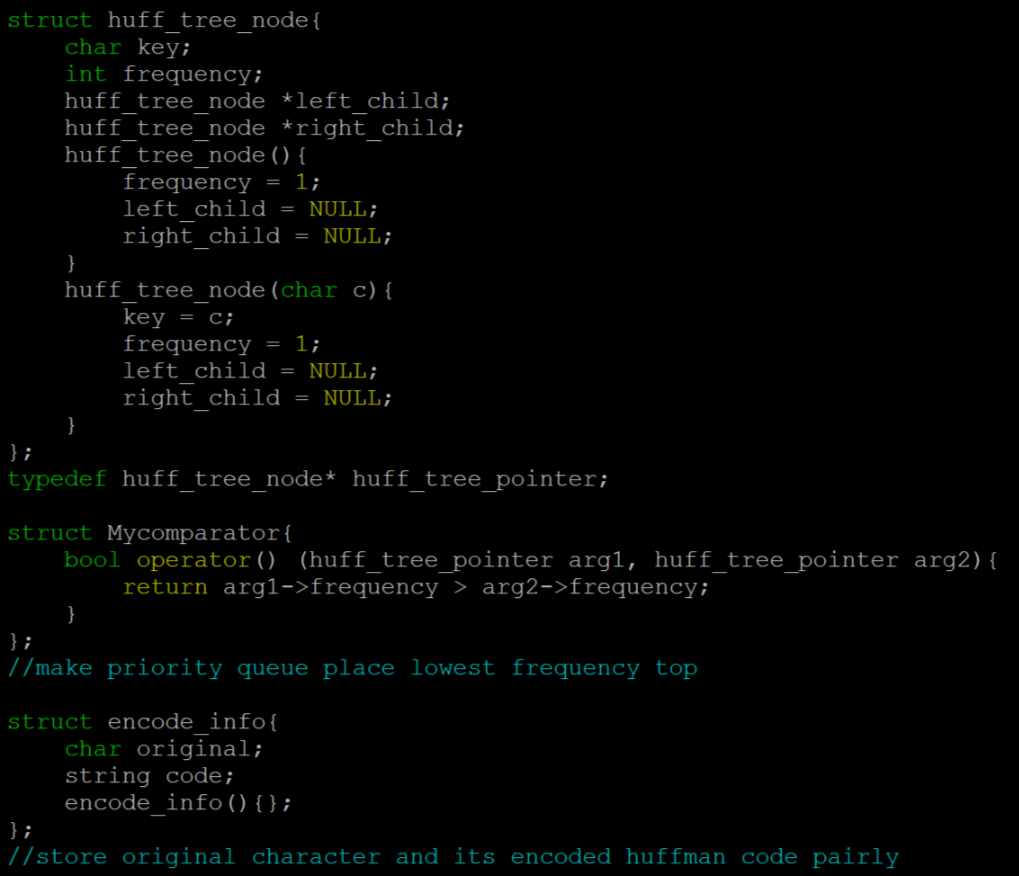
CSE3081-1 Machine Problem 3 Master of Compression

2학년 컴퓨터 공학과 20171697 최민영

<Description of Algorithm>

**사용한 구조체)**

huff\_tree\_node:

원래문자 = key

빈도수 = frequency

tree 왼쪽 자식 = left\_child

tree 오른쪽 자식 =

right\_child

encode\_info:

원래문자 = original

허프만코드 = code

Encode시 사용했던 자료구조, 자료형:

빈도수와 문자를 같이 저장하기 위한 huff\_tree\_node의 포인터 배열( vector이용 )

Huffman algorithm에서 가장 적은 빈도수를 갖는 문자를 알기 위해, 트리의 노드 포인터로 구성된 priority queue. Mycomparator을 이용해 트리 노드의 포인터가 가리키고 있는 빈도수가 작은 순으로 위에 있도록 하였다.

