山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

学号：202200400053	姓名： 王宇涵	班级： 2202
上机学时：4	实验日期： 2024-04-10	

课程设计题目：

外排序

软件开发环境：

Clion 2023.1.1

报告内容：

1. 需求描述

1.1 问题描述

应用输者树结构模拟实现外排序。

1.2 基本要求

1. 设计并实现最小输者树结构 ADT，ADT 中应包括初始化、返回赢者，重构等基本操作。

2. 应用最小输者树设计实现外排序，外部排序中的生成最初归并串以及 K 路归并都应用最小输者树结构实现；

3. 验证你所实现的外排序的正确性。（1）随机创建一个较长的文件作为外排序的初始数据，设置最小输者树中选手的个数，验证生成最初归并串的正确性。获得最初归并串的个数及最初归并串文件，每一最初归并串使用一个文件。（2）使用以上生成的归并串，设置归并路数，验证 K 路归并的正确性。获得 K 路归并中各趟的结果，每一趟的结果使用一个文件。

*4. 获得外排序的访问磁盘次数，并分析其影响因素。

1.3 输入说明

输入界面设计

这里我们没有特定的界面设计，而是通过文件读入的方式进行数据的输入.

data.in 文件为外排序的初始数据，其中第 1 行为元素个数，第 2 行开始为元素数值， properties.txt 文件中给出最小输者树的大小(树中选手的个数)和归并路数。

输入样例

见测试样例，较长不进行列举。

1.4 输出说明

输出界面设计

这里我们没有特定的界面设计，而是通过文件输出的方式进行数据的输出.

output.txt 为最终排序结果，Segments 文件夹内为排序过程中产生的文件，如 Seg0-1.txt 为第 1 个初始顺串，Seg1-1.txt 为第 1 趟归并排序中产生的第一个结果文件。

输出样例

见测试样例，较长不进行列举。

2. 分析与设计

2.1 问题分析

本次问题是通过输者树结构模拟实现外排序。首先需要实现输者树该数据结构的 ADT 基础操作，其次需要模拟外排序的过程，通过文件的输入和输出操作来实现大量数据的排序，并通过动态生成大量数据来验证结果的正确性. 最后通过改变输者树大小，内存大小，k 路归并的 k，输入数据多少等变量，来探

究磁盘访问次数的不同影响因素.

2.2 主程序设计

我们设计 LoserTree.h, LoserTree.cpp, LoserNode.h, main.cpp 分别定义了最小输者树的类和函数实现, 外排序的具体模拟过程, 树结点的定义, 测试主函数, 最终通过测试函数中生成动态操作序列, 读入文件, 输出文件来分析外排序性能.

2.3 设计思路

主体思路是先实现基础功能, 再通过课程给定的样例进行测试功能的正确性, 最终自己实现动态操作序列, 并调整变量, 来观察排序性能的变化规律.

2.4 数据及数据类型定义

我们这里主要说明最小输者树的成员变量和模拟外排序的静态变量.

最小输者树的成员变量 ：

int MaxSize：表示树的最大容量。

int n：表示当前树的大小。

int LowExt：表示最低层的外部节点数。

int offset：表示树的偏移量。

int* t：表示败者树数组。

T* e：表示元素数组。

模拟外排序的静态变量 ：

int n, k : 表示最小输者树的大小和 k 路归并

const int MEMORY_SIZE = 5000 : 内存的大小

int bufferSize : 内存中缓冲区的大小

int outPutNum = 1;//标记初始化顺串的个数

int diskVisit = 0; // 磁盘访问次数

int randomNum = 0;// 初始大规模数据个数

2.5. 算法设计及分析

最小输者树算法

1. 初始化败者树 (Initialize)：

这个函数用于初始化败者树，首先根据输入的数组大小确定树的节点个数，然后根据节点数计算树的结构。接着进行一系列比赛，确保树中每个节点都存储了胜者的索引。最后对树进行一次层序遍历，将每个节点的胜者替换为对应的败者。

