1. **소스코드 및 주석**

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <fstream>

#include <string>

#include <unordered\_map>

using namespace std;

struct huffmanNode {

char a;

int fre;

huffmanNode\* left;

huffmanNode\* right;

};

struct cmp {

bool operator()(huffmanNode\* a, huffmanNode\* b) {

return a->fre > b->fre;

}

};

class HuffmanTree {

public:

~HuffmanTree() {

releaseTree(root);

root = nullptr;

while (!pq.empty()) pq.pop();

}

private:

huffmanNode\* root = nullptr;

unordered\_map<char, int> ascii;

unordered\_map<char, string> code;

priority\_queue<huffmanNode\*, vector<huffmanNode\*>, cmp> pq;

string text;

public:

void Start() {

ifstream file("sample.txt");

while (getline(file, text)) {

for (char ch : text) {

++ascii[ch];

}

++ascii['\n'];

}

cout << endl << "=============================" << endl << " (1) 문자 빈도 카운트 및 출력" << endl << "=============================" << endl;

for (auto& kv : ascii) {

cout << kv.first << " : " << kv.second << endl;

}

for (const auto& iter : ascii) {

auto\* node = new huffmanNode{ iter.first, iter.second, nullptr, nullptr };

pq.push(node);

}

MakeTree();

FindTree(root, "");

}

void showCode() {

for (auto& kv : code) {

cout << kv.first << " : " << kv.second << "\n";

}

}

void encodeFile(const string& inFile, const string& outFile) {

ifstream fin(inFile);

ofstream fout(outFile);

cout << "3-결과. sample.txt -> sample.enc 인코딩 완료" << endl << endl;

char ch;

while (fin.get(ch)) {

fout << code[ch];

}

fin.close();

fout.close();

}

void decodeFile(const string& inFile, const string& outFile) {

ifstream fin(inFile);

ofstream fout(outFile);

cout << "4-결과. sample.enc -> sample.dec 디코딩 완료\n";

huffmanNode\* node = root;

char bit;

while (fin.get(bit)) {

if (bit == '0') node = node->left;

else node = node->right;

if (node->left == nullptr && node->right == nullptr) {

fout.put(node->a);

node = root;

}

}

fin.close();

fout.close();

}

void verifyFile(const string& origFile, const string& decFile) {

ifstream a(origFile);

ifstream b(decFile);

cout << "5. sample.dec와 sample.txt 일치 여부: ";

string lineA, lineB;

bool match = true;

while (true) {

bool eA = bool(getline(a, lineA));

bool eB = bool(getline(b, lineB));

if (!eA && !eB) break;

if (lineA != lineB) { match = false; break; }

}

cout << (match ? "Yes\n" : "No\n");

a.close(); b.close();

}

void compressionStats(const string& origFile, const string& encFile) {

ifstream o(origFile, ios::ate | ios::binary);

ifstream e(encFile, ios::ate | ios::binary);

long long origBits = o.tellg() \* 8;

long long encBits = e.tellg();

double ratio = (1.0 - double(encBits) / origBits) \* 100.0;

cout << "6. 압축 통계:" << endl;

cout << " 원본 비트 수: " << origBits << "\n";

cout << " 압축 후 비트 수: " << encBits << "\n";

cout << " 압축률: " << ratio << "%\n";

o.close(); e.close();

}

private:

void MakeTree() {

int rounds = pq.size() - 1;

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

auto\* left = pq.top(); pq.pop();

auto\* right = pq.top(); pq.pop();

auto\* parent = new huffmanNode{ 0, left->fre + right->fre, left, right };

pq.push(parent);

}

root = pq.top();

}

void FindTree(huffmanNode\* node, string bits) {

if (!node) return;

FindTree(node->left, bits + '0');

FindTree(node->right, bits + '1');

if (node->a != 0) {

code[node->a] = bits;

}

}

void releaseTree(huffmanNode\* node) {

if (!node) return;

releaseTree(node->left);

releaseTree(node->right);

delete node;

