外部排序-优化顺串生成阶段

09022313 李晶晶

问题描述/需求分析

在之前的外部排序中，初始顺串的生成是已经固定好RUN\_SIZE，在本次实验中，将优化顺串生成阶段，增大顺串长度。

系统结构/算法设计

采用败者树的方式生成不等长的顺串，

初始顺串生成完毕后，采用哈夫曼树的方式进行2路归并

功能模块设计

败者树生成不等长顺串

class Node

{

public:

int value;

int index;

bool vaild = true;

Node(int v, int i) :value(v), index(i) {}

Node() {}

};

class LoserTree

{

public:

int size;

int leaf;

vector<Node> tree;

int max;

LoserTree(int n,vector<int> input)

Node build()

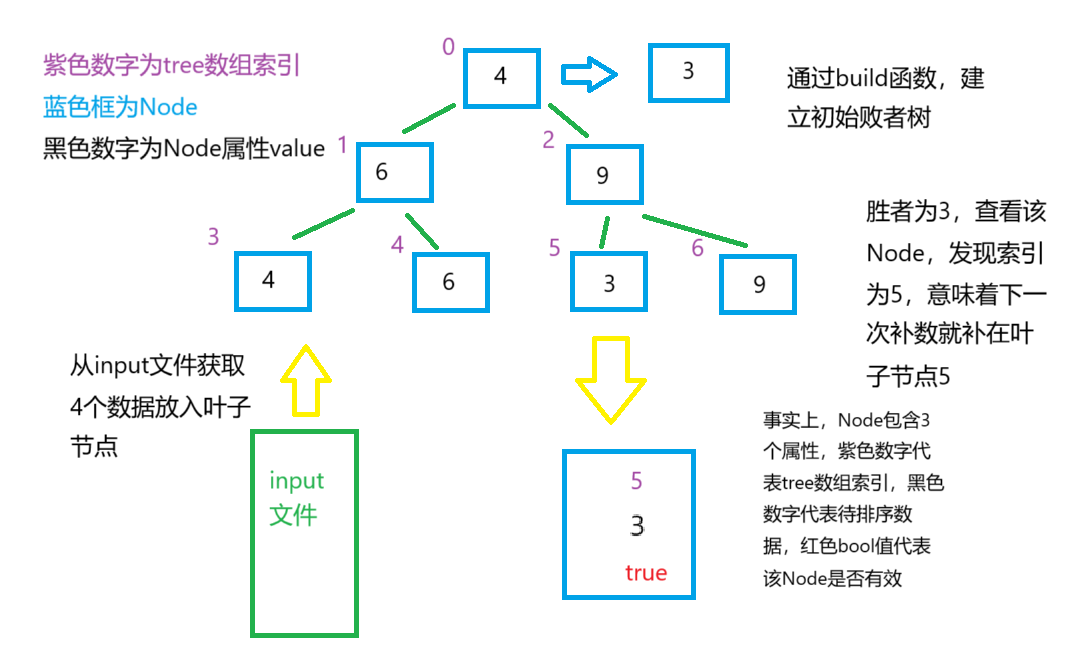
Node update(int index,int num)

bool checkVaild()

