**Huffman Coding for File Compressiong**

20161566 권형준

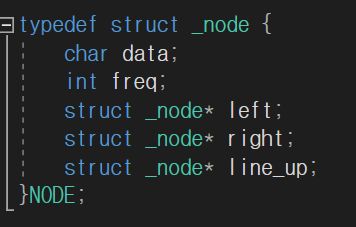
1. **실험 환경**

|  |  |
| --- | --- |
| CPU | Intel Core i5-8265U |
| CPU 속도 | 1.80 GHz |
| 메모리 용량 | 8.00 GB |
| 시스템 종류 | 64비트 운영체제, x64기반 프로세서 |
| OS 종류 | Windows 10 |

1. **실험 구조**

실험에 사용된 자료구조와 알고리즘을 설명하기 위해서 코드가 진행되는 순서대로 설명하겠습니다.

우선 **compressing** 파트부터 시작하겠습니다. 처음 파일을 열어 그 안에 있는 모든 값들을 char변수형으로 입력을 받습니다. 이때 story에 값들을 저장하는데 동적할당을 사용하였으며, 처음부터 얼마나 많은 양의 변수를 입력 받을지 모르기 때문에 realloc을 사용하여 계속 동적할당을 해주었습니다. 다음 이 값들을 하나의 기차처럼 연결을 하여 어떤 character가 몇 번 나타나는지 살펴보았습니다. 여기서 NODE를 처음 사용하는데 구조는 다음과 같습니다.



처음에 아직 left, right에 연결하기 전에 기차처럼 연결하기 위해서 line\_up이라는 포인터를 사용하였습니다. 따라서 대략적으로 나타내면, 각 노드들을 가로로 line\_up으로 연결을 하고 left와 right에는 NULL값을 넣어주었습니다. 만약 넣어주려는 character의 값이 들어있는 노드가 없다면 끝에 새로운 노드를 **Add\_Child\_for\_line\_up**을 사용해서추가해서 character을 넣어주고, 만약 있다면 freq의 값에 +1을 해주어 빈도수를 높였습니다.

이 과정이 끝난 이후 허프만 코딩을 하기 위해서 트리 형태로 만들어주어야 했습니다. 우선 가장 작은 빈도수를 가진 노드 두개를 합쳐서 하나의 노드의 left, right 자식으로 설정하고 그 노드를 다시 저 기차에 끼워 넣는 과정을 반복하려고 했습니다. 그러기 위해서 우선 **sort\_nodes()**함수를 사용하였으며 가장 앞에서부터 빈도수가 작은 노드들이 오게 설정하였습니다. 그 두 노드들을 뽑아서 새로운 노드의 자식으로 넣고 다시 기차에 넣어서 이 과정을 기차에 오직 한 노드가 남을 때까지 반복하였습니다.

다음은 **DFS(temp,story[i],temp\_string,temp\_string\_length)**함수를 사용하여 노드의 머리에서부터 story[i]의 character값을 담고 있는 노드를 찾았습니다. 이때 그냥 찾은 것이 아니라 찾는 길에 재귀함수를 호출하였으며, 재귀함수를 호출할 때마다 temp->left, 즉 left 자식 노드로 들어가서 탐사를 하면 temp\_string에 1을, temp->right, 즉 right 자식 노드로 들어가서 탐사를 하면 temp\_string에 0을 넣어주었습니다. 이러한 이유는, 찾으려고 한 character값을 찾았을 때 그 temp\_string에 저장된 값이 바로 그 character의 이진 코드, prefix 코드이기 때문입니다. Story에 저장된 character의 값들 순서대로 result배열에 이 prefix코드를 넣어주었습니다. 만약 story에 abc가 들어있고 a가 00 b 가 01 c가 11이라면 result에는 000111이 들어가게 만들었습니다.

다음 출력 파일을 binary파일 형식으로 열었으며,



파일의 내용을 출력하기 전에 암호화해서 넣으면 나중에 해독을 할 수 있게 정보를 저장해야 합니다. 우선 가장 중요한 정보인 총 result의 길이와, 해독해야 되는 character의 개수를 압축파일에 저장했습니다. 다음 해독해야 되는 character들과 그 character들의 prefix코드를 압축파일에 저장했습니다. 이 과정에서 **DFS\_fprintf**함수를 사용하였습니다. 이 함수는 말 그대로 DFS로 모든 노드를 돌아 자식이 없는 노드, 즉 data를 가지고 있는 노드의 data값들과 그 노드에 도달하면서 만들어진 temp\_string에 저장된 prefix코드 그리고 그 prefix코드를 읽기 쉽게 하기 위해서 prefix코드의 길이를 압축파일에 저장하였습니다.

이제 이 파일에 출력하는 일만 남았습니다. 우선 result배열의 총 수가 8의 배수라면 가만히 놔두고, 만약 8의 배수가 아니라면 그 수보다 크면서 가장 가까운 8의 배수가 되도록 남은 공간을 0으로 채웠습니다. 이러한 이유는 result배열의 수를 8개씩 char변수로 저장을 해야 하는데, 8의 배수가 아니면 남은 1~7개의 result 배열 값 들에서 오류가 발생하기 때문입니다. 다음으로는 result배열을 8개씩 끊어서 **make\_dacade(char a)**함수에 넣어주었습니다. 이 함수는 입력 값을 이진수로 바꿔서 unsigned char에 저장을 하여 그 unsigned char을 반환합니다. 이 값을 그대로 출력 binary파일에 출력했습니다.