}

};

int main() {

HuffmanTree T;

T.Start();

cout << endl << "=============================" << endl << " (2) 1에서 추출한 값을 사용하여 huffman code를 완성하시오." << endl << "=============================" << endl;

T.showCode();

cout << endl << "=============================" << endl << " (3) 완성된 huffman code를 사용하여 sample.txt 파일을 encode 하시오. (sample.enc)" << endl << "=============================" << endl;

T.encodeFile("sample.txt", "sample.enc");

cout << endl << "=============================" << endl << " (4) 3번에서 enc된 파일을 원래 sample 파일로 decode 하시오. (sample.dec)" << endl << "=============================" << endl;

T.decodeFile("sample.enc", "sample.dec");

cout << endl << "=============================" << endl << " (5) 디코드된 sample.dec 파일과 원본인 sample.txt 파일이 일치하는지 확인하시오." << endl << "=============================" << endl;

T.verifyFile("sample.txt", "sample.dec");

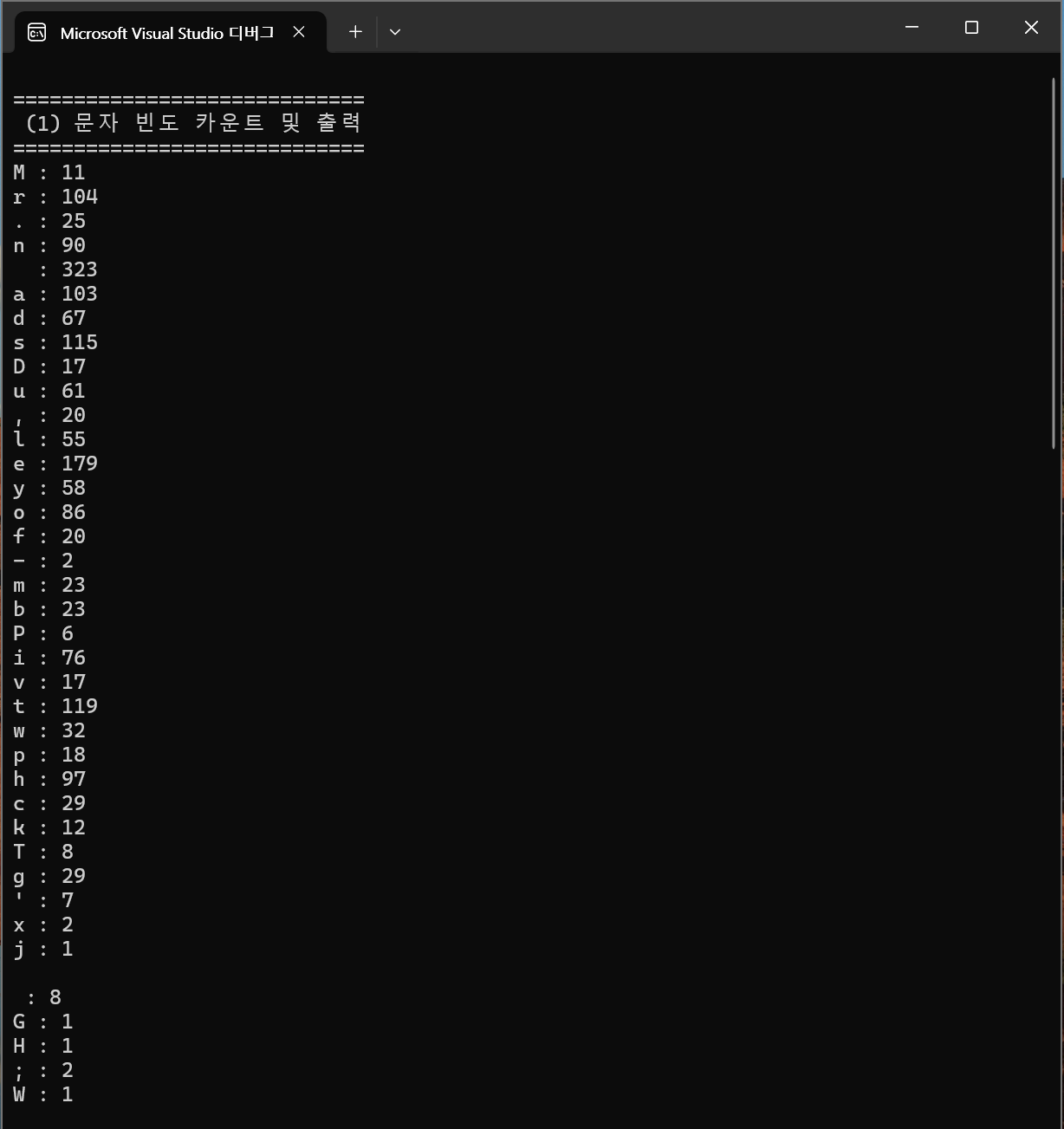
cout << endl << "=============================" << endl << " (6) 압축전과 후의 비트수를 계싼하여 압축률(%)을 계산하시오." << endl << "=============================" << endl;

