|  |  |
| --- | --- |
| 实验序号 | 4 |
| 学号 |  |
| 姓名 |  |
| 成绩 |  |
| 截止日期 |  |
| 批阅 |  |

1. 问题说明
2. 测试文件有两个。一个文件（p04\_textData\*\*\*.in）是考生信息数据，包含了准考证号(准考证编号规律参考实验二) ，成绩，名次，同名次的人数。这个文件里的数据，按成绩从高到低排序。把这些数据从磁盘文件读入到内存，并组织合适的数据结构。然后从p04\_textData\*\*\*b.in逐一读取每个准考证，在内存里查询这个准考证号，并输出查询结果（要求程序运行时间在9秒内）。
3. 【输入】
4. 测试文件有两个。p04\_textData\*\*\*.in和p04\_textData\*\*\*b.in。
5. 【输出】
6. 输出一个文本文件(p04\_textData\*\*\*.out)，保存查询结果，每个考生一行，每行输出查询到的准考证号、成绩、名次和同名次人数。

二．数据结构说明

namespace my {  
 // 二叉搜索树  
 template<typename T, typename Compare = std::less<T> >  
 // Compare 为比较函数对象，用于比较两个元素的大小  
 class BinaryTree {  
 private:  
 // 节点类  
 struct node {  
 T data; // 数据  
 node \*left; // 左子树  
 node \*right; // 右子树  
 node \*parent; // 父节点  
  
 // 构造函数  
 explicit node(const T &d, node \*l = NULL, node \*r = NULL, node \*p = NULL) : data(d), left(l), right(r),  
 parent(p) {}  
 };  
  
 node \*root; // 根节点  
 Compare cmp; // 比较函数对象  
  
 public:  
 // 空构造函数  
 // 将节点设置为空 并且传入比较器  
 BinaryTree() : root(NULL), cmp(Compare()) {}  
  
 // 析构函数  
 ~BinaryTree() {  
 clear();  
 }  
  
 // 清空树  
 void clear() {  
 clear(root);  
 }  
  
 // 清空当前节点下的树  
 void clear(node \*t) {  
 if (t == NULL) return;  
 clear(t->left);  
 clear(t->right);  
 delete t;  
 }  
  
 // 查找元素  
 node \*find(const T &x) const {  
 return find(x, root);  
 }  
  
 // 在当前节点下查找元素  
 node \*find(const T &x, node \*t) const {  
 if (t == NULL || t->data == x) return t; // 如果当前节点为空那么返回NULL或者当前节点的值等于x 返回当前节点  
 if (cmp(x, t->data)) return find(x, t->left); // 如果x小于当前节点的值 那么在左子树中查找  
 else return find(x, t->right); // 否则在右子树中查找  
 }  
  
 // 判断是否包含元素  
 bool contain(const T &x) const {  
 return find(x) != NULL;  
 }  
  
 // 插入元素  
 bool insert(const T &x) {  
 if (root == NULL) { // 如果根节点为空 那么直接插入  
 root = new node(x); // 创建新节点  
 return true; // 创建新节点返回true  
 }  
 node \*t = root; // 否则从根节点开始查找  
 while (true) {  
 if (t->data == x) return false; // 如果有原来节点 那么返回false  
 if (cmp(x, t->data)) { // 如果x小于当前节点的值  
 if (t->left == NULL) { // 如果左子树为空  
 t->left = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->left; // 否则继续在左子树中查找  
 } else {  
 if (t->right == NULL) { // 如果右子树为空  
 t->right = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->right; // 否则继续在右子树中查找  
 }  
 }  
 }  
  
  
 bool remove(const T &x) {  
 node \*t = find(x);  
 if (t == NULL) return false;  
 if (t->left != NULL && t->right != NULL) {  
 node \*tmp = t->right;  
 while (tmp->left != NULL) tmp = tmp->left; //  
 t->data = tmp->data;  
 t = tmp;  
 }  
 node \*child = t->left != NULL ? t->left : t->right;  
 if (t == root) {  
 root = child;  
 if (root != NULL) root->parent = NULL;  
 } else {  
 node \*parent = t->parent;  
 if (parent->left == t) parent->left = child;  
 else parent->right = child;  
 if (child != NULL) child->parent = parent;  
 }  
 delete t;  
 return true;  
 }  
 };  
  
