모두 쓰이므로 Template으로 구성하였다. top == capacity - 1의 부분은 stack에 크기가 짝 차게 된다는 뜻이므로 stack의 크기를 늘려줘야한다. 따라서 현재 할당된 stack의 크기의 2배만큼을 늘려준다. stack을 늘릴필요가 없거나 늘렸다면 stack[++top]에 현재들어온 데이터를 push한다. 이때 후위증가이므로 stack[top]에 데이터를 넣어주고 top을 증가시킨다.</p>
<p><Pop></p> <pre> if (isempty()) no pop stack[top--].~T(); </pre>	<p>stack이 비어있다면 pop이 불가능하다. 따라서 pop을 수행할 수 없다. 이후 stack[top--]를 삭제시킨다. 이때 top을 후위감소로 줄임으로서 stack[top]을 삭제한뒤 top을 pop한 노드 다음으로 늦게 push된 데이터를 가리키게 된다.</p>

BST	code & explanation
<pre> <~BST()> if (node != NULL) { Destroy(node->GetLeft()); Destroy(node->GetRight()); delete node } </pre>	<p>recursive를 이용하여 alphabetBST와 WordBST를 연결하여 삭제를 시켜준다. 이때 Postorder방식으로 제거하였다.</p>
<pre> <Insert> pCur = root, parent = 0 while(pCur) { parent = pCur if(node alphabet< pCur alphabet) go left child else if(node alphabet< pCur alphabet) go right child else return } pCur = node if(root have a data) { if(pCur->alphabet < parent->alphabet) parent Leftchild is pCur else parent Rightchild is pCur } else root = pCur </pre>	<p>먼저 pCur에 root를 넣고 pCur의 부모노드를 parent라고 하는데 0으로 초기화시킨다. while문으로 alphabetnode가 삽입될 위치를 찾는다. 현재 pCur의 alphabet보다 작으면 왼쪽으로 내려가고 pCur의 alphabet보다 크면 오른쪽으로 간다. 비교를 하며 내려가다 위치를 찾으면 pCur은 NULL이 되고 parent는 노드가 삽입될 부분의 부모노드가 된다. 그럼 pCur에 node를 대입하고 parent의 alphabet보다 작으면 parent의 왼쪽자식,크면 오른쪽 자식으로 set 하여준다. 만약 root가 존재하지 않다면 노드가 처음 들어온 것이므로 노드를 root로 하여준다. wordBST의 Insert는 alphabet을 word로 바꾸어 strcmp를 이용하여 비교하여 같은 방식으로 insert가 수행된다.</p>
<pre> <Print> if(strcmp(method,order)==0) go order function </pre>	<p>order에서 사용자가 입력한 BST 출력 방식과 비교하여 같은 문자를 가지면 그 방식을 적용한 함수를 호출한다. 적용된 ordering 방식이 없다면 false을 return 한다.</p>
<pre> <Search> pCur = root while(pCur) { if(alphabet < pCur alphabet) go left else if(alphabet < pCur alphabet) go right else return pCur } if(pCur == NULL) return 0 </pre>	<p>먼저 pCur에 root를 대입한다. 반목문에서 현재 가리키고 있는 알파벳과 찾을 알파벳의 크기를 비교해 작으면 왼쪽 크면 오른쪽으로 이동하며 찾는다. 크기가 작지도 않고 크지도 않으면 같다는 의미로 pCur을 return한다. wordBST에서는 alphabet대신 word를 비교하여 찾는다. 이후 같은 word가 있다면 같은 word를 반환한다.</p>
<pre> <Save> </pre>	<p>SAVE는 만약 순서대로 저장한다면 SAVE한</p>