T.compressionStats("sample.txt", "sample.enc");

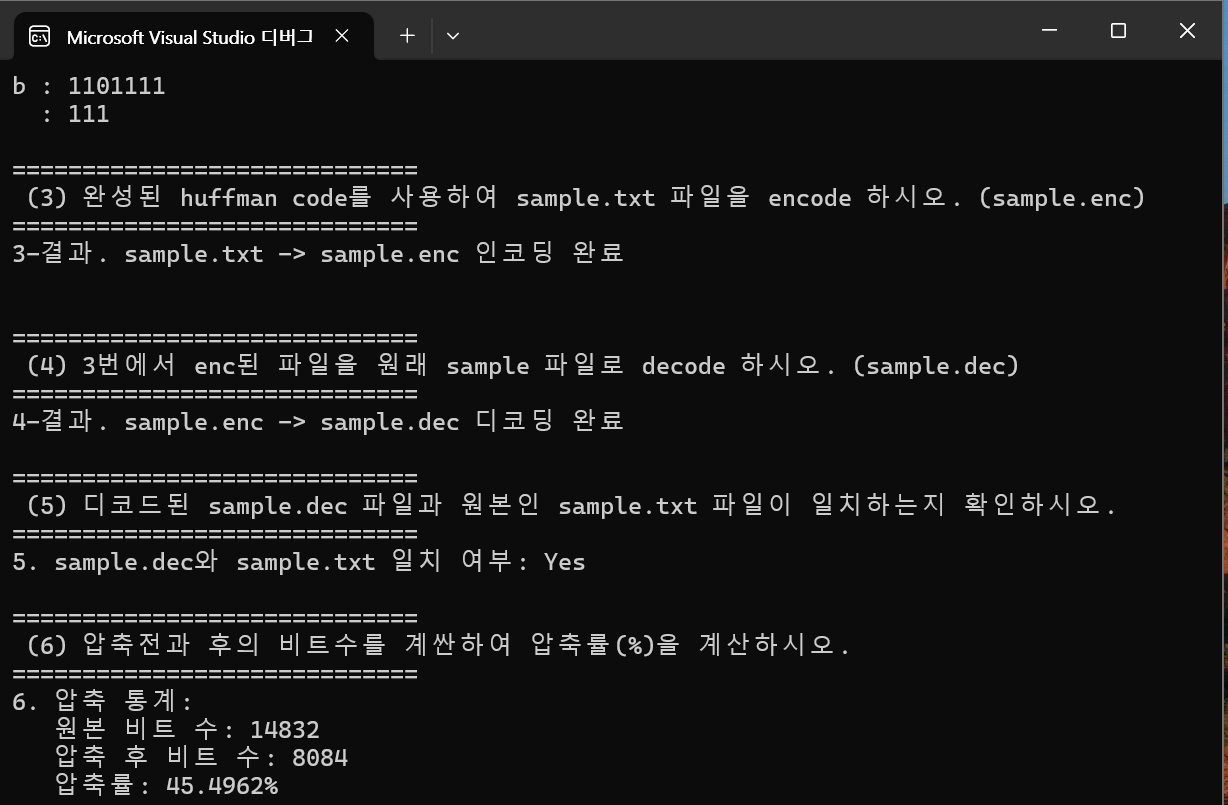
return 0;

