설계과제 최종보고서

설계과제명:

적절한 자료구조를 활용, 제한사항을 만족하는 검색엔진의 구현

교과목명	자료구조와 실습
담당교수	
팀 원	

설계과제 요약서

과 제 요 약 서

설계과제명	검색 엔진에 알맞은 자료구조 선택 및 검색 엔진 구현
주요기술용어	시간 복잡도, 이진검색 트리, 해시 테이블, 원형 큐, 체이닝

1. 과제 목표

주어진 제약요소 및 시간 복잡도를 고려한 검색엔진을 구현한다. 크게 두가지 부분으로 구성되는데, (1)색인: 동일 폴더 내 주어진 문서파일(doc001~100.txt)로부터 문자를 읽어 색인으로 저장하고, 읽어 들인 문서의 수, 색인된 문자의 수, 색인 도중 발생한 비교연산 횟수 정보를 출력한다. (2)검색: 단어를 입력 받으면 해당 단어가 많이 등장한 문서 순서대로 단어의 정보를 출력한다. 단어는 앞뒤 각 3 단어를 포함하여 출력하며, 출력이 완료되면 구성된 자료구조 내에서의 비교 연산 횟수 정보를 출력한다.

2. 수행내용 및 방법

강의시간에 배운 자료구조들 색인, 검색 단계에 알맞은 자료구조를 선택하고, 검색엔진을 구현한다. 프로젝트 중 색인 단계에는 체이닝 기반 해시 테이블과 이진검색 트리의 조합이, 검색 단계에는 원형 큐가 주로 사용되었다. 1 인 프로젝트로 진행되었으며, 사용될 자료구조의 탐색, 자료구조 간 비교 및 선택, 부수적인 프로그램 작성, 색인 및 검색 프로그램 작성 및 설계 지침서에 언급된 예제 수행의 단계를 거쳤다. 각 단계는 수행 계획서에 언급된 순서와 유사하게 수행되었다.

3. 수행 결과

설계 지침서에 언급된 예시(doc001 ~ 004.txt)에 대하여 검색을 수행.

색인 단계: 입력 받은 문서의 개수, 색인된 단어의 개수, 색인 구성 중 수행된 문자열 비교 횟수 출력 구현

검색 단계: 단어가 많이 포함된 문서부터 출력, 단어 전후 3 단어 포함 출력, 문자열 비교연산 횟수 출력 구현

#을 문자열의 첫 글자로 입력 받을 때까지 계속 단어 입력 가능.

4. 결과 분석

읽어 들이는 파일이 많은 경우를 고려하여 체이닝 기반 해시 테이블 및 이진검색 트리를 활용하여 검색엔진을 구현하였다. 요구사항(3.수행 결과 : 색인/검색 참조)에 따라 정상적으로 작동하며, 특히 검색 수행 시 문자열 비고 연산횟수, 즉 시간 복잡도가 해시테이블의 특성과 유사한 모습을 보였다. 또한 검색 결과 출력을 위해 문자를 읽어올 때 char* 타입의 원형 큐를 사용하여, 공간 복잡도를 감소시켜 공간적 이점을 얻었다.

1. 서론

1.1 설계과제 목적

강의를 통해 습득한 자료구조들에 대하여 각 자료구조들의 특성 및 성능을 비교하여 가장 검색엔진에 적합해 보이는 자료구조를 선택한다. 이후 선택된 자료구조에 기반하여 효율적인 알고리즘 및 제한 요소를 고려하여 검색엔진을 실제로 구현한다. 위 과정을 통해 자료구조에 대해 좀 더 깊게 공부하고, 프로그램 설계 및 구현 능력을 향상시킨다.

1.2 설계과제 내용

[1] 프로그램 요구사항 및 제한 요소 분석

글 및 그림을 통해 설계 지침서에 언급된 정보를 요약하고, 떠오르는 정보를 기록한다. 우선 프로그램의 기능에 대하여 각 부분의 요구사항을 요약한다. 검색 단어의 특징, 검색되는 문서의 내용, 색인 후 결과 출력 시 제한 요소, 단어 구성 방식 및 각 단계에서 떠오르는 아이디어를 정리한다.

위 정리된 정보들을 이용하여 일차적으로 프로그램을 어떻게 구성할지 고려했다. 예를 들어, 색인 단계에 사용될 자료구조를 물색하는 단계 중, [ASCII 기반, 알파벳만 인식, 대소문자 구분 안함] 이라는 정보를 보고, 알파벳 첫 글자로 구분하는 자료구조가 있을까 고려하다가, 해시 테이블을 떠올렸다. 이외의 경우, 완벽히 일치하는 단어만 검색한다는 점에서, strncmp 함수를 떠올리는 등, 브레인스토밍을 수행했다. (실제 고려사항은 [추가 1]에서 볼 수 있다.)

[추가 1]

검색엔진의 기능 a-z, A-Z

- 영문문서, 검색: ASCII, 파일명 doc001.txt ~ doc100.txt, 띄어쓰기 구분X,
- 색인 (어떤 단어가 어느문서에 몇번이나 나타나는가) ~ 몇번째에 나타나는지 기록해두어야함 (색인과정) 저장 가능!
- 검색 (문서명, 단어출현빈도, 출현부 전후 3단어)

제한

1) 검색 단어
   - 대·소문자구분X
   - 검색단어포함 대상X, (inform ~ information)
   - 입력은 알파벳만 → 정확히 그 단어만
   - * 반대 경우는?
2) 검색 대상문서 파일 내용
   - ASCII (영문자 + 제어문자 + 문장부호), 단어사이는 공백문자, ('') 로 구분 (공백 및 개행문자)
   - 영문자 부분만 단어에 허용 (문장부호 무시)
3) 색인 및 검색 결과 출력, 제한 요소
   - 색인후 ① 총 문서수 ② 색인된 총 단어수 ③ 스트링 비교연산 횟수 출력 (색인 완료끼리 몇번에 str 비교?)
   - 검색후 ① 스트링 비교연산수 (자료구조 내 자료들과 비교가 일반적 어떻게 되는가?)
   - 검색단어 출력: 가장 많이 포함된 파일순,
4) 검색 방식
   - 자료구조, 알고리즘 선택, 가능 : bint, 기능상 완성도 및 효율성을 계산,

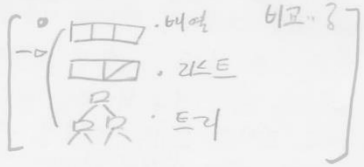
① → toLower 함수. 122 97

- 검색, 대소구분X, 단어는 소문자로 표기 → 'a'-'a' ~ 'z'-'a' 안에서 보관가능!

