HW7 REPORT

학번: B911026

이름 : 김민욱

1번 문제 해설

1번 문제를 해결하기 위해선 다음과 같은 기능을 하는 함수들과 전역변수들이 필요하다.

1. 전역 변수 목록

(1) "words.dat" 파일에 있는 단어들을 저장할 char 자료형 2차원 배열

(2) Hash Table에 이미 값이 존재하는지 확인할 bool 자료형의 1차원 배열

이 두 변수를 다음과 같이 초기화한다.

```
char words[5757][5];
// 2-D array for store words
bool hashingCheck[HASH_PRIME] = {0,};
// check whether hash table is already filled or not
```

2. 함수 목록

(1) "words.dat" 파일에서 단어들을 불러와 배열에 집어넣는 함수

(2) 임의의 단어를 Hash값으로 변환하는 함수

(3) Hash값을 Hash Table에 넣을 때 충돌 횟수를 Count하는 함수.

(1)번 함수 설명

```
void putWordsInArray() {
  FILE *fp = fopen("words.dat", "r");
  char line[100];

  for(int i=0; i<4;i++){
    fgets(line, sizeof(line), fp); // ignore first 4 lines.
  }

  for(int i=0; i<NUMBER_OF_WORDS;i++){
    fgets(line, sizeof(line), fp);
    strncpy(words[i],line,5); // put first 5 characters in words array
  }
}
```

"words.dat" 파일은 최초 4줄엔 불필요한 문장이 들어있으며 그 다음 줄부터 작성된 word들도 첫 번째 5글자를 제외하고는 불필요한 값들이 들어있는 경우가 있다.

따라서 이를 해결하기 위해 4회 반복하는 for문으로 최초 4줄을 무시하고 이후 word의 개수(5757개)만큼 반복하는 for문을 활용해 각 줄을 임시변수 line에 읽어오고 strncpy 함수를 사용해 최초 5글자를 위에서 전역변수로 선언한 words 배열에 집어넣는다.

(2)번 함수 설명

```
int hash(char key[5]) {
  int i;
  long long x;
  x = 0;

  for (i = 0; i < 4; i++) {
    x = x + key[i];
    x = x << 8;
  }

  x = x + key[4];
  return x % HASH_PRIME;
}
```

위 함수는 char 자료형 5글자를 long long형 정수로 바꾸는 역할을 한다. 이를 구현하기 위해 한 글자를 x에 더하고 char 자료형의 크기인 8bit(1byte) 만큼 left bit shifting 연산하여 하위 8bit을 0으로 만드는 과정을 반복한다.

이후 정수로 바꾼 key값을 미리 지정한 Hash Prime으로 Modulo 연산을 하여 Hash 값을 리턴한다.

(3)번 함수 설명

```
int countCollisions(){
  int hashValue;
  int numberOfCollisions = 0;

  for(int i =0; i < NUMBER_OF_WORDS; i++){
    hashValue = hash(words[i]);

    if(hashingCheck[hashValue] == true){
      numberOfCollisions++;
    } else {
      hashingCheck[hashValue] = true;
    }

  }
  printf("%d",numberOfCollisions);
}
```

충돌은 같은 Hash값이 두 개 이상 존재할 때 일어난다. 따라서 충돌 횟수를 count하려면 모든 word에 대해 (2)번 함수를 호출했을 때의 return값이 이전에 이미 계산됐던, 즉 hash 값이 중복된 경우의 수를 count해야한다.

이를 구현하기 위해 위에서 선언한 전역변수 배열 hashingCheck를 사용한다.

어떤 word의 hash값이 n이고 hashingCheck[n]의 값이 **false** 라면 n은 최초로 계산된 hash 값이므로 hashingCheck[n]의 값을 **true**로 바꾼다. 만약 hashingCheck[n]의 값이 **true**라면 hash 값이 중복 즉, 충돌이 일어난 것이므로 numberOfCollisions의 값을 1 증가시킨다.

결과

(1) 각 Hash Prime에 대한 충돌 횟수

Composite number	Prime number	Ratio
M1 : 2305	M2 : 1821	1.26
M3 : 2733	M4 : 1235	2.21
M5 : 1235	M5 : 655	1.88

(2) 분석

비슷한 크기의 숫자이더라도 **소수가 아닌 수**는 **소수**에 비해 충돌이 약 1.2 ~ 2.2배 많이 일어났다. 하지만 Hash Prime의 크기와 이 차이는 **비례하지 않는 것**으로 보인다.

소수가 아닌 수는 Hash prime의 크기와 충돌 횟수가 관계없는 것으로 보이나, **소수**는 Hash prime의 값이 커질수록 충돌 횟수가 적어지는 것으로 측정됐다. 따라서 modulo 연산을 사용한 Hash 함수의 성능을 높이기 위해선 작성할 때는 메모리, 하드웨어 등 여러 요소를 고려해 **가능한 큰 소수**을 Hash prime으로 사용해야 할 것으로 보인다.

2번 문제 해설

1번 문제에서 사용한 전역변수와 함수들을 활용한다.

1. 전역 변수 목록

(1) "words.dat" 파일에 있는 단어들을 저장할 char 자료형 2차원 배열

(2) Hash Table에서 각 chain에 존재하는 node의 수를 저장하는 배열

이 두 변수를 다음과 같이 초기화한다.

```
char words[5757][5];
// 2-D array for store words
int numberOfNodesInChain[HASH_PRIME] = {0,};
```

2. 함수 목록

(1) Hash Table의 각 chain의 노드의 개수를 구하는 함수

(2) key comparison을 counting 하는 함수

(3) 1번 문제의 (1),(2)번 함수

(1)번 함수 설명

```
int makeHashTable(){
  int hashValue;
  int numberOfCollisions = 0;

  for(int i =0; i < NUMBER_OF_WORDS; i++){

    hashValue = hash(words[i]);

    numberOfNodesInChain[hashValue]++;

  }
}
```

Hash table에서 chaining 방식을 사용할 때, 같은 hash 값을 가지는 node들은 linked list로 저장된다. 하지만 2번 문제에서는 key comparison을 counting하는 것이 목적이므로 우리는 **각 linked list의 길이만**을 구한다.

이를 구현하기 위해 hash prime만큼의 길이를 가지는 전역변수 int형 배열을 선언하고 hash 값이 발생할 때마다 해당 값을 index로 가지는 배열의 element의 값을 1씩 증가시킨다.

(2)번 함수 설명

```
int countKeyComparisons(){
  int totalKeyComparisons = 0;

  for(int i = 0; i < HASH_PRIME; i++){
    int keyComparisonsInChain = 0;

    int n = numberOfNodesInChain[i];
    keyComparisonsInChain += (n*(n+1))/2;

    totalKeyComparisons += keyComparisonsInChain;
  }
  return totalKeyComparisons;
}
```

n개의 node를 가지는 linked list에서 **모든 node를 검색**하려면 **1+2+….+(n-1)+n =** $\frac{n*(n+1)}{2}$ 번 만큼의 key comparison이 필요하다.

따라서 Hash table에서 모든 key값을 검색하려면 numberOfNodesInChain **배열의 모든 요소**에 대해 $\frac{n*(n+1)}{2}$ 연산을 실행하고 결과값을 더하면 총 key comparison 횟수를 구할 수 있다.

결과

M4 = 11117 을 hash prime으로 사용했을 때 key comparison은 총 **7216회** 일어난다.