}

1. **실행화면 캡처**







1. **고찰**

*사실 이번과제는 난이도가 높아서 C++언어 문법을 공부하는데 상당한 시간을 썻습니다 다음은, 몰랐던 C++언어 문법과 동작과정입니다*

**3-1. C++언어 문법 & 동작과정**

**[1] HuffmanTree 객체 생성: 빈도집계 준비**

T 객체가 메모리 위에 만들어짐 (기본 생성자 실행, 멤버 변수들은 선언 시 초기값(nullptr, 빈 컨테이너)으로 세팅)

**[2] Start 호출: 파일 읽기 → 빈도 집계 → 트리 구성 → 코드 생성**

[2]-1: ifstream file("sample.txt"); → 파일 열기

[2]-2: while (getline(file, text)) {...}

[2]-2-a: 한 줄 읽어 text에 저장

[2]-2-b: for (char ch : text) → 각 문자에 대해 ++ascii[ch] (빈도 집계)

[2]-3: for (const auto& iter : ascii)

[2]-3-a: new huffmanNode{ … } → 문자·빈도 노드 생성

[2]-3-b: pq.push(node) → 우선순위 큐에 삽입

[2]-4: MakeTree(): 허프만 트리 구성

[2]-4-a: rounds = pq.size() - 1

left = 가장 작은 빈도

right = 두 번째로 작은 빈도

parent = 합친 노드

[2]-4-b: 반복문 i=0…rounds-1

[2]-4-c: root = pq.top(); (최종 루트 설정)

[2]-5: 재귀 호출로 왼쪽(+'0')/오른쪽(+'1') 내려가기

[2]-5-a: 리프 노드 도달 시 code[char] = bits 저장

**[3] 허프만 코드 출력**

[3]-1: for (auto& kv : code) 순회

[3]-2: cout << kv.first << " : " << kv.second 출력

**[4] 인코딩 수행**

[4]-1: sample.txt에서 문자 하나씩 읽어 code[char]를 sample.enc에 기록

**[5] 디코딩 수행**

[5]-1: sample.enc에서 비트 하나씩 읽어 트리 탐색하여 문자 복원, sample.dec에 기록

1. **디코드된 파일과 원본 파일 비교**

[6]-1: 매개변수가 참조변수임에 유의

[6]-1-a: 파일 이름이므로, 참조변수를 안 사용해도 괜찮았을거 같다

[6]-2: ifstream를 이용하여 파일 읽기

**[7] 압축 전/후 비트 수 및 압축률 계산**

[7]-1: 매개변수가 참조변수임에 유의

[7]-1-a: 파일 이름이므로, 참조변수를 안 사용해도 괜찮았을거 같다

[7]-2: ifstream를 이용하여 파일 읽기

[7]-3: 핵심은 파일의 바이트 크기를 계산하는 코드

[7]-3-a: // origFile을 읽고 ios::binary를 비용하여 바이트 단위로 읽는다

// 동시에, ios::ate에 의해 커서가 맨 끝으로 이동!

ifstream o(origFile, ios::ate | ios::binary);

// 커서가 맨끝이므로, 파일 처음부터 끝까지의 비이트 길이를 의미함

// origBits에서 파일 비이트 길이를 8 곱해서 바이트 단위로 바꿔 입력

// (참고로, 바이트 크기가 너무 클 수 있어서 예제에서의

// 자료형은 long long를 사용하는 듯하다)

long long origBits = o.tellg() \* 8;

[7]-3-b: 디코딩된 파일의 크기도 동일하게 작성

**[8] 프로그램 정상 종료 → 프로그램 종료, 이후 T 소멸자(~HuffmanTree()) 자동 호출**

[8]-1: releaseTree(root) → 트리 노드 메모리 전부 해제

[8]-2: while (!pq.empty()) pq.pop(); → 큐 비우기

**3-2. 그외의 C++언어 문법**

**소멸자(~HuffmanTree)**

* 프로그램 종료 시 호출.
* releaseTree(root)로 트리 메모리 해제.
* root를 nullptr로 초기화.
* pq(우선순위 큐) 비우기.