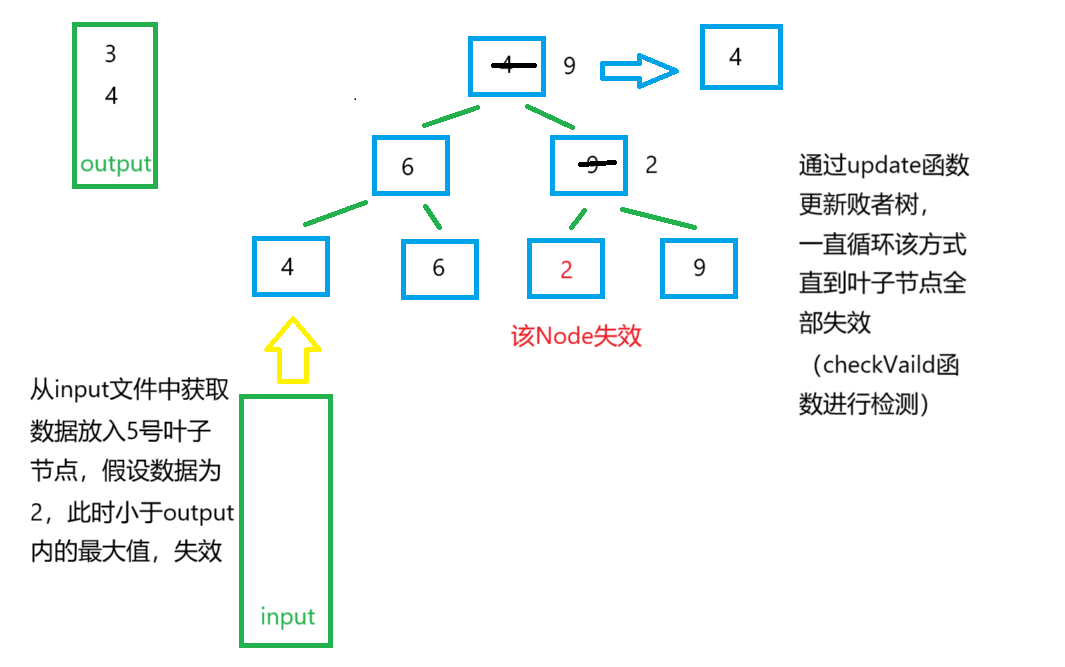
void turntrue()

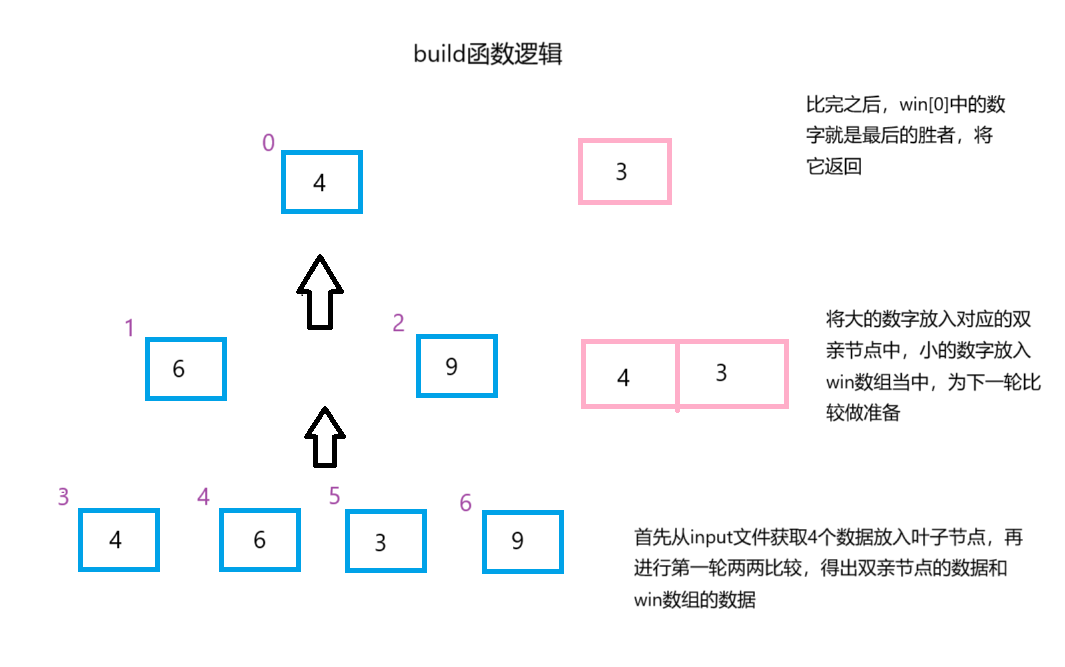
};