<pre> currentNode = root if (root == NULL) return false while (1) { while(currentNode) { bst = currentNode->bst bst -> save s.push(currentNode) currentNode->go left } if(isempty) break; currentNode = s.top s.pop currentNode = currentNode->right } return true </pre>	<p>뒤에 LOAD가 되면 트리구조에 트리가 skew가 되므로 저장을 proorder 방식으로 하여 준다. 따라서 stack을 사용한다. 처음에 root가 비어있다면 이 alphabetBST에 데이터가 없으므로 false를 return 한다. currentNode를 root로 설정하여주고 반복문을 들어간다. preorder방식이므로 부모노드, 왼쪽자식, 오른쪽자식 순서대로 저장을 한다. alphabetNode가 존재할 때까지 반복문을 도는데 먼저 currentNode의 wordbst save로 이동한다. 이후 currentNode를 stack에 push하고 currentNode의 왼쪽자식으로 이동한다. 왼쪽자식이 존재한다면 다시 그 노드의 wordbst의 save로 이동하고 왼쪽으로 이동한다. 이후 가장 왼쪽 노드까지 이동하였으면 반복문을 나와 currentNode를 stack의 top으로 설정하여 준다. 이후 top을 pop하고 currentNode를 오른쪽 자식으로 이동하여준다. stack에 노드가 비어있으면 반복문을 빠져나가는데 이때는 모든 노드가 쓰여졌다는 뜻이므로 참을 return 해준다. wordBST로 이동하였을때도 마찬가지로 bst를 호출하는 부분에서 출력을 일어나게한다.</p>
<p><iterative pre-order></p> <pre> currentNode = root; while (1) { while(currentNode) { bst = currentNode->BST bst->Print("I_PRE") s.push(currentNode) currentNode = currentNode->go left } if(s.isempty()) return currentNode = s.top s.pop currentNode = currentNode->go right } </pre>	<p>currentNode를 root로 설정하여주고 반복문을 들어간다. alphabetNode가 존재할 때까지 반복문을 도는데 부모노드를 현재 가리키므로 같은방식의 wordBST의 Print를 호출한다. 이후 currentNode를 stack에 push하고 currentNode의 왼쪽자식으로 이동한다. 왼쪽자식이 존재한다면 다시 그 노드의 wordbst의 print로 이동하고 왼쪽으로 이동한다. 이후 가장 왼쪽 노드까지 이동하였으면 반복문을 나와 currentNode를 stack의 top으로 설정하여준다. 이후 top을 pop하고 currentNode를 오른쪽 자식으로 이동하여준다. stack에 노드가 비어있으면 반복문을 빠져나가는데 이때는 모든 노드가 쓰여졌다는 뜻이므로 참을 return 해준다. 마찬가지로 wordbst 부분에서는 alphabst iterative pre-order와 같은 방식이다. bst를</p>

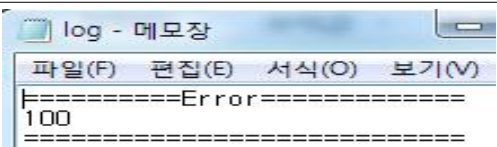
	호출하는 부분을 출력해주는 visit함수로 바꾸어 출력을 해준다.
<pre> <Recursive pre-order> if(currentNode) { bst = currentNode->bst bst->print preorder(currentNode->left) preorder(currentNode->right) } </pre>	재귀함수이므로 root를 넘겨주는 한번만 호출될 수 있는 함수를 오버로딩시켜 만든다. 따라서 처음 재귀함수 안에는 currentNode를 root로 받는다. 이후 먼저 부모노드의 bst를 R_PRE방식으로 호출하고 부모노드의 왼쪽자식을 인자로 자신인 함수를 다시 호출한다. 가장 왼쪽자식의 왼쪽으로 이동하였으므로 currentNode는 NULL이므로 함수를 빠져나와 이번엔 오른쪽자식을 인자로 재귀함수를 호출한다. 이와 같은 방식으로 currentNode가 모두 출력 할때까지 재귀함수가 호출되어진다. 마찬가지로 wordBST에서는 bst호출부분을 visit함수로 바꾸면 구성은 똑같다.
<pre> <Recursive in-order> if(currentNode) { inorder(currentNode->left) bst = currentNode->bst bst->print inorder(currentNode->right) } </pre>	재귀함수이므로 root를 넘겨주는 한번만 호출될 수 있는 함수를 오버로딩시켜 만든다. 따라서 처음 재귀함수 안에는 currentNode를 root로 받는다. 이후 부모노드의 왼쪽자식을 인자로 자신인 함수를 호출한다. 가장 왼쪽자식의 왼쪽으로 이동하였으므로 currentNode는 NULL이므로 함수를 빠져나와 부모노드의 bst를 R_inorder 방식으로 호출한다. 이 후 오른쪽자식을 인자로 재귀함수를 호출한다. 이와 같은 방식으로 currentNode가 모두 출력 할때까지 재귀함수가 호출되어진다. 마찬가지로 wordBST에서는 bst호출부분을 visit함수로 바꾸면 구성은 똑같다.
<pre> <iterlative in-order> currentNode = root while (1) { while(currentNode) { s.Push(currentNode) currentNode = currentNode->go left } if (s.isempty()) return currentNode = s.Top() s.Pop() } </pre>	currentNode를 root로 설정하여주고 반복문을 들어간다. alphabetNode가 존재할 때까지 반복문을 도는데 currentNode를 stack에 push하고 currentNode의 왼쪽자식으로 이동한다. 왼쪽자식이 존재한다면 다시 그 노드의 왼쪽으로 이동한다. 이후 가장 왼쪽 노드까지 이동하였으면 반복문을 나와 currentNode를 stack의 top으로 설정하여준다. 이후 top을 pop하고 부모노드를 현재 가리키므로 같은방식의 wordBST의 Print를 호출한다. currentNode를 오른쪽