0 ~ 25

a ... 첫글자로 보관 가능할듯?
getLoc(ch 2) → 배열도 가능?
return (z-a) → 고려된 개수

→ 버퍼크기



검색 반드시 그 단어만 ~ strncmp(D, S, MAX)

② 단어사이 공백or개행으로 구분 → 그냥 fputs / fscanf 등 사용 ~ 한번에씩 받을거라면?
문장부호 무시 → 알파벳만 표시 → isalpha() / 소문자로 바꿀 필요
'ApPlE' → apple (소문자 ⊕ 알파벳 only 조합으로!)

③ 색인관련 → 문서수 → 읽은 파일개수, 규칙적, → 파일읽기 성공마다 ⊕
→ 색인된 단어수: 단어가 새롭게 "추가" 될때 → 자료구조 내에 없었을 때 → insert 하면서 추가!
→ 스트링 비교연산수: 순수하게 전체 strcmp 수 → 함수 따로 만들어서 ++,
← 문서 내 비교 ← 자료구조에 넣을 때 비교

[2] 분석된 요구사항에 대한 적절한 자료구조 탐색 및 선택

위 단계에서 분석된 요구사항에 기반하여 적절한 자료구조를 선택한다.

색인 단계의 경우 요구사항을 고려하여 선택된 3 개의 자료구조 [해시 테이블, 리스트, 이진탐색 트리]의 삽입/탐색 시 시간 복잡도 및 특징을 비교하였다. 이를 분석하면, 해시 테이블이 O(1)으로 최선의 선택이었고, 최종적으로 해시 테이블을 선택했다. 이때, 검색 시 속도를 좀 더 빠르게 만들기 위한 방식을 고려하였는데, 기존 체이닝 해시 테이블의 경우, 어떤 입력과 동일한 해시 값을 가지는 정보에 대해 버켓의 값을 일일이 비교하며 위치를 찾아야 하므로, 버켓 내에서는 O(n)의 속도로 검색하게 구성되어 있었다. 이러한 특징을 개선해보고자 평균적인 검색 속도가 O(log n)인 이진검색 트리를 리스트 대신 해시테이블의 체이닝으로 구성하기로 결정했다.

검색 단계의 경우 (정확히는 단어를 검색하여 파일 내부에서 단어를 출력하는 단계), 초기에 [리스트, 큐, 원형 큐]라는 3 개의 자료구조를 고려했으나, 리스트와 리스트 기반의 큐는 비슷한 위치에 있으므로, 대신 [리스트 기반 큐, 배열 기반 원형 큐] 두가지 자료구조를 비교하기로 했다. 기본적으로 두 자료구조 모두 큐에 속하므로 스펙상 차이는 크게 존재하지 않았다. 이때, 단어는 해당 단어 및 앞뒤로 3 단어씩 총 7 단어만 출력된다는 정보에 기인하여 굳이 큐를 동적할당 방식으로 만들 필요가 없다고 판단, 동적할당 과정이 없어 공간을 덜 사용하는, 고정된 크기의 정보를 저장하는 배열 기반의 원형 큐를 채택하기로 결정했다. 이후, 구체적인 작동 방식을 개략적으로 설정했다.
(고려 내용은[추가 2]을 참고)

[추가 2]

색인 · 단어 · 문서 멀티 : 해당 단어를 저장하는데, 얼마나 시간이 필요한지?

	삽입	탐색	비고
① 해쉬테이블 배열	O(1).	O(1)	해싱 알고리즘 나쁘면 O(n)으로 상승, 공간 필요
② 리스트	O(1) / O(n)	O(n)	포인터 +1
③ BST	O(logn)	O(logn)	최악의경우 O(n) 포인터 +2

해시테이블의 성능이 가장 좋아보임. 충분한 공간 사용시, 획기적으로 성능 향상 가져오게 된다.
해시테이블 의 크기는 적절히 정하는게 중요하나, 너무 커서는 안됨
방법 ① 첫 문자에 대해 해싱
② 1000 사이즈 정도의 해싱. …
해싱 알고리즘은 어떻게 구성해야 할지 고민도 필요.
BST의 경우 최악의 상황에는 O(n)의 비교를 하나, 보통 이런 식은 흔치 않음 · 구현 어렵지 않고, 탐색이 해시테이블만큼 바르지는 않으나, 평균적으로 O(logn) 이 되므로, 단어 많아도 최소한 리스트 기본보다는 빠름,
성능 보완하기 위해, 해시테이블 → BST 순으로 접근하도록 만들기!

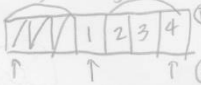
단어 위치 정보의 사용…
장점 : 단어 위치 정보를 기반으로 해당 위치로 str 비교 없이 바로 이동 가능,
단점 : 모든 단어 저장…, → 1 단어당 10 byte 정보 필요하다고 가정해도 단어수가 문서당 100개면
10 * 100 * 100 = 100000 byte의 저장 공간 필요…, 공간이 낭비되는 경향 존재,
결론 : 처음 생각할때는 좋아 보였으나, 공간복잡도 고려하면 최악. 채택X, (실제로 루틴의 획기적 감소 하는 것도 X)

검색

앞쉬 3단어 저장할 공간이 필요 → 3+1+3=7, 최대 7개 공간 필요
자료 타입 char*, 배열로는 너무 클듯 & 단어별 크기 편차 큰데 → 동적 할당 이용해야 될,
방식, ① 리스트, ② 큐, ③ 원형 큐
리스트와 큐는 거의 비슷하게 동작 가능, 리스트 전기,
사실, 고정된 크기를 가지면 큐를 사용할 이유가 없음, 동적할당으로 큐 만들 이유도 없어보임,
필요한 기능은 "삽입" "삭제"

	삽입	삭제	공간 사용	비고
① 리스트기반 큐	O(1)	O(1)	약 144 bytes	char* 포인터 (8) + 다음 큐 포인터 (8), 0~7 사이의 노드개수 계산, 동적할당으로 생성, front · rear 포인터도 필요 → 총 9개 노드필요 16X9=144
② 원형 큐	O(1)	O(1)	약 64 bytes	char* 포인터 7개 + int 타입 front/rear (56, 2X4)