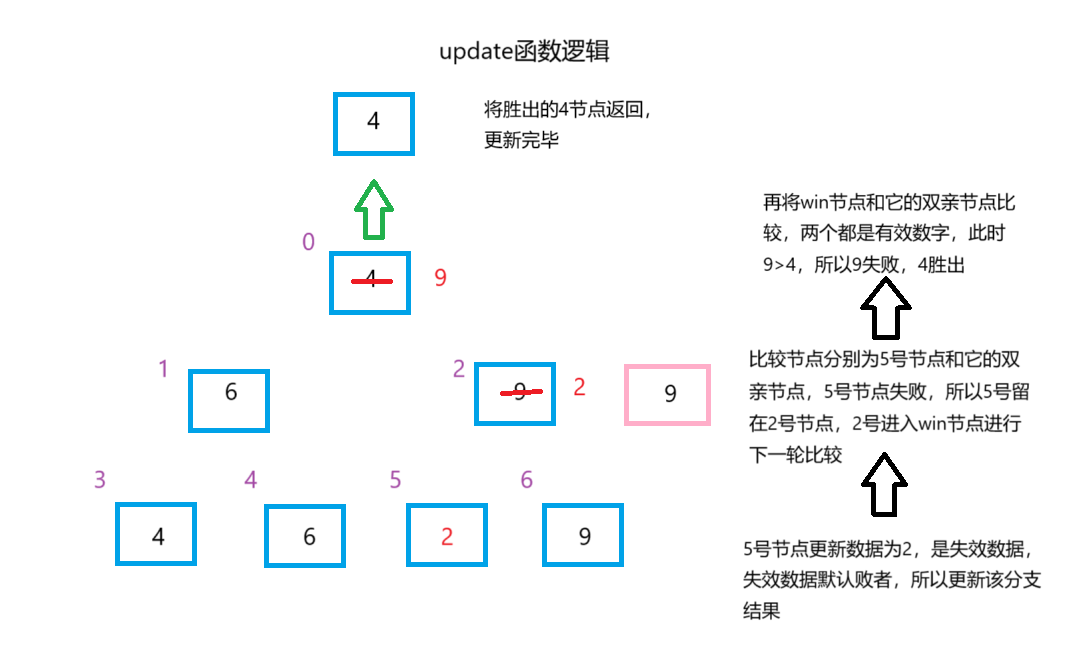
顺串生成大致流程



设置input缓冲区大小为leaf大小，input中的数据预取自file input.txt。







当叶子节点中所有节点均为false，则该初始顺串生成完毕。

此时败者树构建完毕，初始顺串生成完毕，存储到临时文件，文件名形如0tempx

并将叶子节点均重置为true，重复上述过程。

接下来通过哈夫曼树进行2路归并

哈夫曼树进行2路归并

class HuffmanNode

{

public:

int length;

string filename;

HuffmanNode(int l,string f):length(l),filename(f){}

HuffmanNode(){}

};

class HuffmanTree

{

public:

vector<HuffmanNode> tree;

int ioReadCount = 0; // To count read operations

int ioWriteCount = 0; // To count write operations

static bool compareByAttribute1(const HuffmanNode& a, const HuffmanNode& b)

{

return a.length < b.length;

}

// Function to count the number of data entries in the file

int countData(const string& filename)

HuffmanTree(int num)

