Assignment11 Document

#자기정보

전공	소프트웨어전공	학번	2015004011
이름	김도현	개발환경	Xcode

#함수소개

1. BINARY_DECODING();
   - Output.txt에서 문자를 하나씩 가져와 마찬가지로 2진수의 표현범위만큼 읽어서, 여기서도 각 자리에 맞는 2의 제곱수를 더해줘 원래의 decode string을 만들어준다. 각 자리에 맞는 2의 제곱수를 더해줘야 10진수가 되므로 중요한 부분. 그리고 decode string이 완성되면 그것을 토대로, 다시 STRUCT_INITIAL_TABLE(RESTRUCT_TABLE)을 호출하여 재 구축 TABLE을 형성한다. 초기 사전에는 크기가 1인 문자들만 생성 되어있으므로, 하나씩 사전에 추가하면서, 해독한다. 해독 방법은 숫자에 맞는 code를 가진 key를 recoverdinput.txt에 넣고, 이전에 읽은 문자열에 이번에 읽은 문자열의 맨 처음문자를 붙여서 사전에 넣어주면서 사전을 구축한다. 여기서 code를 가진 NODE를 발견하지 못했다면, 이전에 읽은 문자열에다가 이전에 읽은 문자열의 맨 첫 문자를 더하여 같은 방식으로 NODE를 생성하여 linked list로 이어준다. 또, 첫 시행에는 이전 문자열이 없으므로, 추가하지 않고 넘어간다. 그렇게 하면 사전도 원래 모양대로 구축되고, recoveredinput.txt에도 원래의 input.txt와 같은 내용이 복구되게 된다.
2. BINARY_COMPRESS();
   - 이제 compress string에 압축 된 것을 토대로 bit string을 만드는데, 사전의 크기에 따라 이진수의 표현 크기를 정한다. Ex) 크기 9 이면 이진수 표현범위 4, 크기 33이면 표현범위 6. 정하고 나면 /, %를 이용하여 이진수로 변환 시키는데 역순으로 저장하는 부분이 중요하고, 표현범위보다 짧은 문자는 앞에 0을 채워주는 부분이 필요하다. 그렇게 해서 모든 compress string의 숫자를 bit string으로 변경해주고, Huffman에서 사용한 bit 압축을 이용하여, shift를 사용하여 output.txt에 압축시킨다.
3. COMPRESS();
   - LZW압축의 핵심이 되는 부분인데, 과정은 먼저 문자열을 앞에서부터 뒤로 읽으면서, 사전에 포함되지 않는 문자열이 될 때까지 길이를 늘려가며 읽는다. 사전에 포함되지 않은 것이 있으면, 그 문자열을 사전에 새로 NODE를 생성하여 linked list로 연결시켜주고, 맨 마지막 문자 제제외한 것 사전에 포함되어 있었을 것이므로, 그 포함 되어 있던 문자열의 code를 입력해준다. Code가 모인 것이 나중에 compress string이 된다. 만약 추가시켜주는데 마지막이라면 맨 마지막 문자도 제외하지 말고 compress string에 추가시켜준다. 또한, 사전에 포함되어 있어서 추가가 안되고 길이를 늘리다가 string이 끝나면, 그 부분도 동일하게 없애지 말고, compress에 넣어 완전히 압축한다.

4. STRUCT_INITIAL_TABLE(link* TABLE, int* TABLE_SIZE);
    - 가장 처음에 실행되는 함수로, string을 읽어 들여 한 문자씩 비교하면서 빈도를 측정한 다음, NODE를 생성하여 그 문자를 각각 넣어주는데 여기서 문자도 크기가 차이가 나므로 동적 할당을 해준다. 또한, 각 NODE마다 code를 붙여 순서를 확인하고, Linked list로 연결하여 사용하고 TABLE_SIZE를 정하여 정보를 제공하는데 이용된다.
5. SUB_STRING(char* copy, char* string, int start, int end);
    - copy라는 문자열에 string의 index를 start부터 end까지 설정하여 그 부분을 copy에 입력하고 마지막 자리에 NULL을 넣어 주어 문자의 배열과 string을 구분시켜준다. 계속 덮어쓰기 가능한 이유이기도 하다.
6. FREE(link* TABLE);
    - 할당한 메모리는 마지막에 해제를 시켜줘야 하므로, 총 2가지 사전이 생성되게 되는데, 그 HEADER NODE를 parameter로 넣어주면, 각 NODE의 key도 동적 할당 하였으므로, 해제시켜준 뒤, 앞에서부터 뒤로 가면서 모든 NODE 해제;

#압축률

LZW압축 = input.txt크기 21byte, output.txt크기 5byte = 5 / 21 = 약23.8%;

Huffman = input.txt크기 21byte, output.txt크기 3byte = 3 / 21 = 약14.28%;

Huffman이 압축률이 더 큰 것으로 나타났습니다. 제 생각엔 소문자만 사용하고, 공백도 제외한 LZW압축이라 차이가 나는 것 같습니다. (Huffman은 공백, 대문자, 쉼표, 마침표 포함);

그리고 Huffman은 빈도와 사용문자개수에 따라 차이 나는데 예제는 a, b만 사용했으므로 Huffman이 압축률이 더 크다고 생각합니다.

#실행화면

```
Enter the string : abababbabaabbabbaabba

< 초기 사전 상태 >
HASH_TABLE[0] key = [a] code = 0
HASH_TABLE[1] key = [b] code = 1
TABLE_SIZE = 2

< 압축을 통해 완성된 사전 >
HASH_TABLE[0]'s key = a code = 0
HASH_TABLE[1]'s key = b code = 1
HASH_TABLE[2]'s key = ab code = 2
HASH_TABLE[3]'s key = ba code = 3
HASH_TABLE[4]'s key = aba code = 4
HASH_TABLE[5]'s key = abb code = 5
HASH_TABLE[6]'s key = bab code = 6
HASH_TABLE[7]'s key = baa code = 7
HASH_TABLE[8]'s key = abba code = 8
HASH_TABLE[9]'s key = abbaa code = 9

compress_string = 012233588
bit_string = 00000001001000100011001101011000100 0

'output.txt'에 압축!!!
'output.txt'에서 읽어와 압축풀기!!!

decode_bit_string = 00000001001000100011001101011000100 0
decode_string = 012233588

< 초기 사전 상태 >
HASH_TABLE[0] key = [a] code = 0
HASH_TABLE[1] key = [b] code = 1
TABLE_SIZE = 2

< 재구축된 사전 >
RESTRUCT_TABLE[0]'s key = a code = 0
RESTRUCT_TABLE[1]'s key = b code = 1
RESTRUCT_TABLE[2]'s key = ab code = 2
RESTRUCT_TABLE[3]'s key = ba code = 3
RESTRUCT_TABLE[4]'s key = aba code = 4
RESTRUCT_TABLE[5]'s key = abb code = 5
RESTRUCT_TABLE[6]'s key = bab code = 6
RESTRUCT_TABLE[7]'s key = baa code = 7
RESTRUCT_TABLE[8]'s key = abba code = 8
RESTRUCT_TABLE[9]'s key = abbaa code = 9

--------------------------------
```