 // 求C风格字符串长度  
 size\_t strlen(const char \*s) {  
 size\_t len = 0;  
 while (s[len] != '\0') {  
 len++;  
 }  
 return len;  
 }  
  
 // 拷贝C风格字符串  
 void strcpy(char \*dest, const char \*src) {  
 size\_t i = 0;  
 while (src[i] != '\0') {  
 dest[i] = src[i];  
 i++;  
 }  
 dest[i] = '\0';  
 }  
  
 // 判断是否为空白字符  
 bool isspace(char c) {  
 return c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f' || c == '\v';  
 }  
  
 // 求最小值  
 template<typename T>  
 T min(const T &a, const T &b) {  
 return a < b ? a : b;  
 }  
  
 // 求最大值  
 template<typename T>  
 T max(const T &a, const T &b) {  
 return a > b ? a : b;  
 }  
  
 // 字符串类  
 class string {  
 public:  
  
 // 默认构造函数 构造一个不包含任何字符的字符串  
 string() : m\_data(new char[1]), m\_size(0), m\_capacity(1) {  
 m\_data[0] = '\0';  
 }  
  
 // 构造函数 从C风格字符串中构造字符串  
 explicit string(const char \*str) : m\_data(new char[my::strlen(str) + 1]), m\_size(my::strlen(str)),  
 m\_capacity(my::strlen(str) + 1) {  
 my::strcpy(m\_data, str);  
 }  
  
 // 拷贝构造函数  
 string(const string &other) : m\_data(new char[other.m\_capacity]), m\_size(other.m\_size),  
 m\_capacity(other.m\_capacity) {  
 my::strcpy(m\_data, other.m\_data);  
 }  
  
 // 析构函数  
 ~string() {  
 delete[] m\_data;  
 }  
  
 // 清空字符串  
 void erase() {  
 delete[] m\_data;  
 m\_data = new char[1];  
 m\_data[0] = '\0';  
 m\_size = 0;  
 m\_capacity = 1;  
 }  
  
 // 追加C风格字符串  
 void append(const char \*str, size\_t n) {  
 if (m\_size + n + 1 > m\_capacity) {  
 reserve(m\_size + n + 1);  
 }  
 my::strcpy(m\_data + m\_size, str);  
 m\_size += n;  
 m\_data[m\_size] = '\0';  
 }  
  
 // 复制构造函数  
 string &operator=(const string &other) {  
 if (this != &other) {  
 char \*temp = new char[other.m\_capacity];  
 my::strcpy(temp, other.m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_size = other.m\_size;  
 m\_capacity = other.m\_capacity;  
 }  
 return \*this;  
 }  
  
 // 追加字符串  
 string operator+(const string &other) const {  
 string newString;  
 newString.m\_size = m\_size + other.m\_size;  
 newString.m\_capacity = newString.m\_size + 1;  
 newString.m\_data = new char[newString.m\_capacity];  
 my::strcpy(newString.m\_data, m\_data);  
 my::strcpy(newString.m\_data + m\_size, other.m\_data);  
 return newString;  
 }  
  
 // 散列仿函数  
 class hash {  
 public:  
 size\_t operator()(const string &str) const {  
 size\_t hash = 0;  
 for (size\_t i = 0; i < str.size(); ++i) {  
 hash = hash \* 131 + str[i];  
 }  
 return hash;  
 }  
 };  
  
 // 相等运算符  
 bool operator==(const string &other) const {  
 if (m\_size != other.m\_size) { // 如果长度不相等 那么直接返回false  
 return false;  
 }  
 for (size\_t i = 0; i < m\_size; ++i) { // 否则逐个比较字符  
 if (m\_data[i] != other.m\_data[i]) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 bool operator!=(const string &other) const {  
 return !(\*this == other);  
 }  
  
 // 小于运算符  
 bool operator<(const string &other) const {  
 size\_t minSize = min(m\_size, other.m\_size); // 求最小长度  
 for (size\_t i = 0; i < minSize; ++i) {  
 if (m\_data[i] < other.m\_data[i]) { // 逐个比较字符  
 return true;  
 } else if (m\_data[i] > other.m\_data[i]) {  
 return false;  
 }  
 }  
 return m\_size < other.m\_size; // 如果前面的字符都相等 那么长度小的字符串小  
 }  
  