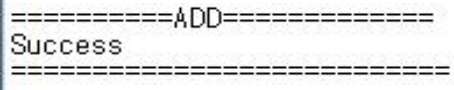

<pre> bst = currentNode->BST bst->Print("I_IN") currentNode = currentNode->go right } </pre>	<p>자식으로 이동하여준다. stack에 노드가 비어있으면 반복문을 빠져나가는데 이때는 모든 노드가 쓰여졌다는 뜻이므로 참을 return 해준다.</p> <p>마찬가지로 wordbst 부분에서는 alphabst iterative in-order와 같은 방식이다. bst를 호출하는 부분을 출력해주는 visit함수로 바꾸어 출력을 해준다.</p>
<pre> <Recursive post-order> if(currentNode) { postorder(currentNode->left) postorder(currentNode->right) bst = currentNode->bst bst->print } </pre>	<p>재귀함수이므로 root를 넘겨주는 한번만 호출될 수 있는 함수를 오버로딩시켜 만든다. 따라서 처음 재귀함수 안에는 currentNode를 root로 받는다. 부모노드의 왼쪽자식을 인자로 자신인 함수를 호출한다. 이후 가장 왼쪽자식의 왼쪽으로 이동하였으므로 currentNode는 NULL이므로 함수를 빠져나와 오른쪽자식을 인자로 재귀함수를 호출한다. 이 후 부모노드의 bst를 R_postorder 방식으로 호출한다. 이와 같은 방식으로 currentNode가 모두 출력 할 때까지 재귀함수가 호출되어진다. 마찬가지로 wordBST에서는 bst호출부분을 visit함수로 바꾸면 구성은 똑같다.</p>
<pre> <iterative post-order> if(!root) return s.push(root) AlphabetNode *prev = NULL while (!s.isEmpty()) { AlphabetNode *curr = s.Top if (!prev prev->Left == curr prev->Right == curr) { if (curr->Left) s.Push(curr->Left) else if (curr->Right) s.Push(curr->Right) } else if (curr->Left == prev) { if (curr->Right) s.Push(curr->Right) } } </pre>	<p>iterative post order 방식은 기존의 iterative 방식과는 고려해야 할부분이 많다. 먼저 root를 stack에 push 하여준다. 처음에는 prev를 NULL로 초기한 뒤 stack이 비어있지 않을때까지 반복하는 반복문을 들어간다. 처음 curr을 stack의 top을 가리키는 변수를 선언하여 준다. 만약 처음 실행되었거나 이전에 가리키고 있던 노드의 자식들이고 현재스택의 top의 왼쪽자식있으면 왼쪽을 자식을 stack에 넣고 왼쪽자식이 없고 오른쪽자식이 있다면 오른쪽자식을 넣는다. 처음이아니고 prev의 자식들이 아니라면 이미 부모노드와 왼쪽자식노드는 stack에 insert된 것이므로 현재노드의 오른쪽노드를 stack에 insert한다. 이 또한 아니라면 현재 출력해야 하는 노드이므로 alphabetBST에서는 WordBST를 호출하고 WordBST에서는 파일에 순서대로 출력한다. 이후 출력한 노드를 Pop하여 준다. 이 조건들을 실행하였</p>