프로그램의 특성상 리스트 기반의 큐는 필요 없다,
원형큐 채택,
작동 방식, front - rear 사이의 cur 이라는 것을 설정,



① 맨 처음 단어 3개는 앞 단어가 없으니, 처음의 NULL은 3개 넣어 첫번째 값이 cur이 될수 있게 설정, feof 아니고, 큐 꽉 찰때까지 넣어줄 것,
② front와 cur은 항상 3의 거리를 유지, 계속 반복는 delete & insert (feof 아닐때만)
③ cur이 찾는 단어인 경우, front ~ rear까지 NULL 아닌 값을 출력,

rear이 멈추고, front의 delete만 계속 수행되다가 cur · rear 넘어서면 (단어 다읽음)
이후 front에서 delete만 계속 수행 (큐 빌때까지)

① 동일 단어가 연속으로 나올 수도 있음,
② 단어가 최소 3단어 안에 나오면 앞 단어가 3개 이상 보장 X,

[3] 선택된 자료구조 기반 프로그램 구현

기초적으로 필요한 함수(파일 이름 설정, 단어 읽어 오기)들을 우선 구현한 후, 근본적인 기능인 색인 및 검색 기능을 구현한다. 구현하는 예시는 다음과 같다.

[1] void setFileName(int number)

: 1~100 사이의 숫자를 입력 받아, 파일 이름을 나타내는 filename 의 숫자 부분만 교체하는 함수로, 100 의 자리 숫자까지 각 자리를 구분하여 저장한 이후, 원 파일 이름의 각 자리와 교체한다.

[2] char* strInit(char* str)

: 문서 내에서 입력 받은 단어를 색인 과정에서 사용할 수 있도록 변경하여 돌려준다. 이때, 알파벳이 아닌 글자는 삭제, 알파벳은 소문자로 변경하여 만들어지는 단어를 동적할당해 돌려준다.

[3] char* getrawStr(char* str)

: 검색 이후, 원형 큐에 저장 될 단어를 저장하는 함수로, 단순히 문자열을 동적할당 하여 돌려준다.

[a]색인 기능 구현

색인 기능 구현을 위해 구조체 기반의 자료구조를 설계한다. 해시 테이블 역할의 "Dict" , 이진검색 트리 역할로, 각 단어를 의미하는 "Word" 구조체, 해당 단어가 포함된 파일 정보를 기억하는 링크드 리스트 기반의 "File" 구조체를 설계하고, 정보를 저장하기 위해 필요한 여러가지 종속적인 함수들을 설계한다. 종속적인 함수들은 기본적인 입출력, 생성, 삭제, 비교 등의 기능을 수행한다.

[b]검색 기능 구현

검색 수행 시 필요한 기능을 구현한다. 단어 검색 자체 기능은 main 함수에 구현한다. 색인 단계에서 구현한 3 개의 구조체를 통해 입력 받은 단어를 검색하여, 정보를 알아낸다. 검색 기능은 정확히 말하면 검색 후 "단어 출력 기능" 이라고 부를 수 있는데, 최대 7 단어를 출력하는 특징을 고려하여 "cQueue" 라는 원형 큐 기반 자료구조를 설계하고, 이에 종속적인 함수(add, delete, print 등 원형 큐의 기능 및 단어 출력 동작과 관련된 함수)를 구현한다. 구현된 함수를 이용하여 문자열을 요구사항대로 출력할 수 있도록 프로그램을 조정한다.

위 내용은 직접적으로 자료구조 및 알고리즘과 관련되므로, 프로그램 구성에서 추가적으로 설명하겠다.

[4] 프로그램 작동 확인, 개선점 파악 및 보완

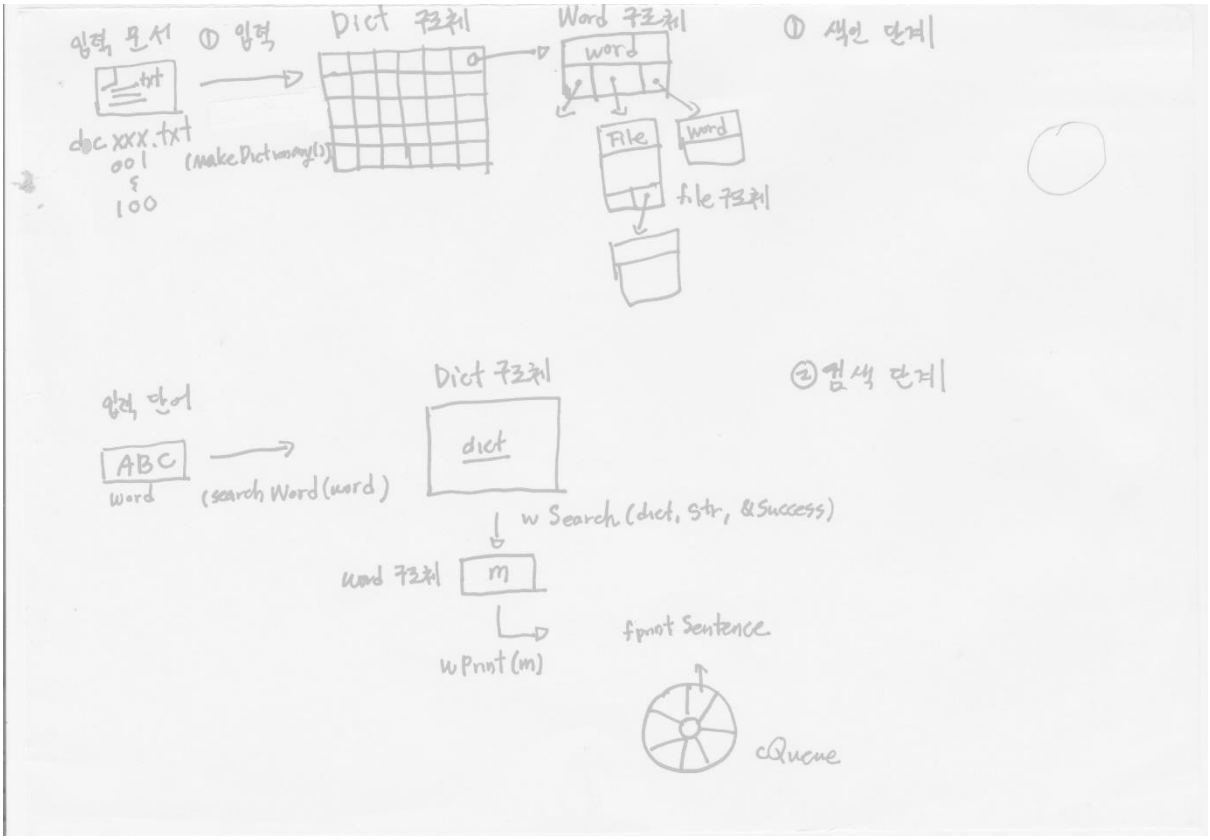
위 과정을 거쳐 프로그램을 실행하고, 디버깅 과정을 거친다. 입력될 파일을 프로그램이 저장되어 있는 파일에 두고 프로그램을 실행하여 결과를 파악한다. 문제점이 발생하면 디버깅 과정 및 보완 과정을 거친다. 이 과정은 실제 구현 과정 전반에서 수행되었다. 주로 문자열을 처리하는 부분에서 발생하는 NULL 문제등을 디버깅하였다.