Huffman code를 위한 트리를 왼쪽을 우선으로 살피는 DFS 방식으로 살펴보기 위한 stack. 이 stack은 봐야할 트리 노드 포인터와 이 노드 포인터와 매칭되는 huffman code를 쌍으로 저장하도록 하였다. ( stack< pair < huff\_tree\_pointer, string > > )

encode하여 파일로 출력한 문자들을 저장하고 있는 vector< unsigned char> 이용

Decode시 사용했던 자료구조, 자료형:

encode와 같이 트리를 왼쪽을 우선으로 살피는 DFS식으로 살펴보기 위한 stack사용.

원래 문자를 저장하는 vector<char> 사용

읽어들인 문자를 이진수로 저장하기 위한 int [8] 사용

**Compressed file 구성)**

(원래 문자 종류 개수) (원래 문자들) (트리 구조를 Encode한 문자 길이와 나머지) (트리 구조 Encode한 문자들) (원래 파일을 Encode한 문자길이와 나머지) (원래 파일을 Encode한 문자들)

이렇게 구성되어있다.

1. 원래 문자 종류 개수: 다음으로 올 원래 문자들을 길이에 맞게 읽기 위해 출력함
2. 원래 문자들: Huffman Tree를 왼쪽 자식 우선으로 DFS식으로 방문한 순서에 따른 원래문자들
3. 트리 구조를 Encode한 문자 길이와 나머지: 다음으로 올 트리 구조 Encode 문자들을 길이에 맞게 읽기 위해 문자 길이를 출력하고 마지막 문자가 꼭 8자리 2진수를 ASCII코드로 바꾼 게 아니라 1~7자리 2진수를 ASCII코드로 바꾼 것일 수 있으므로 나머지를 출력
4. 트리 구조 Encode한 문자들: Huffman Tree를 왼쪽 자식 우선으로 DFS식으로 방문하면서 방문한 노드가 자식을 가지면 1과, 자식을 가지지 않으면 0과 대응되게하여 트리 구조를 0과 1로 표현한 문자들이다. ( 이를 compress한 표현으로 밑에 ‘과정 속 문자 출력 방식’을 사용하였다. )
5. 원래 파일을 Encode한 문자길이와 나머지: 3번과 같은 의미로 쓰였다.
6. 원래 파일을 Encode한 문자들: 입력파일 하나의 문자마다 Huffman코드로 변형시키고 압축하여 표현하기 위해 밑에 ‘과정 속 문자 출력 방식’을 사용하였다.

**Encode한 순서)**

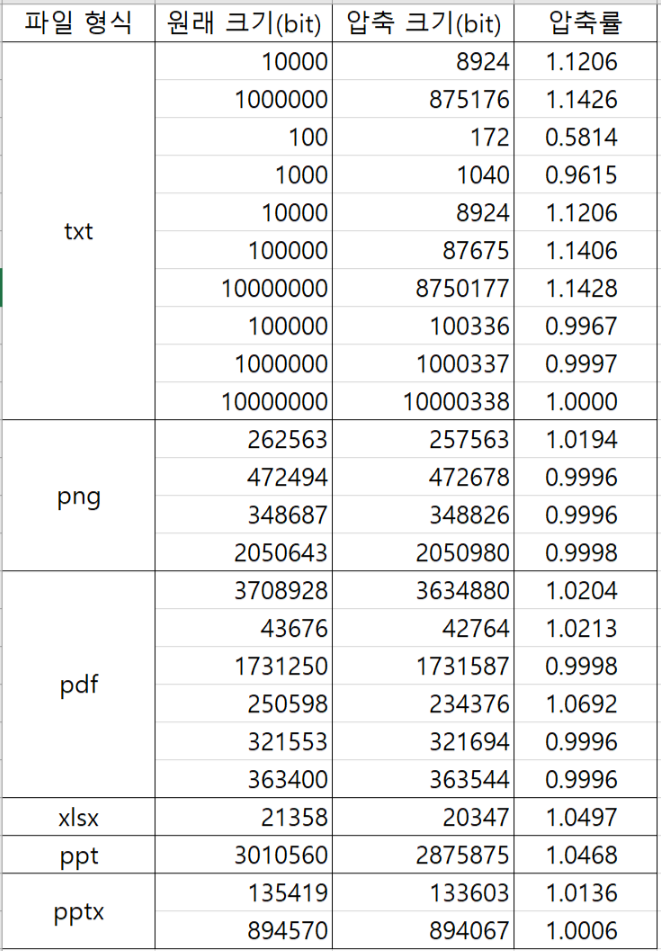
1. Count\_frequency 함수: 문자마다의 쓰인 횟수를 저장한다.
   1. 만약 아무것도 들어있지 않거나 한 문자로만 이루어져있다면 예외적인 압축파일을 만들도록 하였다.
   2. Make\_huffman\_tree 함수: huffman algorithm을 이용하여 tree를 만든다
2. Encode\_Tree\_description 함수: 위에서 설명한 방식으로 트리의 leaf에 저장되어있는 원래 문자들을 출력한다. 그리고 나서 트리구조를 encode하여 출력한다. 그리고 이러한 과정 중에 원래 문자에 해당하는 Huffman code를 구해 매칭시켜 배열에 저장한다.
3. Delete\_Tree 함수: 만들었던 트리 구조를 free시킨다
4. Encode\_File 함수: 입력된 파일을 한 문자씩 읽으면서 해당문자에 매칭되는 Huffman code를 출력파일로 출력한다.