 // operator >  
 bool operator>(const string &other) const {  
 return other < \*this;  
 }  
  
 // operator <=  
 bool operator<=(const string &other) const {  
 return !(other < \*this);  
 }  
  
 // operator >=  
 bool operator>=(const string &other) const {  
 return !(\*this < other);  
 }  
  
 // 取出字符串中的字符  
 char &operator[](size\_t index) {  
 return m\_data[index];  
 }  
  
 // 取出字符串中的字符  
 const char &operator[](size\_t index) const {  
 return m\_data[index];  
 }  
  
 // 返回字符串长度  
 size\_t size() const {  
 return m\_size;  
 }  
  
 // 返回字符串容量  
 size\_t capacity() const {  
 return m\_capacity;  
 }  
  
 // 扩容  
 void reserve(size\_t new\_capacity) {  
 if (new\_capacity > m\_capacity) {  
 char \*temp = new char[new\_capacity];  
 my::strcpy(temp, m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_capacity = new\_capacity;  
 }  
 }  
  
 // 重新设置字符串长度  
 void resize(size\_t new\_size) {  
 if (new\_size > m\_capacity) {  
 reserve(new\_size);  
 }  
 for (size\_t i = m\_size; i < new\_size; ++i) {  
 m\_data[i] = '\0';  
 }  
 m\_size = new\_size;  
 }  
  
 // 在字符串末尾追加字符  
 void push\_back(char c) {  
 if (m\_size + 1 > m\_capacity) {  
 reserve(m\_capacity \* 2);  
 }  
 m\_data[m\_size] = c;  
 m\_data[m\_size + 1] = '\0';  
 ++m\_size;  
 }  
  
 // 返回C风格字符串  
 const char \*c\_str() const {  
 return m\_data;  
 }  
  
 // C++流输出运算符  
 friend std::ostream &operator<<(std::ostream &os, const string &str) {  
 os << str.m\_data;  
 return os;  
 }  
  
 // C++流输入运算符  
 friend std::istream &operator>>(std::istream &is, string &str) {  
 char c = '\0'; // 将str中的数据清空  
 while (is.get(c) && my::isspace(c)); // 跳过空白字符  
 if (is) { // 如果输入流正常  
 str.m\_size = 0; // 将str中的数据清空  
 do {  
 if (str.m\_size + 1 > str.m\_capacity) { // 如果空间不足 那么扩容  
 str.reserve(str.m\_capacity \* 2);  
 }  
 str.m\_data[str.m\_size] = c;  
 ++str.m\_size;  
 } while (is.get(c) && !my::isspace(c)); // 读取非空白字符  
 str.m\_data[str.m\_size] = '\0'; // 在末尾添加'\0'  
 if (is) is.unget(); // 如果输入流正常 那么将最后读取的字符放回输入流  
 else is.clear(is.rdstate() & ~std::ios\_base::failbit); // 否则清除错误标志  
 }  
 return is;  
 }  
  
 private:  
 char \*m\_data; // 字符串数据  
 size\_t m\_size; // 字符串长度  
 size\_t m\_capacity; // 字符串容量  
 };  
}

三．算法说明

#include <functional>  
#include <fstream>  
#include <chrono>  
#include <iostream>  
  
namespace my {  
 // 二叉搜索树  
 template<typename T, typename Compare = std::less<T> >  
 // Compare 为比较函数对象，用于比较两个元素的大小  
 class BinaryTree {  
 private:  
 // 节点类  
 struct node {  
 T data; // 数据  
 node \*left; // 左子树  
 node \*right; // 右子树  
 node \*parent; // 父节点  
  
 // 构造函数  
 explicit node(const T &d, node \*l = NULL, node \*r = NULL, node \*p = NULL) : data(d), left(l), right(r),  
 parent(p) {}  
 };  
  
 node \*root; // 根节点  
 Compare cmp; // 比较函数对象  
  
 public:  
 // 空构造函数  
 // 将节点设置为空 并且传入比较器  
 BinaryTree() : root(NULL), cmp(Compare()) {}  
  