<pre> } else { bst = curr->BST bst->Print s.Pop() } prev = curr } if (s.isEmpty()) break </pre>	<p>을 경우 prev 포인터를 현재노드로 연결하여주고 다시 반복문을 들어갈 때는 curr을 스택의 top노드를 가리키게 하므로 prev는 이전노드를 가리키게 된다. 이처럼 반복문이 stack에 노드가 삽입되어져 있을때까지 돌다가 노드가 없다면 반복문을 빠져나온다. 이 의미는 모든 노드가 post order방식으로 출력이 되었다는 의미이다.</p>
<pre> <iterlative level-order> currentNode = root while(currentNode) { bst = currentNode->bst if(currentNode->left) q.alPush(currentNode->left) if(currentNode->right) q.alPush(currentNode->right) if(q.alisempty) return currentNode = q.alpop; } </pre>	<p>먼저 currentNode를 root로 설정하여준다. currentNode가 존재하면 반복문을 들어가 먼저 currentNode의 bst를 불러준다. level order이기 때문에 부모노드 -> 왼쪽자식 -> 오른쪽자식 level 순서로 queue에 넣어준다. currentNode의 왼쪽 자식을 queue에 넣어주고 이후 오른쪽 자식을 queue에 넣어준다. 이후 queue가 비어있으면 return을 해주고 비어있지않다면 currentNode에 pop해준 노드를 넣어준다. wordBST에서도 같은 방식을 적용하여 BST호출부분을 VISIT함수로 바꾸어주면 Level Order가 된다.</p>

[Result Screen]

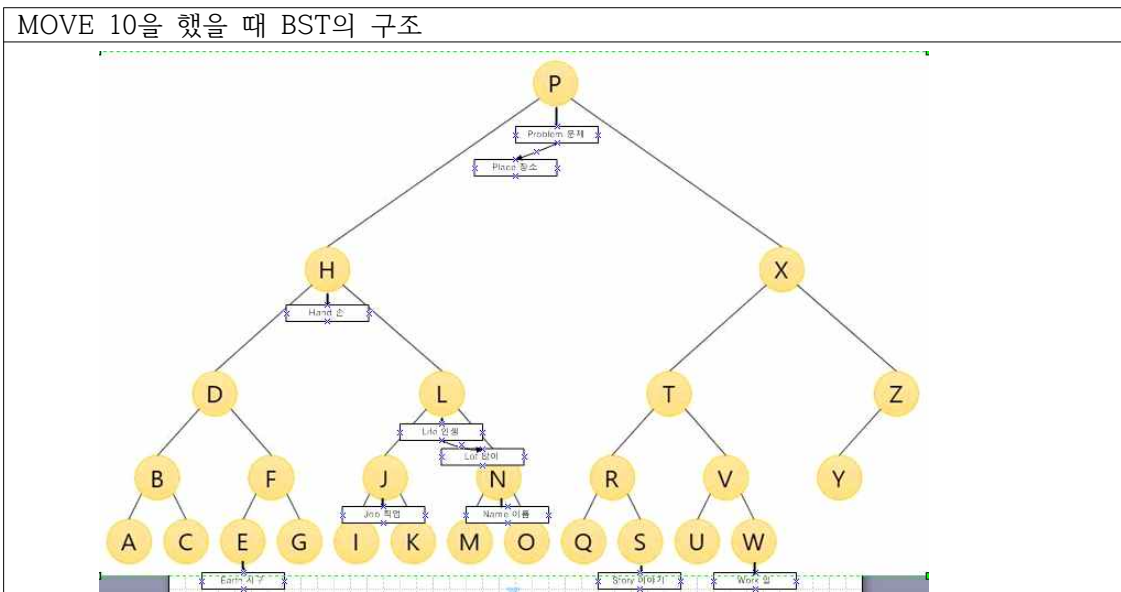
command.txt	
	command.txt 파일로써 명령어들의 집합이다. 처음 파일에서 명령어들을 읽을 때 빈칸이 있어도 그 다음 줄을 읽어 명령어를 수행하기 때문에 문제가 없이 작동한다.