时间复杂度 O(n) n 为输者树大小。

2. 进行比赛 (Play)：

这个函数从指定节点开始进行比赛，比赛的过程是不断地更新父节点，直到根节点为止。每次比赛都会更新当前节点存储的胜者索引。

时间复杂度: O(logn) n 为输者树大小。

3. 重构胜者树 (RePlay)：

这个函数用于在胜者树已经初始化的情况下重新构建树，确保树中每个节点存储的是当前的胜者。它从胜者开始向上更新树的结构，直到根节点为止。

时间复杂度: O(logn) n 为输者树大小。

模拟外排序算法

1. 初始化顺串（initOrderings）：

首先从文件中读取数据，并将其分割为初始化顺串。

然后使用败者树进行初始化，将每个顺串的第一个元素作为初始化选手，构建初始的败者树。

在每次读入一组数据时，选取败者树的胜者作为输出，同时将相应顺串的下一个元素替换为新的选手，并重构败者树。

时间复杂度 O(n + mlogn) : n 为输者树大小, m 为数据的总个数。

2. 合并排序（mergeSort）：

采用了多路归并的思想，将多个初始化顺串进行合并排序。

使用多层归并树来完成合并操作，每层归并树的节点表示一个初始化顺串。

通过多次归并将初始化顺串逐步合并为有序序列。

时间复杂度 O(k + mlogn) : k 路归并, n 为输者树大小, m 为数据的总个数。

3. 测试

基础要求:

与测试样例进行对比，全部通过!

分析磁盘访问次数:

1. 固定 n 和 k, 内存大小, 改变初始数据的个数

```
n: 2000; k: 4; 初始数据个数为: 10739; 内存大小为: 5000; 初始顺串个数为: 4
磁盘访问次数为: 19
```

```
n: 2000; k: 4; 初始数据个数为: 12537; 内存大小为: 5000; 初始顺串个数为: 4
磁盘访问次数为: 21
```

```
n: 2000; k: 4; 初始数据个数为: 31354; 内存大小为: 5000; 初始顺串个数为: 9
磁盘访问次数为: 84
```

```
n: 2000; k: 4; 初始数据个数为: 68601; 内存大小为: 5000; 初始顺串个数为: 18
磁盘访问次数为: 255
```

可发现磁盘访问次数随着初始数据个数增大而增大

2. 固定 n , k, 初始数据的个数, 改变内存大小

```
n: 2000; k: 4; 初始数据个数为: 50000; 内存大小为: 1000; 初始顺串个数为: 14
磁盘访问次数为: 535
```

```
n: 2000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 14
磁盘访问次数为: 136
```

```
n: 2000; k: 4; 初始数据个数为: 50000; 内存大小为: 10000; 初始顺串个数为: 14
磁盘访问次数为: 86
```

可发现磁盘访问次数随着内存大小增大而减小

3. 固定 k，初始数据的个数，内存大小,改变 n

```
n: 1000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 218
```

```
n: 2000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 14
磁盘访问次数为: 136
```

```
n: 5000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 6
磁盘访问次数为: 115
```

```
n: 10000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 4
磁盘访问次数为: 58
```

可发现磁盘访问次数随着 n 大小增大而减小

4. 固定 n，初始数据的个数，内存大小,改变 k

```
n: 1000; k: 2; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 234
```

```
n: 1000; k: 3; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 186
```

```
n: 1000; k: 4; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 218
```

```
n: 1000; k: 10; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 287
```

```
n: 1000; k: 15; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 388
```

```
n: 1000; k: 20; 初始数据个数为: 50000; 内存大小为: 5000; 初始顺串个数为: 26
磁盘访问次数为: 487
```

可发现磁盘访问次数随着 k 增大而先减小后增大，如图 k = 3 左右取到极小值点，此时效率最好

总结 ：磁盘访问次数与初始数据个数成正相关，与内存大小，输者树大小成负相关，随 k 增大先减小后增大，其中 k 有权衡性能的最优值.

K 变化原因分析 ：k 很小时归并趟数会很大，而 k 很大时内存的缓冲区会很小.