 // 析构函数  
 ~BinaryTree() {  
 clear();  
 }  
  
 // 清空树  
 void clear() {  
 clear(root);  
 }  
  
 // 清空当前节点下的树  
 void clear(node \*t) {  
 if (t == NULL) return;  
 clear(t->left);  
 clear(t->right);  
 delete t;  
 }  
  
 // 查找元素  
 node \*find(const T &x) const {  
 return find(x, root);  
 }  
  
 // 在当前节点下查找元素  
 node \*find(const T &x, node \*t) const {  
 if (t == NULL || t->data == x) return t; // 如果当前节点为空那么返回NULL或者当前节点的值等于x 返回当前节点  
 if (cmp(x, t->data)) return find(x, t->left); // 如果x小于当前节点的值 那么在左子树中查找  
 else return find(x, t->right); // 否则在右子树中查找  
 }  
  
 // 判断是否包含元素  
 bool contain(const T &x) const {  
 return find(x) != NULL;  
 }  
  
 // 插入元素  
 bool insert(const T &x) {  
 if (root == NULL) { // 如果根节点为空 那么直接插入  
 root = new node(x); // 创建新节点  
 return true; // 创建新节点返回true  
 }  
 node \*t = root; // 否则从根节点开始查找  
 while (true) {  
 if (t->data == x) return false; // 如果有原来节点 那么返回false  
 if (cmp(x, t->data)) { // 如果x小于当前节点的值  
 if (t->left == NULL) { // 如果左子树为空  
 t->left = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->left; // 否则继续在左子树中查找  
 } else {  
 if (t->right == NULL) { // 如果右子树为空  
 t->right = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->right; // 否则继续在右子树中查找  
 }  
 }  
 }  
  
  
 bool remove(const T &x) {  
 node \*t = find(x);  
 if (t == NULL) return false;  
 if (t->left != NULL && t->right != NULL) {  
 node \*tmp = t->right;  
 while (tmp->left != NULL) tmp = tmp->left; //  
 t->data = tmp->data;  
 t = tmp;  
 }  
 node \*child = t->left != NULL ? t->left : t->right;  
 if (t == root) {  
 root = child;  
 if (root != NULL) root->parent = NULL;  
 } else {  
 node \*parent = t->parent;  
 if (parent->left == t) parent->left = child;  
 else parent->right = child;  
 if (child != NULL) child->parent = parent;  
 }  
 delete t;  
 return true;  
 }  
 };  
  
 // 求C风格字符串长度  
 size\_t strlen(const char \*s) {  
 size\_t len = 0;  
 while (s[len] != '\0') {  
 len++;  
 }  
 return len;  
 }  
  
 // 拷贝C风格字符串  
 void strcpy(char \*dest, const char \*src) {  
 size\_t i = 0;  
 while (src[i] != '\0') {  
 dest[i] = src[i];  
 i++;  
 }  
 dest[i] = '\0';  
 }  
  
 // 判断是否为空白字符  
 bool isspace(char c) {  
 return c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f' || c == '\v';  
 }  
  
 // 求最小值  
 template<typename T>  
 T min(const T &a, const T &b) {  
 return a < b ? a : b;  
 }  
  
 // 求最大值  
 template<typename T>  
 T max(const T &a, const T &b) {  
 return a > b ? a : b;  
 }  
  
 // 字符串类  
 class string {  
 public:  
  
 // 默认构造函数 构造一个不包含任何字符的字符串  
 string() : m\_data(new char[1]), m\_size(0), m\_capacity(1) {  
 m\_data[0] = '\0';  
 }  
  
 // 构造函数 从C风格字符串中构造字符串  
 explicit string(const char \*str) : m\_data(new char[my::strlen(str) + 1]), m\_size(my::strlen(str)),  
 m\_capacity(my::strlen(str) + 1) {  
 my::strcpy(m\_data, str);  
 }  
  
 // 拷贝构造函数  
 string(const string &other) : m\_data(new char[other.m\_capacity]), m\_size(other.m\_size),  
 m\_capacity(other.m\_capacity) {  
 my::strcpy(m\_data, other.m\_data);  
 }  
  