1.3 진행 일정 및 개인별 담당분야

성명	담당 분야	참여도(%)
	자료구조 고안 및 프로그램 설계(과제 전반)	100

2. 프로그램의 구성

2.1 전체 구성도



위 구성도는 자료구조에 집중하여 나타낸 모습을 가진다.
해당 구성도는 크게 색인 단계와 검색 단계로 나뉜다.

[색인 단계]

[1] **DictionaryProject.exe** 파일을 실행하면, "makeDictionary" 함수가 실행된다.
[2] 프로그램은 파일과 **동일한 폴더 내에 위치한 문서** (doc001.txt~doc100.txt)에 대해 순차적으로 접근한다. 이때, 파일이 존재하면 file_count 를 1 증가시키고, [3] 과정을 수행한다.
[3] 문자열 처리를 거친(strInit) 단어에 대해 Dict 내에서 탐색을 시도한다. 단어는 해시함수를 통해 Dict 내부에 Word 구조체로 저장된다. 이 과정 중, 문자열 비교연산이 발생하면 cmp_insert_count 를 증가시킨다. Word 함수는 다른 해시 함수를 이용하여 트리 내부에 대한 단어의 위치를 지정한다. 파일 관련 정보(파일 이름, 등장 횟수)는 File 구조체로 저장되어, Word 구조체에서 포인터 형식으로 참조되게 된다. 만약 단어가 이미 자료구조 내에

존재한다면, File 구조체를 만들거나, File 구조체의 count(파일 내 단어 개수)만 증가시킨다.
반대의 경우, Word 구조체 및 File 구조체를 생성하고
[4] 색인 과정에서 얻어낸 프로그램 관련 정보를 출력한다.

[검색 단계]

[1] 색인과정을 끝마친 후, 단어 입력을 기다린다.
[2] 단어를 입력 받으면, 해당 단어를 "searchWord" 함수를 통해 Dict 내에서 탐색한다. Dict 자료구조는 단어를 문자열 처리(strInit)한 후, 해시 함수를 통해 해시 값을 얻는다. 해당 해시 값에 대해 "wSearch"을 수행한다.
[3] wSearch 를 통해 얻은 Word 구조체의 File 구조체 정보를 이용하여 파일이 저장된 경로를 가져온다. 이 과정 중, 문자열 비교연산이 수행되면, cmp_search_count 를 증가시킨다. 이후, 얻은 정보를 이용하여 파일 내 단어 정보를 출력한다. 이 과정에서 단어들은 cQueue 자료구조: 원형 큐에 순간마다 저장되면서 내부에 속한 단어 정보를 출력하게 된다. (fprintSentence).
[4]검색 단계가 끝난 후, 비교 연산 횟수를 출력한다.

2.2 자료구조 및 알고리즘

주된 자료구조를 설명하기 위해서 File – Word – Dict 순서로 설명하도록 한다.
우선, 부수적인 cQueue 자료구조 및 hash 함수를 설명하겠다.

[1]

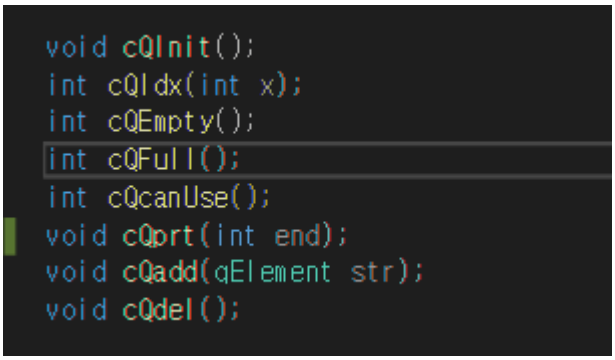
[A] cQueue 자료구조

```
#define QSize 8 // 최대 원소 7개

typedef char* qElement;

qElement cQueue[QSize];
int front;
int rear;
int cur;
```

cQueue 는 원형 큐 자료구조이다. 특이한 점은 내부에 cur 을 추가로 정의하고 있다는 점이다.

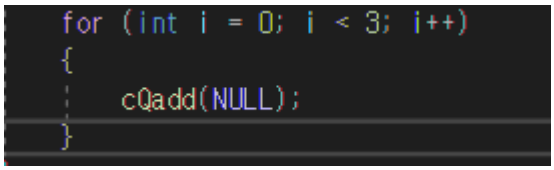
cur 은 검색 이후 문서 내 단어를 출력하는 단계에서, 검색 대상이 되는 단어를 의미한다. 문장을 출력할 때, cur 과 값이 같은 지 여부를 비교한 후, 문장을 출력하게 된다. 원형 큐의 경우 최대 7 개의 데이터 밖에 가질 수 없으므로, front 나 rear 을 기준으로 단어를 비교하면, 앞 3 단어를 다시 읽어올 수 없거나, 연속된 단어를 인식하는 부분 등에 있어서 문제가 발생한다. 따라서, cur 이라는 변수를 따로 두어 어떤 경우에도 front 부터 rear 까지의 정보를 정상적으로 출력 가능하게 설계했다,

함수는 다음과 같다.



대부분은 일반적인 원형 큐와 완전히 동일한 행동을 하므로, 몇 개 함수만 언급하겠다.

{1} cQInit : 원형 큐를 초기화하는 부분이다. Front, rear, cur 값을 0 으로 초기화 한다.
중요한 부분은 다음과 같다.