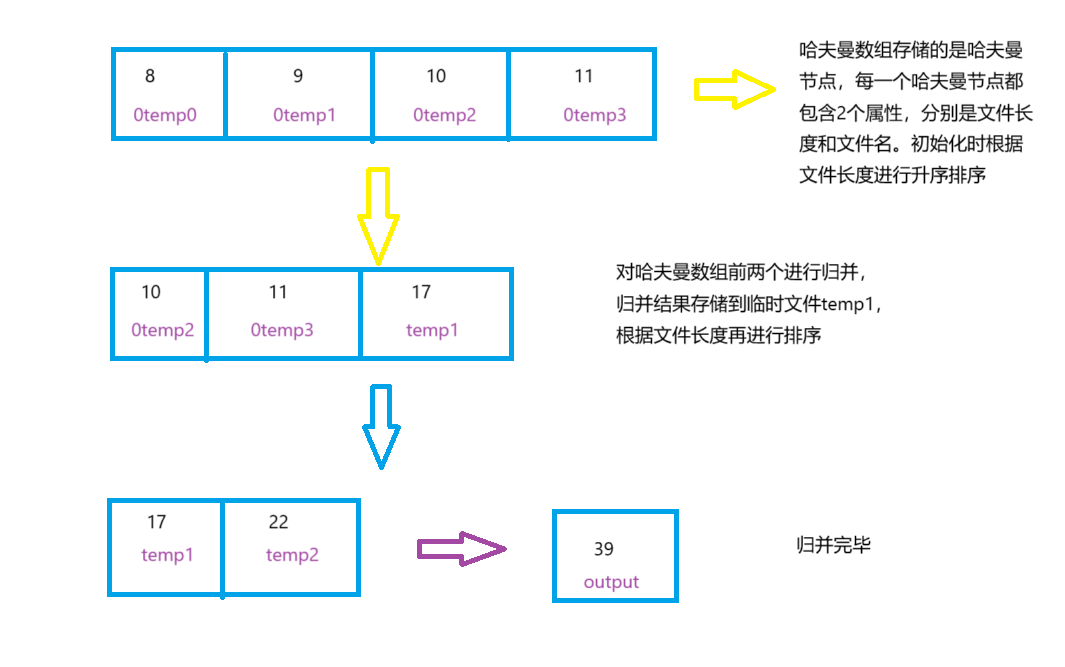
// Function to merge two chunks into an output file

void mergeChunks(const string& file1, const string& file2, const string& outputFile)

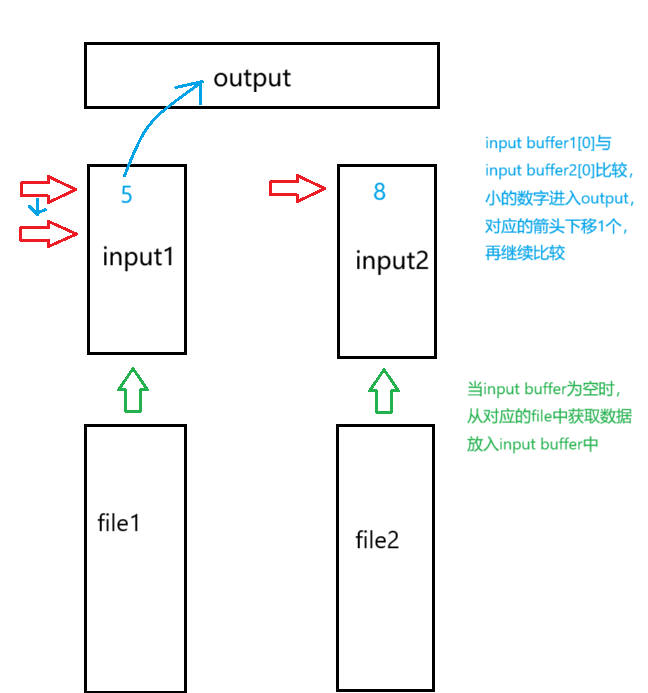
int merge()

};