**과정 속 문자 출력 방식)**

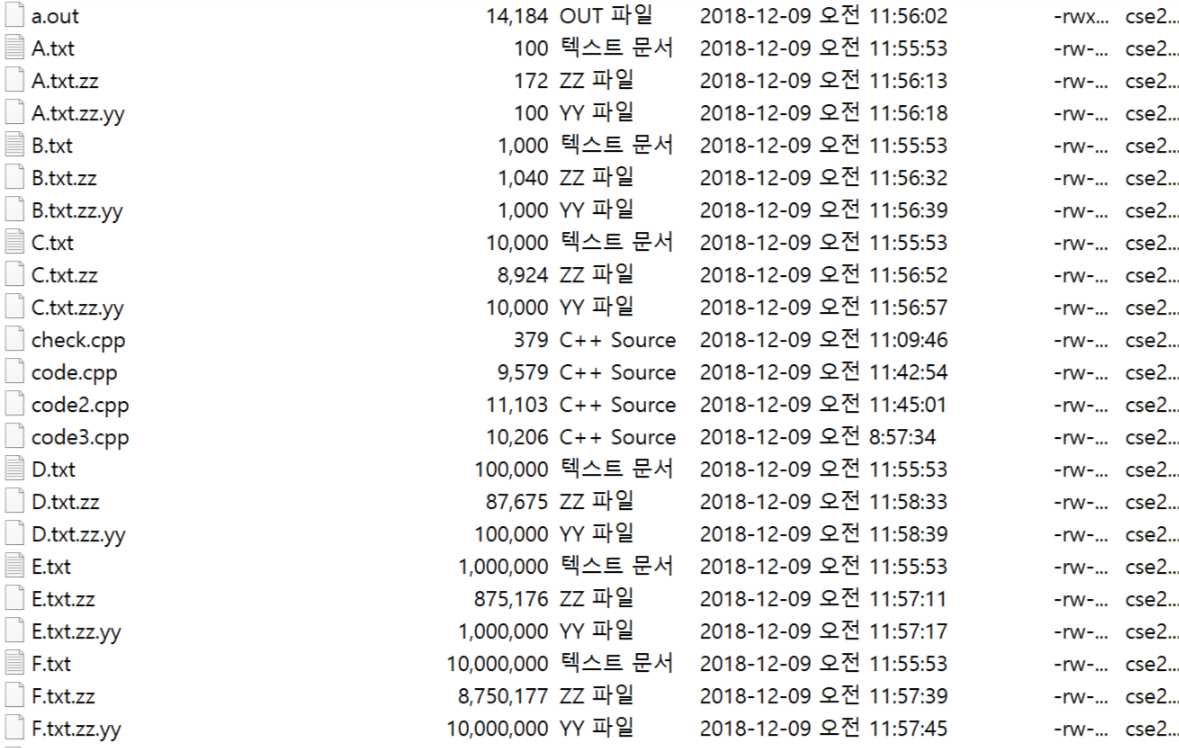
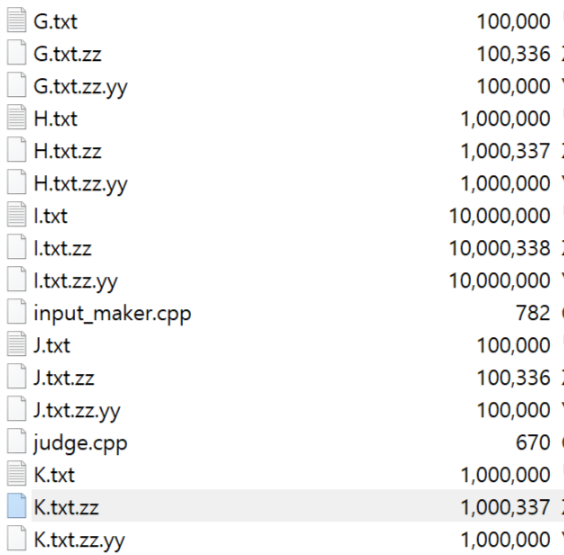
실제로는 내가 만드는 encode 문자들은 0과 1로 구성되어있으나 이를 Make\_encode\_list 함수를 이용해서 0과 1을 buffer에 들어오는 순서대로 쌓아두었다가 길이가 8이 되면 buffer를 8자리의 2진수로 생각해 이에 대응하는 ASCII 코드 character로 바꿔 이를 출력하였다. 예를 들어 buffer에 01000001이 저장되어있다면 ‘A’를 출력하도록 하였다. 그래서 decode할 때는 하나의 character을 읽어 이를 unsigned int로 바꿔서 생각하고 이에 해당하는 이진수를 int [8] 배열에 저장하여 해석하도록 하였다. 위의 예를 똑같이 사용하면 ‘A’를 읽어들이면 int binary[8] 배열에 0, 1, 0, 0, 0, 0, 0, 1 이렇게 저장되도록 하였다. 이러한 역할을 하는 것이 Convert\_to\_binary 함수이다.