**private 멤버**

* huffmanNode\* root : 허프만 트리의 루트 포인터.
* unordered\_map<char, int> ascii : 문자 빈도 저장.
* unordered\_map<char, string> code : 문자별 허프만 코드 저장.

**우선순위 큐 설명**

* 목적: 빈도가 작은 노드를 먼저 꺼내기 위해 사용.
* 선언:
  + 1. priority\_queue<구조체명\*, vector<구조체명\*>, cmp>
  + 2. cmp에서 정렬하는 메소드를 오버로딩한다
  + 3. 루트가 최소값인 우선순위 큐 선언
    - 3-a. 왜냐하면 cmp에서 첫번째의 fre값(a->fre)가 두번째의 fre값(b->fre)보다 클때 TRUE로 반환하기에 오름차순임에 유의
  + 4. 오름차순이므로 fre 값이 가장 작은 요소가 top()에 위치해 제일 먼저 꺼내진다.
* cmp 구조체로 fre(빈도) 기준 오름차순 정렬 (fre 작은 것부터 나옴).
* fre 값이 가장 작은 노드가 top()에 위치.
* 주요 메소드
  + push(): 노드 추가.
  + pop(): top() 제거.
  + top(): 현재 가장 우선순위 높은(= fre 가장 작은) 노드 반환.

**3-3. 주석을 포함한 전체 소스코드**

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <fstream>

#include <string>

#include <unordered\_map>

using namespace std;

struct huffmanNode {

char a;

int fre;

huffmanNode\* left;

huffmanNode\* right;

};

struct cmp {

bool operator()(huffmanNode\* a, huffmanNode\* b) {

return a->fre > b->fre;

}

};

class HuffmanTree {

public:

// [4] 프로그램 정상 종료

// + main()에서 return될때 호출되는 메소드이다

// + 소멸자: 객체가 수명 종료될 때 자동 호출되어 트리 메모리 해제

~HuffmanTree() {

releaseTree(root);

root = nullptr;

while (!pq.empty()) pq.pop();

}

private: // + 접근 제어 지정자: 이 아래 멤버들은 클래스 외부에서 직접 접근 불가

huffmanNode\* root = nullptr;

unordered\_map<char, int> ascii;

unordered\_map<char, string> code;

// + 우선순위 큐: 빈도가 작은 노드를 먼저 꺼내기 위해 사용 (cmp 비교자 사용)

// + 선언 형식

// 1. priority\_queue<구조체명\*, vector<구조체명\*>, cmp>

// 2. cmp에서 정렬하는 메소드를 오버로딩한다

// 3. 루트가 최소값인 우선순위 큐 선언

// 3-1. 왜냐하면 cmp에서 첫번째의 fre값(a->fre)가 두번째의 fre값(b->fre)보다 클때 TRUE로 반환하기에 오름차순임에 유의

// 4. 오름차순이므로 fre 값이 가장 작은 요소가 top()에 위치해 제일 먼저 꺼내진다.

// + 메소드

// + push() : 우선순위 큐에 원소를 추가한다

// + pop() : 우선순위 큐에서 top의 원소를 제거한다

// + top() : 우선순위 큐에서 가장 높은 우선순위 원소를 반환한다

priority\_queue<huffmanNode\*, vector<huffmanNode\*>, cmp> pq;

string text;

public:

// [2] 빈도집계부터 코드생성까지 한 번에 수행

// [2]-1: ifstream file("sample.txt"); → 파일 열기

// [2]-2: while (getline(file, text)) {...}

// [2]-2-a: 한 줄 읽어 text에 저장

// [2]-2-b : for (char ch : text) → 각 문자에 대해 ++ascii[ch](빈도 집계)