 // 析构函数  
 ~string() {  
 delete[] m\_data;  
 }  
  
 // 清空字符串  
 void erase() {  
 delete[] m\_data;  
 m\_data = new char[1];  
 m\_data[0] = '\0';  
 m\_size = 0;  
 m\_capacity = 1;  
 }  
  
 // 追加C风格字符串  
 void append(const char \*str, size\_t n) {  
 if (m\_size + n + 1 > m\_capacity) {  
 reserve(m\_size + n + 1);  
 }  
 my::strcpy(m\_data + m\_size, str);  
 m\_size += n;  
 m\_data[m\_size] = '\0';  
 }  
  
 // 复制构造函数  
 string &operator=(const string &other) {  
 if (this != &other) {  
 char \*temp = new char[other.m\_capacity];  
 my::strcpy(temp, other.m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_size = other.m\_size;  
 m\_capacity = other.m\_capacity;  
 }  
 return \*this;  
 }  
  
 // 追加字符串  
 string operator+(const string &other) const {  
 string newString;  
 newString.m\_size = m\_size + other.m\_size;  
 newString.m\_capacity = newString.m\_size + 1;  
 newString.m\_data = new char[newString.m\_capacity];  
 my::strcpy(newString.m\_data, m\_data);  
 my::strcpy(newString.m\_data + m\_size, other.m\_data);  
 return newString;  
 }  
  
 // 散列仿函数  
 class hash {  
 public:  
 size\_t operator()(const string &str) const {  
 size\_t hash = 0;  
 for (size\_t i = 0; i < str.size(); ++i) {  
 hash = hash \* 131 + str[i];  
 }  
 return hash;  
 }  
 };  
  
 // 相等运算符  
 bool operator==(const string &other) const {  
 if (m\_size != other.m\_size) { // 如果长度不相等 那么直接返回false  
 return false;  
 }  
 for (size\_t i = 0; i < m\_size; ++i) { // 否则逐个比较字符  
 if (m\_data[i] != other.m\_data[i]) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 bool operator!=(const string &other) const {  
 return !(\*this == other);  
 }  
  
 // 小于运算符  
 bool operator<(const string &other) const {  
 size\_t minSize = min(m\_size, other.m\_size); // 求最小长度  
 for (size\_t i = 0; i < minSize; ++i) {  
 if (m\_data[i] < other.m\_data[i]) { // 逐个比较字符  
 return true;  
 } else if (m\_data[i] > other.m\_data[i]) {  
 return false;  
 }  
 }  
 return m\_size < other.m\_size; // 如果前面的字符都相等 那么长度小的字符串小  
 }  
  
 // operator >  
 bool operator>(const string &other) const {  
 return other < \*this;  
 }  
  
 // operator <=  
 bool operator<=(const string &other) const {  
 return !(other < \*this);  
 }  
  
 // operator >=  
 bool operator>=(const string &other) const {  
 return !(\*this < other);  
 }  
  
 // 取出字符串中的字符  
 char &operator[](size\_t index) {  
 return m\_data[index];  
 }  
  
 // 取出字符串中的字符  
 const char &operator[](size\_t index) const {  
 return m\_data[index];  
 }  
  
 // 返回字符串长度  
 size\_t size() const {  
 return m\_size;  
 }  
  
 // 返回字符串容量  
 size\_t capacity() const {  
 return m\_capacity;  
 }  
  
 // 扩容  
 void reserve(size\_t new\_capacity) {  
 if (new\_capacity > m\_capacity) {  
 char \*temp = new char[new\_capacity];  
 my::strcpy(temp, m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_capacity = new\_capacity;  
 }  
 }  
  
 // 重新设置字符串长度  
 void resize(size\_t new\_size) {  
 if (new\_size > m\_capacity) {  
 reserve(new\_size);  
 }  
 for (size\_t i = m\_size; i < new\_size; ++i) {  
 m\_data[i] = '\0';  
 }  
 m\_size = new\_size;  
 }  
  
 // 在字符串末尾追加字符  
 void push\_back(char c) {  
 if (m\_size + 1 > m\_capacity) {  
 reserve(m\_capacity \* 2);  
 }  
 m\_data[m\_size] = c;  
 m\_data[m\_size + 1] = '\0';  
 ++m\_size;  
 }  
  