시작하자 마자 큐의 내부에 3 개의 NULL 을 삽입한다.
기본적으로 cur 은 항상 front 보다 3 칸 앞에 위치해야 한다. 문장의 출력은 앞 3 단어를 포함한 7 단어가 기본이기 때문이다. 이때, 입력되는 첫 3 단어에 대해서는 front 보다 절대 3 칸 앞에 위치할 수 없다. 앞에 최소 3 개의 단어가 있어야 front 보다 cur 이 3 칸 앞에 있을 수 있는 상황인데, 앞 3 개 단어에 대해서는 자신의 앞에 3 개의 단어가 존재할 수 없기 때문이다. 이러한 문제를 막기 위해, 시작 단계에서 NULL 을 3 개 삽입하여 모든 단어에 대해 일관성 있게 cur 이 front 의 3 칸 앞에 있을 수 있도록 만들었다.
{2} cQprt : 문서 내 검색된 단어를 출력한다. cQueue 의 front+1 부터 rear 까지의 값을 출력하는 함수이며, end 에 feof(file) 값을 입력 받아, 문장이 끝에 도달했는지 여부를 알아내, 문장 끝의 … 출력 여부를 결정한다.

언급하지 않은 함수는 cQueue.h 파일 내에서 찾아볼 수 있다.

[B] hash.h

```
const int priNum1[] = { 59,7,37,19,17, 23,71,29,5,53, 67,2,11,47,41, 31,13,3,43,61 };
const int priNum2[] = { 71,31,61,19,47, 13,5,3,2,67, 37,29,7,41,23, 59,43,53,11,17 };
```

```
int hash(char* str, const int prime[])
{
    int value = 0;
    int count = 0;
    for (char* cur = str; *cur != '₩0'; cur++)
    {
        value += (count < 20) ? (*cur) * (prime[count]) : (*cur);
        count++;
    }
    return value;
    //prime 에 priNum1 또는 priNum2를 넣어 사용}
}
```

Dict 및 Word 자료구조 내부에서 사용되는 hash 는 다음과 같이 구성된다.

4.7 characters

The **average word length** in **English** language is 4.7 characters.

primeNum_: 2~71 사이 20 개의 소수를 무작위로 나열한 것. 엑셀을 이용해 무작위로 섞었다. 구글에 따르면, 영어 단어의 평균적인 길이는 4.7 알파벳 길이로 나타났다. 따라서, 해시 함수의 균일성을 추구하는 과정 중, 5 개 이상의 소수를 고려했는데, 결과적으로 20 개의 소수를 사용하도록 결정했다.
hash: 입력 받은 문자열에 대해 20 번째 자리까지는 주어진 소수와 값을 곱해 더하고, 이후 문자에 대해서는 그냥 folding 방식으로 더한다. 이후, 이 값을 돌려주게 된다. 코드 내에서 hash 를 직접 사용하는 대신 해시 테이블을 위한 hash1(mod TABLE_SIZE 연산 추가), 이진탐색 트리에서 중복도를 줄이기 위해 사용되는 hash2 로 변환되어 사용된다.

프로그램 내부에서 2 개의 소수 배열과 2 개의 hash 함수가 사용되는데, 특징적으로 hash2 및 priNum2 은 Word 이진탐색 트리에 사용된다. Word 자료구조 내에서 자료를 정렬하려면 특정한 값이 필요하다. 만약 단순히 문자열의 길이로 설정한다면, 높은 중복도가 나올 것이고, 기존 해시 함수를 사용하면, Dict 의 크기는 1000 이므로, 100 의 자리까지 동일한 값을

가지므로, 제대로 구분되지 않을 확률이 높다. 따라서, 추가적인 해시 함수를 구현해, 이러한 값들을 제대로 구분할 수 있도록 설계하였다.

[2] File 구조체 (file.h)

```
typedef struct _file
{
    int idx; // 파일 번호. 이후, 이 번호를 사용해서 파일 접근할 예정이다.
    int count; // 단어가 나타난 횟수
    struct _file* before;
    struct _file* next; // 뒷 노드
}File;
```

File 구조체는 단어가 저장되는 파일의 정보를 저장하는 양방향 연결 리스트 자료구조다.

int idx: docXXX.txt 입력 파일의 XXX, 즉 입력 파일을 나타내는 변수
int count: 연결된 단어가 몇 번 docXXX.txt. 파일에 나타났는지를 가리키는 변수

연관된 함수는 다음과 같다

```
File* fInit(int idx) { ... }

void fPrintSentence(File* input, char* curWord) { ... }

void fDeleteAll(File* input) { ... }
```

{1} fInit : idx = idx, count = 1 로 초기화 한다. 나머지는 일반 양방향 연결리스트와 같다.

{2} fPrintSentence : file 의 정보를 이용하여 cQueue 를 이용, 문자열을 출력하는 함수이다.

```
while (!feof(fs) && !cQFull())
{
    if (fscanf(fs, "%s", buffer) == 1)
    {
        if ((temp = getrawStr(buffer)) != NULL) cQadd(getrawStr(buffer));
    }
}
while (!feof(fs))
{
    if (count >= input->count) break;

    temp = strInit(cQueue[cur]);
    //if (temp == NULL) continue;
    if (temp != NULL)
    {
        if (strncmp(curWord, temp, BUF_SIZE) == 0)
        {
            cQprt(feof(fs));
        }
    }
    cQdel();
    if (fscanf(fs, "%s", buffer) == 1)
    {
        if ((temp = getrawStr(buffer)) != NULL) cQadd(getrawStr(buffer));
    }
}
while (!cQEmpty())
{
    temp = strInit(cQueue[cur]);
    if (temp != NULL)
    {
        if (count < input->count && strncmp(curWord, temp, BUF_SIZE) == 0)
        {
            cQprt(feof(fs));
        }
    }
    cQdel();
}
```

위 사진은 해당 함수의 일부분을 캡쳐 한 것이다. 첫번째 while 문을 통해, 우선 cQueue 가 가득 찰 때까지 단어를 입력 받는다. 기본적으로 검색 단계에서 단어는 제한요소 상에 최대 7 개의 단어를 출력하도록 정의하고 있다. 초반 단어가 입력되는 상황에 대해 단어 출력의 일관성 유지를 위해 큐를 가득 채우고 시작한다.

두번째 while 문은 단순히 문서 끝까지 단어를 입력 받고, cur 과 입력된 단어를 비교하여 문장을 출력하는 부분이다.

세번째 while 문은 문서가 끝난 상황이다. 더 이상 읽을 단어가 없으므로 해당 단계에서는 단어를 비교하여 문장을 출력하는 상황만 거친다.

fDelete 함수는 단순한 동적할당 해제관련 코드이므로, 설명을 생략한다.