log.txt < LOAD >	
	처음 프로그램이 시작되었을 때 LOAD가 명령어로 읽히면 아직 단어장 프로그램이 저장되어 있지 않기 때문에 에러코드를 출력한다.

log.txt < ADD >	
	
ADD 명령어가 실행되면 Word.txt 파일에 있던 단어들이 to_memorize_word.txt 파일로 옮겨진다. 이때 to_memorize_word.txt. 파일에 있는 단어들은 Queue의 형태로 프로그램에 저장이 되어있다.	

log.txt < MOVE 10 >	
<pre>=====MOVE===== Success =====</pre>	MOVE 10은 to_memorize에 있는 단어 10개를 memorizing으로 옮기는 부분이다. 따라서 10개가 binary search tree로 구성되어 있음을 알 수 있다.

log.txt < PRINT MEMORIZED >	
<pre>=====Error===== 700 =====</pre>	현재 TEST명령어를 수행하지 않고 바로 MEMORIZED명령어를 실행해 보았다. 현재 memorized 파일에는 단어가 없으므로 에러코드가 출력된다.



log.txt < PRINT MEMORIZING R_PRE, I_PRE >	
<pre>=====PRINT===== problem 문제 place 장소 hand 손 earth 지 life 생명 lot 직업 job 직업 name 이름 story 이야기 work 일 ===== =====PRINT===== problem 문제 place 장소 hand 손 earth 지 life 생명 lot 직업 job 직업 name 이름 story 이야기 work 일 =====</pre>	preorder 방식을 두 가지 방법을 통하여 구현 하였던 것을 출력하였다. 부모노드를 먼저 출력하고 왼쪽자식, 오른쪽자식 순으로 출력이 되므로 왼쪽 그림과 같이 출력이 이뤄진다.

log.txt < PRINT MEMORIZING R_IN, I_IN>	
<pre> =====PRINT===== earth 지구 hand 손 job 직업 life 인생 lot 땅 name 이름 place 장소 problem 문제 story 이야기 work 일 ===== =====PRINT===== earth 지구 hand 손 job 직업 life 인생 lot 땅 name 이름 place 장소 problem 문제 story 이야기 work 일 ===== </pre>	<p>Inorder 방식을 두 가지 방법을 통하여 구현 하였던 것을 출력하였다. 왼쪽자식을 먼저 출력하고 부모노드, 오른쪽자식 순으로 출력이 되므로 왼쪽 그림과 같이 출력이 이뤄진다.</p>

log.txt < PRINT MEMORIZING R_POST, I_POST >	
<pre> =====PRINT===== earth 지구 job 직업 name 이름 lot 땅 life 인생 hand 손 story 이야기 work 일 place 장소 problem 문제 ===== =====PRINT===== earth 지구 job 직업 name 이름 lot 땅 life 인생 hand 손 story 이야기 work 일 place 장소 problem 문제 ===== </pre>	<p>postorder 방식을 두 가지 방법을 통하여 구현 하였던 것을 출력하였다. 왼쪽자식을 먼저 출력하고 오른쪽자식, 그리고 부모노드 순으로 출력이 되므로 왼쪽 그림과 같이 출력이 이뤄진다.</p>

log.txt < PRINT MEMORIZING I_LEVEL>	
<pre> =====PRINT===== problem 문제 place 장소 hand 손 life 인생 lot 땅 job 직업 name 이름 earth 지구 story 이야기 work 일 ===== </pre>	<p>Levelorder 방식을 두 가지 방법을 통하여 구현 하였던 것을 출력하였다. 레벨 순으로 출력이 되므로 왼쪽 그림과 같이 출력이 이뤄진다.</p>

log.txt < PRINT TO_MEMORIZE>	
<pre> =====PRINT===== make 만들다 different 다른 important 중요한 person 사람 clothes 옷 movie 영화 activity 활동 example 예 letter 편지 fire 불 part 부분 plan 계획 plant 식물 fun 재미 listen 듣다 learn 배운다 each 각각 same 같음 bird 새 trip 여행 vacation 휴가 summer 여름 course 강좌 spring 봄 autumn 가을 winter 겨울 space 공간 street 거리 paper 종이 newspaper 신문 face 얼굴 mind 마음 </pre>	<p>TO_MEMORIZE에 QUEUE 형식으로 저장되어 있는 파일을 head부터 순서대로 출력이 이뤄진다. 따라서 Word.txt 파일에 있는 단어와 같아진다.</p>