哈夫曼树大致流程



2个文件归并的思路与前一份报告思路一致



结果测试与分析

实验结果与预期一致

顺串生成

文本

中度可信度描述已自动生成

文本

描述已自动生成

长度11

图形用户界面, 文本, 应用程序

描述已自动生成

长度17

图形用户界面, 文本, 应用程序

描述已自动生成

长度16

图形用户界面, 文本, 应用程序

描述已自动生成

长度6

可以看到生成了一系列不等长顺串

接下来是归并

图片包含 图形用户界面

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

分析可以得出，首先对0temp0和0temp3进行归并生成temp1，长度变为17，接下来对0temp2和0temp1进行归并生成temp2，最后对temp1和temp2进行归并生成temp3，返回3的索引，最后将temp3更名为output文件，归并完成。

IO性能分析

设定leaf为8，以N为横坐标，IO次数为纵坐标，同时与前一份报告中没有优化的外部排序数据做对比

表格

描述已自动生成

蓝色线为未优化的曲线，橙色线为优化过后的曲线。可以看出，优化顺串生成后的IO次数少于未优化。

实验总结

这个实验当中遇到了最多的问题，一开始就是出现了如何处理失效数据的问题，这个问题直接影响力这个败者树应该怎么构建。一开始我想将失效数据直接变成无穷大再重新构建败者树，但这个解决办法让下一轮解除失效后进行排序这一环节无法进行，如果设置真假值的话显然不行，于是加入了vaild属性。接踵而来的问题就是如何拿到下一个填充数据的位置。所以很自然的加入了index属性，这样在返回胜者后可以直接获取胜者在败者树叶子节点的索引，获取下一次填充数据的位置。那么败者树怎么构建？怎么更新？这是下面的问题。构建败者树时引入win数组来存储这一轮的胜者，解决构建问题。根据vaild值，解决更新败者树的问题。后面哈夫曼树构建比较顺畅，有了之前的框架打底，自然就能构建出来。

附录

#include<iostream>

#include<vector>

using namespace std;

class Node

{

public:

int value;

int index;

bool vaild = true;

Node(int v, int i) :value(v), index(i) {}

Node() {}

};

#include<iostream>

#include<vector>

#include<limits>

#include<cmath>

#include"Node.h"

using namespace std;

class LoserTree

{

public:

int size;

int leaf;

vector<Node> tree;

int max;

LoserTree(int n, vector<int> input)

{

size = 2 \* n - 1; leaf = n;

tree.resize(size);

for (int i = 0; i < n; i++)

{

tree[i + n - 1] = Node(input[i], i + n - 1);

}

}

Node build()

{

int pass = log2(size + 1);

vector<Node> win(leaf);

for (int i = 0; i < leaf; i++)

{

win[i] = tree[leaf - 1 + i];

}

for (int i = 1; i < pass; i++)

{

int num\_count = leaf / i;

int new\_count = num\_count / 2;

vector<Node> temp(new\_count);

int start = pow(2, pass - 1 - i) - 1;

for (int j = 0; j < new\_count; j++)

{

tree[start + j] = win[2 \* j].value < win[2 \* j + 1].value ? win[2 \* j + 1] : win[2 \* j];

temp[j] = win[2 \* j].value < win[2 \* j + 1].value ? win[2 \* j] : win[2 \* j + 1];

}

win.resize(new\_count);

for (int j = 0; j < new\_count; j++)

{

win[j] = temp[j];

}

}

max = win[0].value;

return win[0];

}

Node update(int index, int num)

{

tree[index].value = num;

if (num < max)

{

tree[index].vaild = false;

}

else

{

tree[index].vaild = true;

}

int compare = (index - 1) / 2;

Node win = tree[index];

while (compare > 0)

{

if (win.vaild == false && tree[compare].vaild == true)

{

Node temp = win;

win = tree[compare];

tree[compare] = temp;

}

if (win.vaild == true && tree[compare].vaild == true)

{

if (win.value > tree[compare].value)

{

Node temp = win;

win = tree[compare];

tree[compare] = temp;

}

}

compare = (compare - 1) / 2;

}

compare = 0;

if (win.vaild == false && tree[compare].vaild == true)

{

Node temp = win;

win = tree[compare];

tree[compare] = temp;