// [2]-3: for (const auto& iter : ascii)

// [2]-3-a : new huffmanNode{ … } → 문자·빈도 노드 생성

// [2]-3-b : pq.push(node) → 우선순위 큐에 삽입

// [2]-4: MakeTree(): 허프만 트리 구성

// [2]-4-a : rounds = pq.size() - 1

// + left = 가장 작은 빈도

// + right = 두 번째로 작은 빈도

// + parent = 합친 노드

// [2]-4-b : 반복문 i=0…rounds-1

// [2]-4-c : root = pq.top(); (최종 루트 설정)

// [2]-5:

// [2]-5-a : 재귀 호출로 왼쪽(+'0')/오른쪽(+'1') 내려가기

// [2]-5-b : 리프 노드 도달 시 code[char] = bits 저장

void Start() {

ifstream file("sample.txt");

while (getline(file, text)) {

for (char ch : text) {

// + 문자 빈도 카운트

++ascii[ch];

}

++ascii['\n'];

}

cout << endl << "=============================" << endl << " (1) 문자 빈도 카운트 및 출력" << endl << "=============================" << endl;

for (auto& kv : ascii) {

cout << kv.first << " : " << kv.second << endl;

}

for (const auto& iter : ascii) { // + ascii 맵 요소를 const 참조로 순회

// + auto\* node: auto로 타입(huffmanNode\*)을 추론. auto는 우항을 보고 자동으로 결정해줌!

auto\* node = new huffmanNode{ iter.first, iter.second, nullptr, nullptr };

pq.push(node);

}

// 허프만 트리 구성

MakeTree();

// 트리 순회하여 코드 맵 생성

FindTree(root, "");

}

// 코드 출력

// [3] T.showCode(); 호출

// [3]-1: for (auto& kv : code) 순회

// [3]-2: cout << kv.first << " : " << kv.second 출력

void showCode() {

for (auto& kv : code) {

cout << kv.first << " : " << kv.second << "\n";

}

}

// [4] 완성된 허프만 코드를 사용하여 파일 인코딩

// [4]-1: sample.txt에서 문자 하나씩 읽어 code[char]를 sample.enc에 기록

void encodeFile(const string& inFile, const string& outFile) {

ifstream fin(inFile);

ofstream fout(outFile);

cout << "3-결과. sample.txt -> sample.enc 인코딩 완료" << endl << endl;

char ch;

while (fin.get(ch)) {

fout << code[ch];

}

fin.close();

fout.close();

}

// [5] 인코딩된 파일(sample.enc)을 원본으로 디코딩

// [5]-1: sample.enc에서 비트 하나씩 읽어 tree 탐색하여 문자 복원, sample.dec에 기록

void decodeFile(const string& inFile, const string& outFile) {

ifstream fin(inFile);

ofstream fout(outFile);

cout << "4-결과. sample.enc -> sample.dec 디코딩 완료\n";

huffmanNode\* node = root;

char bit;

while (fin.get(bit)) {

if (bit == '0') node = node->left;

else node = node->right;

if (node->left == nullptr && node->right == nullptr) {

fout.put(node->a);

node = root;

}

}

fin.close();

fout.close();

}

// [6] 디코드된 파일과 원본 파일 비교

void verifyFile(const string& origFile, const string& decFile) {

ifstream a(origFile);

ifstream b(decFile);

cout << "5. sample.dec와 sample.txt 일치 여부: ";

string lineA, lineB;

bool match = true;

while (true) {

bool eA = bool(getline(a, lineA));

bool eB = bool(getline(b, lineB));

if (!eA && !eB) break;

if (lineA != lineB) { match = false; break; }

}

cout << (match ? "Yes\n" : "No\n");

a.close(); b.close();