이제는 **decompressing**부분으로 넘어가도록 하겠습니다. 가장 먼저 파일을 읽는데, binary파일을 읽는 형식으로 열었습니다.



다음 가장 먼저 저장한 총 이진코드의 길이와, 해독해야 되는 character의 수를 변수에 저장하였습니다. 다음으로는 해독할 character값들과 그 값들에 해당하는 prefix코드를 저장하였습니다. 저장함과 동시에 tree형태로 다시 만들어 주었는데 이때 사용한 함수가 **AddChild**입니다. AddChild는 root부터 시작하여 자리를 찾아가는 형식인데, 찾아가려는 character값에 해당하는 prefix코드를 가지고 0이면 right자식으로 1이면 left자식으로 가서 prefix코드가 끝나는 곳에 그 character값을 data에 저장하였습니다. 만약 가는 길에 노드가 없다면 만들면서 갔습니다. 이제 트리도 준비가 되었고 총 문자의 개수도 알고 있습니다. 남은 일은 압축파일에 있는 unsigned char들을 읽어와서 이진수로 바꾼 다음 그 이진수를 길게 붙여서 처음부터 해당하는 prefix코드를 찾아 그 prefix코드에 해당하는 character을 출력하면 됩니다.

우선 남은 일 중의 첫 단계로 압축파일의 남아있는 모든 값들을 temp\_result라는 배열에 저장을 하였습니다. 이때 주의해야 하는 것이 압축파일에 적혀서 온 전체 이진 코드였을 때의 문자의 개수는 8의 배수가 아닙니다. 따라서 이 값을 8로 나누게 되면 받아야 되는 입력에서 1이 모자랄 수 있습니다. 이러한 오류를 막기 위해 8의 배수일때와 아닐 때를 나누어 주었습니다. 다음은 드디어 출력파일을 열고 마지막 출력을 할 단계입니다.

**Make\_binary**라는 함수를 통해서 temp\_result에 하나씩 저장해 놓았던 char들을 unsigned char로 읽어서 2진수로 다시 바꾸는 역할을 합니다. 바뀐 2진수는 result배열에 차곡차곡 쌓이게 됩니다. 물론 result와 temp\_result는 사용전에 미리 동적할당을 하였습니다. 이제 result에 길게 쌓여 있는 이진 코드를 처음부터 차근차근 읽어 나가면 됩니다. 이 과정을 하는 함수가 바로 DFS\_find\_char입니다. transfer라는 전역변수를 사용해 result의 처음부터 끝까지 훑어 가면서 root부터 0이면 right 자식, 1이면 left자식으로, 만약 자식이 없다면 그 값을 출력 파일에 출력하고 다시 root로 돌아가는 이 과정을 반복하였습니다. 만약 tranfer변수가 result\_size, 즉 result를 모두 돌면 종료가 되게 하였습니다.

총 7개의 함수를 사용하였으며, 주로 배열을 할 때는 malloc, realloc등을 사용한 동적할당을 하였습니다. 트리를 만들 때와 처음에 linked list를 만들 때 node를 사용하였으며, 압축파일을 쓰고 읽을 때, wb, wr을 사용하여 binary file형식으로 만들었습니다.

1. **실험 예시**

* 모든 ascii code를 사용하여 만든 text파일을 압축한 경우

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 원본 파일 | 압축 파일 | 출력 파일 | 압축률 |
| 예시 1 |  |  |  | 1.1283 |
| 예시 2 |  |  |  | 1.1415 |
| 예시 3 |  |  |  | 1.1426 |

* 숫자와 영문 대소문자만을 이용하여 만든 text파일을 압축한 경우

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 원본 파일 | 압축 파일 | 출력 파일 | 압축률 |
| 예시 1 |  |  |  | 1.3315 |
| 예시 2 |  |  |  | 1.3407 |
| 예시 3 |  |  |  | 1.3403 |

* Random 64개의 ascii code만을 사용하여 만든 text파일을 압축한 경우

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 원본 파일 | 압축 파일 | 출력 파일 | 압축률 |
| 예시 1 |  |  |  | 1.1296 |
| 예시 2 |  |  |  | 1.1429 |
| 예시 3 |  |  |  | 1.1441 |

* 압축률이 다른 이유?

각 input의 종류마다 압축률이 다른 이유는 무엇일까? 압축률을 판단하는 것은 등장하는 character의 frequency와 연관이 있다. 위의 예시에서는 input이 크게 3가지 종류가 있다. 모든 ascii코드를 사용하는 파일, 숫자와 영문 대소문자만을 사용하는 파일, 그리고 128개의 ascii코드중 64개만을 뽑아서 사용하는 경우이다. 하지만 각 input의 예시에 사용한 3개의 다른 파일 크기는 모든 input종류에서 같게 사용하였다. 즉, 크기가 같은 파일에 등장할 수 있는 character의 수가 다르니 당연히 frequency에 차이가 있을 것이고, frequency가 높은 character가 많을수록, 짧은 이진 코드로 나타낸 character가 많을 것이며 자연스럽게 압축 파일의 크기 또한 작아지게 된다. 또한 등장하는 character가 많을수록 압축파일에 저장해야 될 암호화된 character의 개수도 늘어남으로 자연스럽게 압축파일의 크기는 커지게 된다. 이렇기 때문에 input파일의 종류에 따라 압축률은 조금씩 차이를 보인다.