}

if (win.vaild == true && tree[compare].vaild == true)

{

if (win.value > tree[compare].value)

{

Node temp = win;

win = tree[compare];

tree[compare] = temp;

}

}

max = win.value;

return win;

}

bool checkVaild()

{

for (int i = leaf - 1; i < size; i++)

{

if (tree[i].vaild == true)

return true;

}

return false;

}

void turntrue()

{

for (int i = leaf - 1; i < size; i++)

{

tree[i].vaild = true;

}

}

};

#include<iostream>

#include<vector>

using namespace std;

class HuffmanNode

{

public:

int length;

string filename;

HuffmanNode(int l, string f) :length(l), filename(f) {}

HuffmanNode() {}

};

#include<iostream>

#include<vector>

#include <fstream>

#include <string>

#include <algorithm>

#include"HuffmanNode.h"

const int BUFFER\_SIZE = 3;

class HuffmanTree

{

public:

vector<HuffmanNode> tree;

int ioReadCount = 0; // To count read operations

int ioWriteCount = 0; // To count write operations

static bool compareByAttribute1(const HuffmanNode& a, const HuffmanNode& b)

{

return a.length < b.length;

}

// Function to count the number of data entries in the file

int countData(const string& filename)

{

ifstream input(filename);

int count = 0;

int num;

while (input >> num)

{

count++;

}

return count;

}

HuffmanTree(int num)

{

tree.resize(num);

for (int i = 0; i < num; i++)

{

string file = to\_string(0) + "temp" + to\_string(i) + ".txt";

int length = countData(file);

HuffmanNode node(length, file);

tree[i] = node;

}

sort(tree.begin(), tree.end(), compareByAttribute1);

}

// Function to merge two chunks into an output file

void mergeChunks(const string& file1, const string& file2, const string& outputFile)

{

ifstream input1(file1);

ifstream input2(file2);

ofstream output(outputFile);

vector<int> buffer1(BUFFER\_SIZE);

vector<int> buffer2(BUFFER\_SIZE);

size\_t index1 = 0, index2 = 0;

while (index1 < BUFFER\_SIZE && input1 >> buffer1[index1])

{

index1++;

}

ioReadCount++;

while (index2 < BUFFER\_SIZE && input2 >> buffer2[index2])

{

index2++;

}

ioReadCount++;

while (index1 > 0 || index2 > 0)

{

if (index1 == 0)

{

output << buffer2[0] << " ";

copy(buffer2.begin() + 1, buffer2.begin() + index2, buffer2.begin());

index2--;

if (input2 >> buffer2[index2])

{

index2++;

}

}

else if (index2 == 0)

{

output << buffer1[0] << " ";

copy(buffer1.begin() + 1, buffer1.begin() + index1, buffer1.begin());

index1--;

if (input1 >> buffer1[index1])

{

index1++;

}

}

else if (buffer1[0] < buffer2[0])

{

output << buffer1[0] << " ";

copy(buffer1.begin() + 1, buffer1.begin() + index1, buffer1.begin());

index1--;

if (input1 >> buffer1[index1])

{

index1++;

}

}

else

{

output << buffer2[0] << " ";

copy(buffer2.begin() + 1, buffer2.begin() + index2, buffer2.begin());

index2--;

if (input2 >> buffer2[index2])

{

index2++;

}

}

}

input1.close();

input2.close();

output.close();

}

int merge()

{

int fileIndex = 1;

while (tree.size() > 1)

{

string file1 = tree[0].filename;

string file2 = tree[1].filename;

string file = "temp" + to\_string(fileIndex) + ".txt";

mergeChunks(file1, file2, file);

HuffmanNode node(tree[0].length + tree[1].length, file);

tree = vector<HuffmanNode>(tree.begin() + 2, tree.end());

tree.push\_back(node);

sort(tree.begin(), tree.end(), compareByAttribute1);

fileIndex++;

}

return fileIndex;