[3] Word 구조체

```
typedef struct _word
{
    char* str;
    File* list;
    struct _word* lc;
    struct _word* rc;
}Word;
```

Word 구조체는 BST(이진탐색 트리)에 기반한 자료구조이다.

str: 현재 저장된 단어를 가리키는 변수. 동적할당 된 단어를 가리키게 된다.
list: 해당 단어가 저장되어 있는 파일을 가리키는 File 구조체.
lc, rc: 이진탐색 트리에서 child 를 가리키는 변수.

Word 는 기본적으로 hash2 함수의 값에 의해 내부 위치를 지정한다. 이때, 아무리 해시 함수를 거치더라도 같은 값이 나올 수 있다. 따라서, 기존 이진탐색 트리의 동작과는 다르게, left child 가 작거나 같은 값을 가리킨다. 해당 내용은 wSearch 함수 내에 반영되어 있다.

관련 함수는 다음과 같다.

```
Word* wInit(char* item) { ... }

void fInsert(Word* curWord, int fileIdx) { ... }
// ...
Word* wSearch(Word* root, char* str ,int* success) { ... }
//root에서 insert 여부 찾기 시작. str에 대해 wsearch를 이용
void wInsert(Word* root, char* str, int fileIdx) { ... }

void wPrint(Word* target) { ... }
//post order traverse 사용
void wDeleteAll(Word* root) { ... }
```

{1} wInit : 일반적인 tree 의 초기화 코드를 담고 있다.

```
temp->str = item;
temp->list = fInit(-1);// fi
temp->lc = NULL;
temp->rc = NULL;
```

코드의 편의성을 위해 fInit(-1), 즉 File 구조체에 대한 더미 노드를 사용하고 있다.

{2}fInsert: 현재 단어에 대해 File 이 생성되어 있는지 여부를 판단하고, 있으면 count 를 증가시키고 없으면 fInit 을 통해 생성하는 함수.

```
if (current->next->idx == fileIdx) //file을 발견
{
    current->next->count++;
    current = current->next;
    while (current->before->idx != -1) //더미노드가 앞
    {
        if (current->count > current->before->count)//
        {
            int tempIdx = current->idx;
            int tempcount = current->count;

            current->idx = current->before->idx;
            current->count = current->before->count;
            current->before->idx = tempIdx;
            current->before->count = tempcount;

            current = current->before;
        }
        else break;//자기자리 찾음.
    }
    return;
}
```

검색 단계에서 단어는 많이 등장한 문서 순으로 출력되어야 한다. 이 대목에서 File 구조체는 정렬되어야 한다(File 의 count 값). 이때, 경우의 수를 나눠 보자.

: 1 : fInit 을 통해 파일이 생성되는 경우

File 의 count 는 반드시 1 이고, 이 값은 반드시 최소 값이다. 따라서, 필연적으로 가장 작은 값이 되므로, 정렬할 필요가 없다.

: 2 : 이미 존재하여 count 가 존재하는 경우

값이 증가하였으므로, 정렬 과정이 필요하다.

위 코드는 :2: 의 상황을 가정한 코드이다. 원리는 삽입 정렬과 매우 유사하다. count 가 증가한 값이 맨 뒤에 삽입되었다고 "가정" 하고(뒤의 값에 대해서는 고려할 필요가 없다. 해당 File 구조체들의 count 는 이미 정렬된 상태이므로 반드시 현재 변경된 File 구조체보다 작은 count 값을 가진다) before 의 count 와 값을 비교하며, 현재 File 의 값이 더 크다면 자리를 바꾼다. 해당 과정을 통해 Word 내부의 File 구조체는 정렬된 상태를 항상 유지한다.

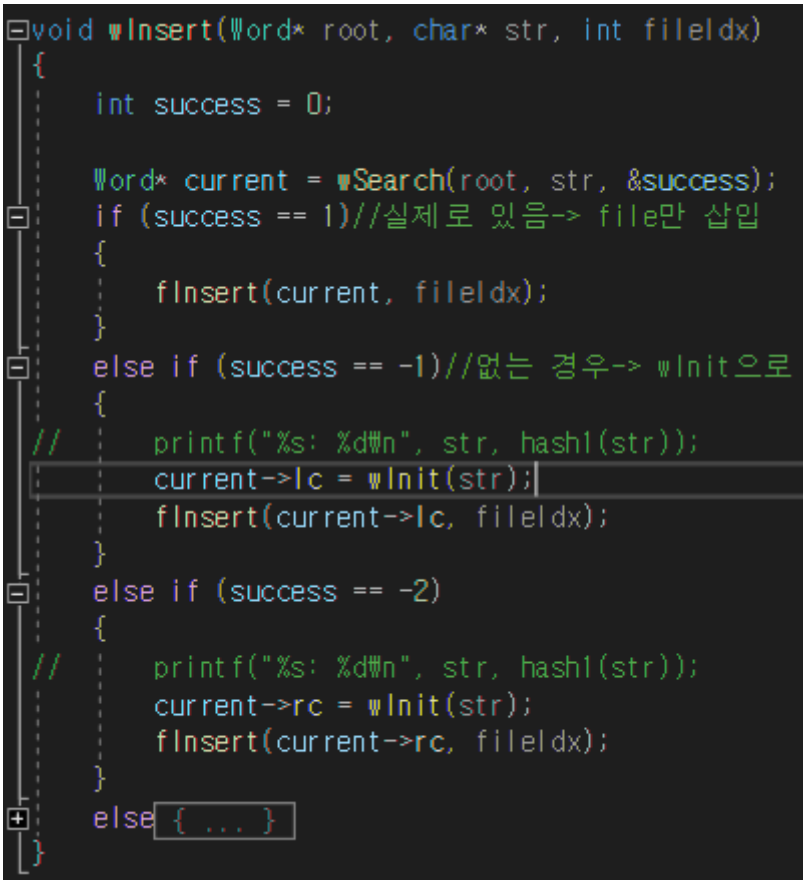
{3} wSearch: 원하는 Word 의 위치를 찾는 함수. 만약 해당 단어가 존재하지 않으면, 해당 단어를 삽입하기 위해 해당 단어의 부모 노드를 반환한다. 존재한다면, 해당 단어를 반환한다.

```
Word* wSearch(Word* root, char* str ,int* success)
{
    Word* current = root;
    Word* before = NULL;

    int hashValue = hash2(str);

    while (current != NULL)
    {
        before = current;

        int route = hash2(current->str) - hashValue;
        cmpMode ? cmp_insert_count++ : cmp_search_count++;

        if (route == 0) { ... }
        if (route <= 0) { ... }
        else { ... }
    }
    //cur == NULL이 나와버리는 상황
    return before;
}
```