 // 返回C风格字符串  
 const char \*c\_str() const {  
 return m\_data;  
 }  
  
 // C++流输出运算符  
 friend std::ostream &operator<<(std::ostream &os, const string &str) {  
 os << str.m\_data;  
 return os;  
 }  
  
 // C++流输入运算符  
 friend std::istream &operator>>(std::istream &is, string &str) {  
 char c = '\0'; // 将str中的数据清空  
 while (is.get(c) && my::isspace(c)); // 跳过空白字符  
 if (is) { // 如果输入流正常  
 str.m\_size = 0; // 将str中的数据清空  
 do {  
 if (str.m\_size + 1 > str.m\_capacity) { // 如果空间不足 那么扩容  
 str.reserve(str.m\_capacity \* 2);  
 }  
 str.m\_data[str.m\_size] = c;  
 ++str.m\_size;  
 } while (is.get(c) && !my::isspace(c)); // 读取非空白字符  
 str.m\_data[str.m\_size] = '\0'; // 在末尾添加'\0'  
 if (is) is.unget(); // 如果输入流正常 那么将最后读取的字符放回输入流  
 else is.clear(is.rdstate() & ~std::ios\_base::failbit); // 否则清除错误标志  
 }  
 return is;  
 }  
  
 private:  
 char \*m\_data; // 字符串数据  
 size\_t m\_size; // 字符串长度  
 size\_t m\_capacity; // 字符串容量  
 };  
}  
  
struct stu {  
 my::string id;  
 int score;  
 int rank;  
 int sameRank;  
  
 stu() {}  
  
 stu(const my::string &id, int score, int rank, int sameRank)  
 : id(id), score(score), rank(rank), sameRank(sameRank) {}  
  
 bool operator==(const stu &rhs) const {  
 return id == rhs.id;  
 }  
  
 bool operator!=(const stu &rhs) const {  
 return !(rhs == \*this);  
 }  
  
 bool operator<(const stu &rhs) const {  
 return id < rhs.id;  
 }  
};  
  
int main() {  
 std::chrono::steady\_clock::time\_point start = std::chrono::steady\_clock::now();  
  
 std::ifstream data("P04\_TextData500000.in");  
 std::ifstream query("P04\_TextData500000b.in");  
 std::ofstream output("500000.out");  
 my::BinaryTree<stu> tree; // 创建二叉搜索树  
 stu s; // 创建学生对象  
 while (data >> s.id >> s.score >> s.rank >> s.sameRank) { // 读取数据  
 tree.insert(s); // 插入数据  
 }  
 my::string id; // 创建学号字符串  
 while (query >> id) { // 读取查询  
 auto it = tree.find(stu(id, 0, 0, 0)); // 查找学号  
 if (it != NULL) { // 如果找到  
 output << it->data.id << " " << it->data.score << " " << it->data.rank << " " << it->data.sameRank << "\n";  
 } else { // 否则输出not found  
 output << "not found" << std::endl;  
 }  
 }  
  
 data.close();  
 query.close();  
 output.close();  
 std::chrono::steady\_clock::time\_point end = std::chrono::steady\_clock::now();  
 std::cout << "Time: " << std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count() << "ms"  
 << std::endl;  
 return 0;  
}

四．结果说明

程序运行时间：

|  |  |  |  |
| --- | --- | --- | --- |
| 文件 | P04\_TextData500.out | P04\_TextData5000.out | P04\_TextData500000.out |
| 时间 | 1ms | 7ms | 1415ms |

程序运行结果：

|  |  |  |  |
| --- | --- | --- | --- |
| 文件 | P04\_TextData500.out | P04\_TextData5000.out | P04\_TextData500000.out |
| 正确率 | 100% | 100% | 100% |
| 输出行数 | 73 | 705 | 70797 |

五．结论和总结

该程序使用了二叉搜索树作为数据结构，用于存储学生数据，以便快速进行学号查询。

程序通过读取输入文件中的学生数据，并将其插入到二叉搜索树中。针对查询文件中的学号，程序能够快速找到对应的学生数据，并将查询结果输出到输出文件中。程序的运行时间取决于输入数据规模，但在合理的范围内，满足了要求。

总体而言，该程序成功完成了学号查询任务，具有较高的效率和正确性。