4. 分析与探讨

本次实验我通过学习最小输者树的数据结构，了解它的思想，并实现了基础功能，通过这个有趣的数据结构成功完成了外排序的模拟，通过了测试样例.

此外，我也成功通过自己生成动态序列，进行文件的输入输出操作，改变输入的变量，成功简洁地完成了外排序磁盘访问次数的性能测量与分析.

遇到的问题如下

1. 生成随机数发现生成的数都在几万的范围内

解决 ： 通过 uniform_int_distribution 函数实现了随机数均匀的产生和分布.

2. 外排序的实现模拟比较复杂，发现代码冗长

解决 ： 及时进行封装，将函数简化，理顺逻辑.

3. **自定义随机生成数据，发现生成数据没有规律性.**

解决 ： 发现初始数据规模设置太小，设置到较大的一个数如 50000，则发现了较好的规律

5. 附录：实现源代码

最小输者树实现

初始化

```
template<class T>
void LoserTree<T>::Initialize(T a[], int size,
                              int(*winner)(T a[], int b, int c))
{// 初始化败者树
    if (size > MaxSize || size < 2)
        return;
    //赋值大小, treeNode 数组
    n = size;
    e = a;

    int i, s;
    for (s = 1; 2*s <= n-1; s += s);        // 计算 s = 2^log (n-1)

    LowExt = 2*(n-s); // 最底层外部节点个数
    offset = 2*s-1;

    // 首先记录 t[1:n-1]中的胜者
    // 对于最低层的外部节点找右孩子进行比赛
    for (i = 2; i <= LowExt; i += 2)
        Play((offset+i)/2, i-1, i, winner); // 前者为父亲节点

    // 处理剩余的外部节点
    if (n % 2) {// 对于奇数 n 的特殊情况，进行比赛
        // 内部和外部节点
        Play(n/2, t[n-1], LowExt+1, winner);
        i = LowExt+3;}
    else i = LowExt+2;

    // i 是最左边剩余的外部节点
```

```
    for (; i <= n; i += 2)
        Play((i-LowExt+n-1)/2, i-1, i, winner);

    // 在 t[0]中记录总体赢家
    t[0] = t[1];

    // 现在对 t 进行层序遍历
    // 将胜者替换为败者
    for (i = 1; i < n; i++) {
        // 将 t[i]设置为在 t[i]中进行比赛的失败者
        int lc = 2 * i;     // i 的左孩子
        int rc = lc + 1;    // i 的右孩子
        // 最终 e[LeftPlayer] 表示比赛的左方选手
        // t[i] 和 e[RightPlayer] 表示另一位选手
        int LeftPlayer, RightPlayer;
        // 确定 LeftPlayer
        if (lc <= n - 1) LeftPlayer = t[lc];
        else // 左孩子是外部节点
        if (lc <= offset)
            LeftPlayer = lc + LowExt - n + 1;
        else LeftPlayer = lc - offset;

        // 确定 RightPlayer
        if (rc <= n - 1) RightPlayer = t[rc];
        else // 右孩子是外部节点
        if (rc <= offset)
            RightPlayer = rc + LowExt - n + 1;
        else RightPlayer = rc - offset;

        // 确定比赛的失败者
        if (LeftPlayer == t[i])
            // RightPlayer 是失败者
            t[i] = RightPlayer;
        else // LeftPlayer 是失败者
            t[i] = LeftPlayer;
    }

}
```

重构

```
template<class T>
void LoserTree<T>::RePlay(int(*winner)(T a[], int b, int c)) // 重构进行之前的胜者树，确保败者树已经初始化
{
    if (n < 2) return;
    int p;     // 比赛节点 (父节点)
```

```
    // 找到第一个比赛节点
    int i = t[0];      // i 是之前的胜者的标号
    if (i <= LowExt)
        p = (offset + i)/2;// 从最低层开始
    else p = (i-LowExt+n-1)/2;

    int LastWinner = i;

    // 进行比赛
    for (; p >= 1; p /= 2) {
        // 在 t[p] 进行比赛
        int NewWinner = winner(e, LastWinner, t[p]);
        // 如果胜者有变化，则更新败者
        if (t[p] == NewWinner) {
            // e[t[p]] 不再是败者
            t[p] = LastWinner;
            LastWinner = NewWinner;}
    }

    // 将总体胜者放入 t[0]
    t[0] = LastWinner;
}
```