위 함수의 입력 값에는 int *success 가 존재한다.
success 는 해당 탐색의 성공 여부를 포인터를 이용해 반환하는 변수이다. 해당 변수는 1, -1, -2 의 값을 가지게 되는데, 각각 성공, 좌측 삽입, 우측 삽입을 나타낸다. 탐색이 성공한 경우, success = 1, 발견한 Word 를 반환한다. 탐색에 실패한 경우 success = -1 or -2, 탐색을 실패한 위치에 대한 부모 노드를 반환한다. (실패는 NULL 인 경우이므로, 부모 노드와 삽입 위치를 알면, 쉽게 Word 를 삽입할 수 있다.)
기본적으로 Insert 와 Search 연산은 특정한 값을 찾는 과정을 동일하게 가지고 있다. 이때, 특정 값이 존재하는지 찾는 과정을 모두 wSearch 연산에게 맡기기 위해 위와 같은 구조를 가지게 되었다. 외부 코드에서는 success 값에 따라 다른 연산을 구현해야 한다.

값의 비교는 hash2 함수로 단어를 해시 값으로 변환하여 수행된다. 새로운 단어를 삽입할 때 최대한 중복을 줄이고, 트리의 균형을 맞추려는 목적으로 해시 함수를 사용하였다.

{4} wInsert: wSearch 결과를 토대로, 다른 행동을 취한다



```
void wInsert(Word* root, char* str, int fileIdx)
{
    int success = 0;

    Word* current = wSearch(root, str, &success);
    if (success == 1)//실제로 있음-> file만 삽입
    {
        fInsert(current, fileIdx);
    }
    else if (success == -1)//없는 경우-> wInit으로
    {
//      printf("%s: %d\n", str, hash1(str));
        current->lc = wInit(str);
        fInsert(current->lc, fileIdx);
    }
    else if (success == -2)
    {
//      printf("%s: %d\n", str, hash1(str));
        current->rc = wInit(str);
        fInsert(current->rc, fileIdx);
    }
    else { ... }
}
```

위에서 언급했듯이, wSearch 는 success 와 Word 두가지 값을 전달하는데, wInsert 는 탐색이 성공한 경우. 즉 단어가 이미 존재하는 경우, 단어가 속한 파일 정보만 전달하면 된다. 반면, 탐색이 실패한 경우. 예컨대 단어가 존재하지 않는 경우, 단어를 추가하고, 단어가 속한 파일 정보를 저장해야 한다.

다른 함수는 특별한 점이 없으므로, 언급하지 않는다.

[4] Dict 자료구조

```
Word* dict[TABLE_SIZE];
```

Dict 는 체이닝 해시 테이블 자료구조이다. 고정된 TABLE_SIZE = 1000 의 길이를 가지며, 사용된 자료구조의 핵심이다, 특정 값에 바로 접근할 수 있는 자료구조로, 위에서 언급한 hash1 함수로부터 해시 값을 얻어 값을 저장한다. 만약 해당 위치에 이미 값이 존재한다면, Word 자료구조에 대한 연산을 수행(hash2 함수를 이용하여 이진탐색 트리 내부에 삽입)하여 자료를 저장하게 된다.

함수는 다음과 같다.

```
⊞void dictInsert(char* str, int fileIdx)

⊞void deleteDictionary() { ... }

⊞void searchWord(char* str) { ... }

⊞void makeDictionary() { ... }
```

{1} dictInsert: Dict 자료구조에 입력된 단어를 저장한다. 만약 해당 버켓에 Word 가 없다면, Word 자료구조를 만들어 넣고, 존재한다면 단순히 wInsert 를 통해 단어를 저장한다.

```
void dictInsert(char* str, int fileIdx)
{
    int hashValue = hash1(str);
    if (dict[hashValue] == NULL) // 처음 값
    {
        dict[hashValue] = wInit(str); // wor
        fInsert(dict[hashValue], fileIdx);
        return;
    }
    //root 값이 존재하는 경우, 왼쪽 오른쪽
    wInsert(dict[hashValue], str, fileIdx);
}
```

{2}searchWord: 검색 단계에 사용되는 함수. 입력 받은 단어가 존재하는지 판단하고(wSearch 사용), 존재한다면 해당 값을 출력한다.

```
void searchWord(char* str)
{
    int hashValue = hash1(str);
    int success = 0;

    Word* myWord = wSearch(dict[hashValue], str, &success);
    if (success == 1)
    {
        wPrint(myWord);
    }
}
```

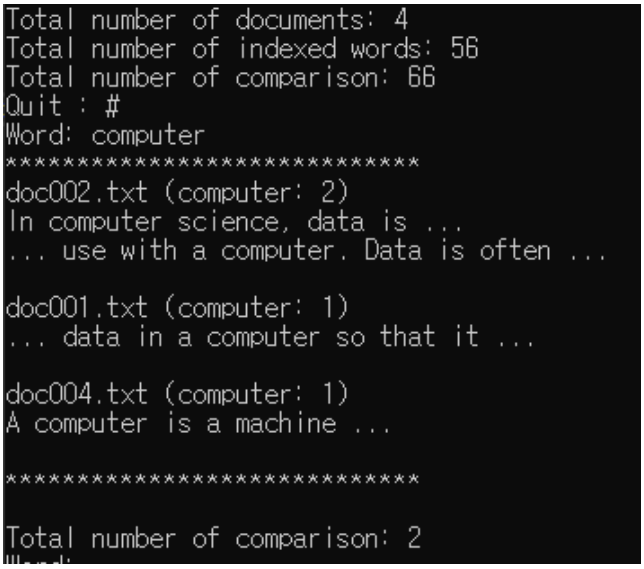
{3}makeDictionary: 색인 단계에 사용되는 함수. 입력 파일에 해당하는 filename 을 이용하여 해당 파일이 존재하는지 판단하고, 존재한다면 해당 파일 내부에서 단어를 차례대로 읽어 dictInsert 함수를 통해 색인화 한다.