}

};

#include <iostream>

#include <vector>

#include <limits>

#include <fstream>

#include <queue>

#include <string>

#include <algorithm>

#include <filesystem>

#include <cstdlib>

#include <ctime>

#include <algorithm>

#include "LoserTree.h"

#include"HuffmanTree.h"

const int leaf = 16;

int ioReadCount = 0; // To count read operations

int ioWriteCount = 0; // To count write operations

// Function to generate an input file with N random numbers

void generateInputFile(const string& filename, int N)

{

ofstream output(filename);

if (!output.is\_open())

{

cerr << "Error creating input file." << endl;

return;

}

// Seed the random number generator

srand(static\_cast<unsigned int>(time(0)));

for (int i = 0; i < N; ++i)

{

output << rand() % 100 << " ";

}

output.close();

}

int generateRun(ifstream& input)

{

vector<int> inputbuffer;

vector<int> outputbuffer;

int num;

while (inputbuffer.size() < leaf && input >> num)

{

inputbuffer.push\_back(num);

}

ioReadCount++;

LoserTree lt(leaf, inputbuffer);

Node win = lt.build();

outputbuffer.push\_back(win.value);

inputbuffer.clear();

int i = 0;// runIndex

while (true)

{

while (lt.checkVaild())

{

if (inputbuffer.empty())

{

while (inputbuffer.size() < leaf && input >> num)

{

inputbuffer.push\_back(num);

}

ioReadCount++;

if (input.eof())

{

goto label;

}

}

if (!inputbuffer.empty())

{

num = inputbuffer.back();

inputbuffer.pop\_back();

win = lt.update(win.index, num);

outputbuffer.push\_back(win.value);

}

} // One run is generated

outputbuffer.pop\_back();

ofstream tempFile(to\_string(0) + "temp" + to\_string(i) + ".txt");

for (int val : outputbuffer)

{

tempFile << val << " ";

}

ioWriteCount++;

tempFile.close();

outputbuffer.clear();

lt.max = 0;

lt.turntrue();

win = lt.build();

outputbuffer.push\_back(win.value);

i++;

}

label:

int index = win.index;

for (int i = leaf - 1; i < 2 \* leaf - 1; i++)

{

if (lt.tree[i].vaild == true&&i!=index)

{

outputbuffer.push\_back(lt.tree[i].value);

}

}

sort(outputbuffer.begin(), outputbuffer.end());

ofstream tempFile(to\_string(0) + "temp" + to\_string(i) + ".txt");

for (int val : outputbuffer)

{

tempFile << val << " ";

}

ioWriteCount++;

tempFile.close();

i++;

vector<int> remain;

for (int i = leaf - 1; i < 2 \* leaf - 1; i++)

{

if (lt.tree[i].vaild == false)

{

remain.push\_back(lt.tree[i].value);

}

}

sort(remain.begin(), remain.end());

ofstream tempFile1(to\_string(0) + "temp" + to\_string(i) + ".txt");

for (int val : remain)

{

tempFile1 << val << " ";

}

ioWriteCount++;

tempFile1.close();

return i;

}

void huffmanTreeMerge(int totalChunks)

{

HuffmanTree ht(totalChunks);

int num = ht.merge();

string oldFileName = "temp" + to\_string(num - 1) + ".txt";

string newFileName = "output.txt";

ioReadCount += ht.ioReadCount;

ioWriteCount += ht.ioWriteCount;

filesystem::rename(oldFileName, newFileName);

}

int main()

{

const string inputFileName = "input.txt";

const int N = 8000;

generateInputFile(inputFileName, N);

ifstream input("input.txt");

input.clear();

input.seekg(0);

int num = generateRun(input);

huffmanTreeMerge(num + 1);

cout << "ioRead count " << ioReadCount << endl;

cout << "ioWrite count " << ioWriteCount << endl;

cout << ioReadCount + ioWriteCount << endl;

}