比赛

```
template<class T>
void LoserTree<T>::Play(int p, int lc, int rc, int(*winner)(T a[], int b, int c))
{// 从 t[p]开始比赛
    // lc 和 rc 是 t[p] 的孩子
    t[p] = winner(e, lc, rc);

    // 如果在 p 在右孩子处还有比赛
    while (p > 1 && p % 2) {// 在右孩子处
        t[p/2] = winner(e, t[p-1], t[p]);
        p /= 2;   // 移动到父节点
    }
}
```

外排序过程实现

初始化顺串

```
void initOrderings() {
    //从 data.txt 中截取数据
    int allNums, readSize;
    ifstream finData("testData.txt", ios::in);
    finData >> allNums;
    readSize = min (n, allNums); // 读取数据的个数，这里我们默认是小于内存大小的，因为内存大小 >
n
```

```
//读入一组数据初始化输者树
int * data = NULL;
LoserTree <LoserNode> lt (n);
readSizeData(data, readSize, finData);
allNums -= readSize;
LoserNode * players = new LoserNode[n];
for (int i = 1 ; i <= n ; i++) {
	players[i] = LoserNode(1, data[i]); // //ID = 1, key = data[i]
}
lt.Initialize(players, n, winner);

//输出到新的文件中
ofstream fOutOrder("Seg0-1.txt", ios::out);
int serialNumber = 1; // 初始顺串号
//开始读
// 1000 -> 700 300
while (allNums > 0) {
	//再读入一组数据
	readSizeData(data, readSize, finData);
	// 每次都读取 readSize 个数据，并且输出 readSize 个数据
	for (int i = 1; i <= readSize; i++) {
		int winnerIdx = lt.Winner();
		int winnerSerial = players[winnerIdx].ID;
		//判断此时是否需要重开一个文件输出
		if (serialNumber < winnerSerial ) {
			fOutOrder.close();
			serialNumber++;
			string initOutPutOrdering = "Seg0-1.txt";
			string outPutOrdering;
			outPutOrdering = initOutPutOrdering .replace
					(initOutPutOrdering.find("0-1"), 3, "0-" + to_string(++outPutNum));
			fOutOrder.open(outPutOrdering);
		}
		fOutOrder << players[winnerIdx].key <<" ";// 输出到目标文件中去
		//判断下一个玩家的顺串类型，并将 winner 替换为该玩家
		if (data[i] >= players[winnerIdx].key) { // 不是没有的数据
			players[winnerIdx] = {winnerSerial, data[i]};
		}
		else {
			players[winnerIdx] = {winnerSerial + 1, data[i]};
		}
		lt.RePlay(winner);
	}
	//读入完数据进行更新
	allNums -= readSize;
```

```
            readSize = min (readSize, allNums);
        }

        //此时输出树中还有初始化的数据没有输出
        for (int i = 1; i <= n; i++) {
            int winnerIdx = lt.Winner();
            int winnerSerial = players[winnerIdx].ID;
            //判断此时是否需要重开一个文件输出
            if (serialNumber < winnerSerial ) {
                fOutOrder.close();
                serialNumber++;
                string initOutPutOrdering = "Seg0-1.txt";
                string outPutOrdering;
                outPutOrdering = initOutPutOrdering .replace
                        (initOutPutOrdering.find("0-1"), 3, "0-" + to_string(++outPutNum));
                fOutOrder.open(outPutOrdering);
            }
            fOutOrder << players[winnerIdx].key <<" ";// 输入对应的赢者(最小值)
            players[winnerIdx] = LoserNode(INT_MAX, INT_MAX); // 此时更新为最大的，保证不会输出
            lt.RePlay(winner);
        }

        finData.close();
}
```