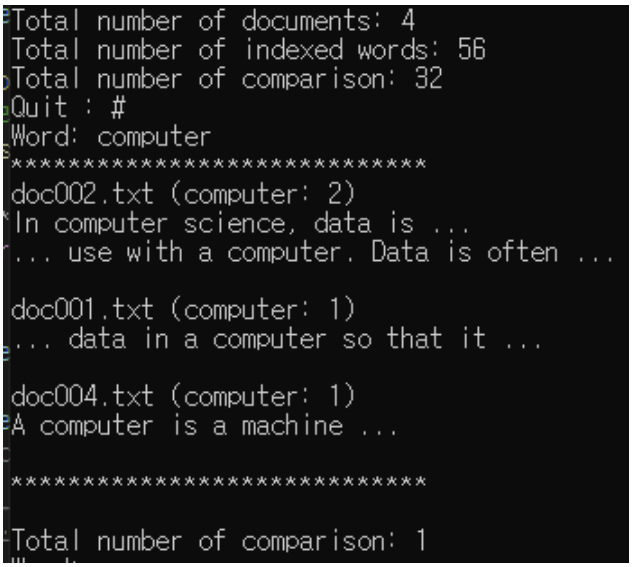
```
void makeDictionary()
{
    FILE* fs;
    char buffer[BUF_SIZE];
    for (int idx = 1; idx <= 100; idx++)
    {
        setFileName(idx);
        if ((fs = fopen(filename, "r"))) //해당 이름의
        {
            file_count++;//읽은 파일 개수 증가
            while (!feof(fs))
            {
                if (fscanf(fs, "%s", buffer) == 1)
                {
                    dictInsert(strInit(buffer), idx);
                }
            }
            fclose(fs);
        }
    }
}
```

3.결과 및 토의

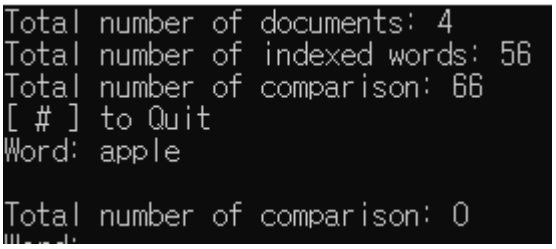
3.1 프로그램 테스트 결과

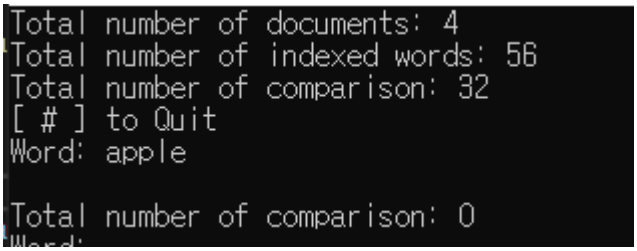
프로젝트 설계 지침서 예시 부분에 나타나 있는 doc001.txt ~ doc.004.txt 을 txt 파일로 저장하여 현재 프로그램이 있는 문서로 옮긴다. 이후 DictionaryProject.exe 파일을 실행 후, 문서 내에 존재하는 단어(computer) 및 존재하지 않는 단어(apple)를 검색한다.
현재 구현된 프로그램은 Word 자료구조 내에서 문자열을 직접 비교하는 대신, 해시로 바꾸어 비교하고 있다(wSearch). 따라서, 비교연산이라도 문자열에 대한 비교연산 회수는 상당히 적은 값으로 나타나게 된다. 이를 고려하여, 해시로 바꾼 값에 대해서도 비교 연산에 추가하는 경우를 왼쪽에, 추가하지 않은 경우를 오른쪽에 표시하도록 하겠다.





(computer 입력)





(apple 입력)

파일은 요구조건에 맞게(문자열 비교부분만 비교연산에 포함) 오른쪽 상태로 제출된다.

3.2 수행 결과에 대한 토의

wSearch 내부 81 번 코드를 주석처리 하든 주석처리 하지 않든, 항상 비교 대상에 해당하는 설계 지침서 내 예시의 경우보다 낮은 문자열 대상 비교 연산횟수를 보였다. 이러한 점은 기본적으로 해시 테이블 기반이므로 성능면에서 효율적인 것이라고 생각한다. 다만, 현재 비교 대상에 한정한다면 굳이 Word 를 이진탐색 트리로 구성할 필요까지는 없는 것 같다. 비교 연산이 대부분 1,2 회 정도에 머무르기 때문이다. 사실 이러한 이유는 입력으로 들어오는 파일이 4 개뿐이기 때문으로 보인다. 입력되는 단어 자체의 수가 테이블의 크기에 비해 매우 적으므로, 이진탐색 트리까지 적용하는 것은 과한 방향일지도 모르겠다.

Word 의 자료 삽입 방식에 대해서는 아쉬움이 남는다. 내가 구현한 hash 함수는 20 자리 이하의 알파벳 문자에 대해서 각 자리에 소수를 곱하고 더하는 연산을 반복하므로 꽤 복잡한 연산을 한다고 볼 수 있다. 이런 관점에서, Word 자료구조 내 자료의 삽입 및 삭제마다

hash2 함수를 사용하므로, 비교회수가 많아질수록 성능적 저하가 존재할 것이다. 이런 부분의 경우, 트리의 불균형을 잡는 다른 방식을 사용하는 것도 좋을 것 같다.

확장성에 대해 생각해보자.

자료구조를 구현할 때 버퍼의 크기를 100 으로 설정했는데, 이는 대부분의 영미 사전에 등재된 가장 긴 단어로 알려진 pneumonoultramicroscopicsilicovolcanoconiosis : 45 자의 2 배에 해당하므로, 비정상적인 문자가 아닌 이상 모든 문자를 받을 수 있다.

입력 받는 파일에 대해서는 확장성이 상당히 떨어진다. 입력 받는 파일이 doc001.txt ~ doc100.txt 라는 제약을 가정하고 이에 딱 맞게 관련 함수들을 구현하였고, 따라서 File 내부에서 문자열 대신 숫자를 저장하는 방식으로 최적화할 수 있었다. 입력되는 파일의 인덱스 역시 0 ~ 999 까지만 인식된다. 이러한 부분은 확장성 부분에서 감점 요인이 된다.

반면, 자료구조 자체는 조건 내에서 확장성이 높다. 전체적으로 동적할당을 이용하여 만들어져 있고, Dict 자체가 체이닝 해시 테이블로 구성되어 있으며, 테이블이 가득 차는 상황에 대비하여 체이닝 부분을 Word : 이진탐색 트리로 구성하여, 버켓 내부 자료에 대해서 O(log n)의 평균적인 탐색 속도를 보이므로 자료가 많은 상황에서도 큰 속도의 저하 없이 작동할 것이다.

#추가

개인적으로 사용한 doc005.txt 를 동봉한다. 주로 디버깅 목적으로 사용하였으며, 직접 디버깅한 범위 내에서는 작동을 확인했다.