}

// [7] 압축 전/후 비트 수 및 압축률 계산

void compressionStats(const string& origFile, const string& encFile) {

ifstream o(origFile, ios::ate | ios::binary);

ifstream e(encFile, ios::ate | ios::binary);

long long origBits = o.tellg() \* 8;

long long encBits = e.tellg();

double ratio = (1.0 - double(encBits) / origBits) \* 100.0;

cout << "6. 압축 통계:" << endl;

cout << " 원본 비트 수: " << origBits << "\n";

cout << " 압축 후 비트 수: " << encBits << "\n";

cout << " 압축률: " << ratio << "%\n";

o.close(); e.close();

}

private:

// 트리 생성: 빈도 작은 두 노드를 합쳐 새로운 내부 노드 생성

void MakeTree() {

int rounds = pq.size() - 1;

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

auto\* left = pq.top(); pq.pop(); // 1) left에서 가장 작은 빈도 노드 넣기. 그다음, pq에서 그노드 삭제

auto\* right = pq.top(); pq.pop(); // 2) right에서 두 번째 작은 빈도 노드 넣기. 그다음 pd에서 그노드 삭제

auto\* parent = new huffmanNode{ 0, left->fre + right->fre, left, right };

pq.push(parent); // 3) 합친 노드 다시 삽입

}

root = pq.top(); // + 루트 노드 설정

}

// + [4] 트리 순회: 리프마다 누적된 비트 문자열을 코드로 저장

void FindTree(huffmanNode\* node, string bits) {

if (!node) return;

FindTree(node->left, bits + '0');

FindTree(node->right, bits + '1');

if (node->a != 0) {

code[node->a] = bits; // + 리프 노드: 문자별 허프만 코드 저장

}

}

// + 메모리 해제: 후위 순회로 삭제

// + main()이 return될때 실행되는 함수안에 들어있는 메소드이다

void releaseTree(huffmanNode\* node) {

if (!node) return;

releaseTree(node->left);

releaseTree(node->right);

delete node; // + new로 할당된 메모리 해제: 객체의 소멸자 호출 및 메모리 반환

}

};

int main() {

// [1] HuffmanTree 객체 생성: 빈도집계 준비

// + T 객체가 메모리 위에 만들어짐 (기본 생성자 실행, 멤버 변수들은 선언 시 초기값(nullptr, 빈 컨테이너)으로 세팅)

HuffmanTree T;

// [2] Start 호출: 파일 읽기 → 빈도 집계 → 트리 구성 → 코드 생성

T.Start();

// [3] 허프만 코드 출력

cout << endl << "=============================" << endl << " (2) 1에서 추출한 값을 사용하여 huffman code를 완성하시오." << endl << "=============================" << endl;

T.showCode();

// [4] 인코딩 수행

cout << endl << "=============================" << endl << " (3) 완성된 huffman code를 사용하여 sample.txt 파일을 encode 하시오. (sample.enc)" << endl << "=============================" << endl;

T.encodeFile("sample.txt", "sample.enc");

// [5] 디코딩 수행

cout << endl << "=============================" << endl << " (4) 3번에서 enc된 파일을 원래 sample 파일로 decode 하시오. (sample.dec)" << endl << "=============================" << endl;

T.decodeFile("sample.enc", "sample.dec");

// [6] 디코드된 파일과 원본 파일 비교

cout << endl << "=============================" << endl << " (5) 디코드된 sample.dec 파일과 원본인 sample.txt 파일이 일치하는지 확인하시오." << endl << "=============================" << endl;

T.verifyFile("sample.txt", "sample.dec");

// [7] 압축 전/후 비트 수 및 압축률 계산

cout << endl << "=============================" << endl << " (6) 압축전과 후의 비트수를 계산하여 압축률(%)을 계산하시오." << endl << "=============================" << endl;

T.compressionStats("sample.txt", "sample.enc");

// [8] 프로그램 정상 종료 → 프로그램 종료, 이후 T 소멸자(~HuffmanTree()) 자동 호출

// [8]-1: releaseTree(root) → 트리 노드 메모리 전부 해제

// [8]-2: while (!pq.empty()) pq.pop(); → 큐 비우기

return 0;

}