归并数据

```
void merge2File(int ** & data, int Size, ofstream & outMerge) {
        int idx[Size + 1];// 记录每个顺串需要取数的位置
        bool st[Size + 1]; //判断每个顺串是否算过了磁盘访问次数
        for (int i = 1 ; i <= Size ; i++) {
            idx[i] = 1; st[i] = false;
        } // 初始化为 1
        //特殊情况，无法初始化输者树，直接输出即可
        if (Size == 1) {
            int cnt = 1;
            while (data[Size][cnt]!= 0x3f3f3f3f) {
                outMerge << data[Size][cnt++] << " ";
            }
            diskVisit += ceil(double(cnt - 1) / bufferSize);
            return;
        }

        //初始化输者树
        LoserNode *players = new LoserNode [Size + 1];
        for (int i = 1 ; i <= Size; i++) {
            players[i] = LoserNode(1, data[i][idx[i]++]); // 所有顺串都标记为 1
```

```
    }
    LoserTree<LoserNode>    lt(Size);
    lt.Initialize(players, Size, winner);

    int winnerIdx = lt.Winner();
    //每次进行比赛，替换掉赢者
    while (players[winnerIdx].key != 0x3f3f3f3f) {
        outMerge << players[winnerIdx].key << " ";
        int nextOne = data[winnerIdx][idx[winnerIdx]++];// 赢者对应的下一个顺串
        if (!st[winnerIdx] && nextOne == 0x3f3f3f3f)  {// 读到了末尾，取出此时的 idx，来判断需要从内存中读几次
            diskVisit += ceil(double(idx[winnerIdx] - 1) / bufferSize); //需要多读 该顺串的长度 / bufferSize 向上取整次
            st[winnerIdx] = true;
        }
        players[winnerIdx].key = nextOne; // 从赢者对应的顺串中取数据
        //如果一个顺串已经读完了
        lt.RePlay(winner); // 重构树
        winnerIdx = lt.Winner();
    }
}
```

外排序

```
void mergeSort() {
    bufferSize = MEMORY_SIZE / (k + 1);
    int round = ceil(log(outPutNum) / log(k)); //归并树的高度, (5 / log4) = 2
    int cur = outPutNum, next; // 记录此时遍历的文件总个数和即将生成的文件个数
    for (int i = 1; i <= round ; i++) {
        int inPutNum = 1; // 记录输入文件名的号码
        int outPutMergeNum = 1;//记录输出文件名的号码
        next = ceil(double (cur) / k); // 记录即将生成的文件个数, ceil 5 / 4 = 2
        //表示输出的文件为第 j 个
        for (int j = 1 ; j <= next; j++) {
            //确定输出文件名
            string mergeFile ;
            if (i == round) {// 修改输出文件名
                mergeFile = "output.txt";
            }
            else {
                //确定输出文件名 1-1 / 1-2 / 1-3..
                string initMergeFile = "Seg1-1.txt";
                mergeFile = initMergeFile.replace
                        (initMergeFile.find("1-1"), 3, to_string(i) + "-" + to_string(outPutMergeNum++));
            }
            ofstream ofMerge (mergeFile, ios::out); // 打开输出文件
```

```
                //确定输入文件名
                int Size = min (cur, k);
                cur -= k;
                //找到对应的输入串  seg0-1 0-2 0-3 , seg1-1 seg1-2 ..  并将文件的数据存入 data 数组中
                int ** data = new int * [Size + 1]; // data[i]表示第 i 个顺串
                for (int s = 1 ; s <= Size ; s++) {
                        data[s] = NULL;
                } //  初始化
                for (int s = 1; s <= Size; s++) {
                        string initOrdering = "Seg0-1.txt";
                        string inputOrdering =initOrdering.replace
                                        (initOrdering.find("0-1"), 3, to_string(i - 1) + "-" + to_string(inPutNum++));
                        ifstream ifOrder (inputOrdering, ios::in); //  打开输入文件
                        readNoSizeData(data[s], ifOrder); //  每次都读取 n 个数据
                }
                //已经得到了 data 数组，此时进行 merge 操作
                merge2File(data, Size, ofMerge);
            }
            cur = next;
        }
}
```