log.txt < MOVE 90 >	
<pre> =====MOVE===== Success ===== </pre>	<p>현재 단어는 10개만 입력이 되어있다. 따라서 100개 이하까지 MEMORIZING에 저장될 수 있는데 90개를 옮기는 것은 가능하다. 따라서 총 100개의 단어가 MEMORIZING에 저장되어 있다.</p>

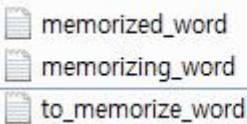
log.txt < MOVE 10>	
<pre> =====Error===== 300 ===== </pre>	<p>MEMORIZING에는 단어가 이미 100개가 들어가기 때문에 더 이상 들어 갈 수 없다. 따라서 에러코드가 출력된다.</p>

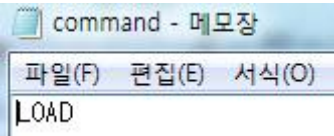
log.txt < UPDATE AREA 구역, HAND 손손 >	
<pre> =====UPDATE===== area 지역 -> 구역 ===== =====UPDATE===== hand 손 -> 손손 ===== </pre>	<p>모든 구조에서 알맞은 단어를 찾아 단어의 뜻을 바꿔준다. 이때 대문자로 입력하여도 변화하여 찾기 때문에 오류가 나지 않는다. 왼쪽 결과화면을 통해 의미가 바뀌었음을 알 수 있다.</p>

log.txt < SEARCH HAnd>	
<pre> =====Search===== hand 손손 ===== </pre>	모든 구조에서 hand를 찾아 알맞은 단어가 있으면 그 단어의 뜻을 출력하고 없다면 에러코드를 출력한다.

log.txt < TEST AREA 구역 TEST HAnD 손손 TEST LiFE 인생 TEST LOT 마니 >	
<pre> =====TEST===== Pass ===== =====TEST===== Pass ===== =====TEST===== Pass ===== =====Error===== 500 ===== </pre>	MEMORIZING에 있는 단어들을 TEST를 하여 단어와 그 단어의 뜻이 맞다면 PASS를 출력하고 MEMORIZIED로 옮겨진다. 이때 단어뜻이 틀리거나 그 단어가 없으면 에러코드가 왼쪽 그림과 같이 출력된다.

log.txt < PRINT MEMORIZED>	
<pre> =====print===== area 구역 hand 손손 life 인생 ===== </pre>	위의 그림에서 3개의 TEST가 PASS하였다. 따라서 PASS한 단어 3개가 MEMORIZIED에 저장되어 있으므로 PRINT가 왼쪽 그림과 같이 출력된다.

log.txt < SAVE >	
<pre> =====SAVE===== Success ===== </pre>	SAVE를 하면 왼쪽그림 아래와 같이 단어 텍스트 파일이 생성되어 단어가 저장이 된다.
	

command.log	
	이후 다시 각각의 단어들이 저장되어 있는 단어들을 지우지 않고 다시 새로 프로그램을 시작하여 LOAD를 하면 각각의 단어장에 있던 단어가 다시 각각의 구조로 저장이 되어 LOAD가 성공적으로 되었다는 출력이 된다.
<pre> =====LOAD===== Success ===== </pre>	

[Consideration]