**Decode한 순서)**

1. 파일을 읽었을 때 비어있는 파일이거나 문자가 하나만 쓰인 경우라고 판단되면, 예외적으로 decode하였다.
2. 원래문자들을 트리를 방문한 순서로 출력해놓은 것을 그대로 읽어 저장한다.
3. 트리 구조 encode 문자들을 읽으면서 해당하는 트리를 만들고 트리의 Leaf를 방문하였음을 판단하면 2에서 저장했던 원래문자를 트리 노드에 넣는다. 이 때 트리의 Leaf에 방문하는 순서대로 2에서 저장한 문자열 앞에서부터 매칭시킨다. 이렇게 huffman tree를 만들어낸다.
4. 원래파일을 huffman code로 구성해놓은 compress file을 읽으면서 0이면 현재 왼쪽 자식 노드로, 1이면 현재 오른쪽 자식 노드로 움직인다. 그러다가 leaf node에 도달하면 이 노드에 저장된 원래 문자를 출력한다. 그리고는 트리의 root로 올라간다.

<Run of program>

파일형식에 따라 큰 차이를 보이진 않았으나 png파일은 대부분 압축이 되지 않았다. pdf 경우 딱 절반은 압축이 되고 절반은 되지 않았다. xlsx와 ppt에서는 압축이 다른 거에 비해 잘 되었던 편이고, pptx 형식에서는 원래 크기와 정말 가깝게 압축이 되었다.



오른쪽은 128가지의 문자만으로 구성된 텍스트파일을 압축했을 때의 결과이고 왼쪽은 256가지의 문자로 구성된 텍스트파일을 압축한 결과이다. 그 결과 오른쪽은 잘 압축되었으나 왼쪽은 그렇지 않았다. 허프만 코드가 빈도수가 적은 순으로 긴 huffman code를 부여하는 방식인데 문자마다 빈도수 차이가 크지 않으면 허프만 알고리즘이 효과적인 압축을 실행시키진 못함을 예상할 수 있다. 또한 빈도수 차이가 크지 않더라도 종류가 많으면 prefix가 겹치지 않게 부여할 huffman 코드가 길어져 더욱더 효과적이지 못할 것이다. 위의 결과는 이 예상과 맞아떨어지는 결과로써 보인다.