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자신의 점수(10)
이명진	주요 알고리즘 짚오	10
고찰	<p>이번 프로젝트를 진행하면서 binary search tree 구조와 각각의 방법으로 단어들을 Ordering 하는 부분이 어려웠었다. BST를 구현하면서 AlphabetBST에서 WordBST를 호출하는 알고리즘을 확실히 이해하지 않고 프로젝트를 구현하다보니 코드나 알고리즘이 뒤죽박죽 꼬였었다. 하지만 다시 제안서를 읽으며 차근차근 생각하며 다시 코드를 새로 만들었다. 이제부터는 먼저 어떻게 구성되어야 하는지, 알고리즘부터 확실히 알면 조금 더 수월하게 코드를 구성할 수 있다는 생각이 들어 다음 프로젝트부터는 조금 더 수월하게 설계를 할 수 있겠다는 생각이 들었다. 하지만 팀 프로젝트로 팀원을 만나 같이 알고리즘을 분석하고 내가 생각하지 못했던 부분의 예외와 코드를 중복해서 사용하였던 부분들을 알면서 많은 도움이 되었다. 설계에서 중복되는 부분들을 어떻게 구현하였는가를 보면서 조금 더 코딩에 대한 시각이 넓어진 느낌을 받았다. 특히 메모리를 해제하는 부분은 각각의 소멸자에서만 해제를 해주면 메모리가 모두 해제가 된다고 생각했었다. 하지만 구현을 할 때 동적할당을 필요할 때 마다 사용하여 소멸자에서 모두 해제되지가 않았었다. 소멸자를 구현하는데도 전체적인 느낌으로 보기까지 많은 시간이 걸렸지만 메모리가 전부 해제되지 않았다는 결과를 알고 다소 충격적이었다. 특히 마지막에 소멸자를 작성하기에는 문제가 너무 많이 생겼었다. 이 문제를 해결하며 느낀 점이 많은데 동적할당을 해줄 때에는 클래스에서는 소멸자를 그때그때 작성하도록 하고 따로 동적할당을 할 때면 그때그때 메모리를 해제하여 메모리 누수가 일어나지 않도록 해야겠다는 생각이 들었다. BST 부분에서 각각을 Ordering 할 때 iterative post order 부분이 생각하기에 쉽지 않았다. iterative는 stack을 사용하며 push, pop으로 print를 하여야 하는데 이 부분에서 어려웠다. stack에서 빠져나오면서 출력을 할지 아니면 어디에서 push를 해주어야 하는지 생각하기에 너무 쉽지가 않았다. 하지만 지난번 설계 HW를 통해 미리 iterative와 recursive 부분을 공부를 해서 이번 프로젝트에 도움이 많이 되었던 것 같다. 처음 공부할 때에 내가 과연 이 다음에 recursive나 iterative를 이용하여 구현 가능한 부분을 이를 이용하여 구현하면 코드가 더 간결해지고 성능도 더 좋아질 수 있다는 생각이 들었다.</p>	

이름	프로젝트에서 맡은 역할	본인 스스로 생각하는 자신의 점수(10)
이종찬	객체지향 위한 재구성	7
고찰	<p>보인은 설계를 재수강하는 학생입니다. 이미 수강한 실습 자료들과 경험을 토대로 이명진 학생과 프로젝트를 구상하였습니다. 기능별 함수를 어떻게 나눌것인지 토의하고, 알고리즘을 스도코드를 통해 서로 이해를 도왔습니다. 이명진 학생이 기능별 알고리즘을 작성하면 저는 그부분을 함수화하고 전체 프로젝트에서 함수 혹은 클래스로서 독자적으로 존재할 수 있도록 구성하였습니다. 각각의 함수들과 클래스들을 프로젝트에서 정상 작동을 할수 있게 수정하면서 수차례 문제에 부딪혔습니다. 과제에서 제공해주신 기본 클래스들과 함수의 선언 형식이 오히려 방해가 되었습니다. 선언 형식의 수정이 불가능한 상황에서 각각의 함수들이 서로만의 기능을 수행하고 중복된 기능은 따로 함수를 빼는 과정에서 기본 형식과의 문제였습니다. 대표적으로 매니저 클래스내에 파일 클래스를 추가하여 5가지의 파일을 읽고 쓰는 기능과 소스를 이곳에서 모두 관리할 수 있도록 작성하였으며, 각각의 파일과 파일의 정보를 파일 노드에 저장하여 필요할 경우에 pop을 이용하여 파일 정보에 접근할 수 있도록 하였습니다. 매번 파일을 읽으면서 매니저 기능을 수행하는 것보다 한번에 모든 파일을 읽어들이고 노드화 시킨다음 매니저의 기능을 수행하게 하는게 더욱 좋은 프로그램의 흐름이라고 생각했습니다.</p>	