第8章 文件管理和外排序

任课教员：张铭、赵海燕、冯梅萍、王腾蛟
http://db.pku.edu.cn/mzhang/DS/
北京大学信息科学与技术学院

为什么需要文件管理和外排序？

- 文件结构(file structure)
  - 对于在外存中存储的数据，其数据结构就称为文件结构(file structure)
  - 数据量太大不可能同时把它们放到内存中
  - 需要把全部数据放到磁盘中
- 文件的各种运算
  - 外排序是针对磁盘文件所进行的排序操作
  - 提高文件存储效率和运算效率

本章的安排顺序

- 8.1 介绍主存和外存的根本差异
- 8.2 在外存中文件的组织方式
- 8.3 管理缓冲池的基本方法
- 8.4 外排序的基本算法

主存储器和外存储器

- 主存储器(primary memory或者main memory，简称“内存”，或者“主存”)
  - 随机访问存储器(Random Access Memory, 即RAM)
  - 高速缓存(cache)
  - 视频存储器(video memory)。
- 外存储器(peripheral storage或者secondary storage，简称“外存”)
  - 硬盘
  - 软盘
  - 磁带

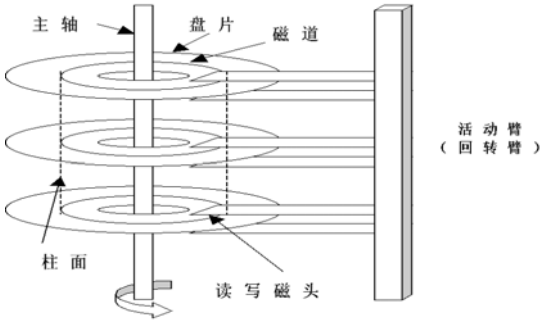
主存储器和外存储器之价格比较

介质	2001年底价格	2002年底价格	2003年早期价格
内存	1	1.5	1
硬盘	0.017	0.013	0.011
软盘	12	7	2.5
磁带	0.008	0.011	0.0075

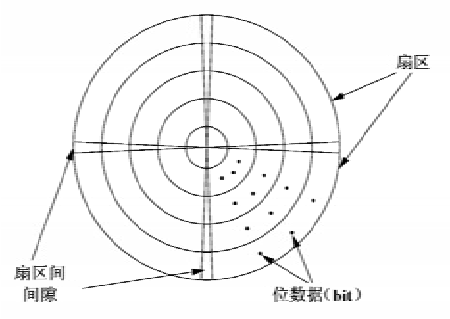
外存的优缺点

- 优点：永久存储能力、便携性
- 缺点：访问时间长
  - 访问磁盘中的数据比访问内存慢五六个数量级。
- 所以讨论在外存的数据结构及其上的操作时，必须遵循下面这个重要原则：
  - 尽量减少访外次数！

磁盘的物理结构



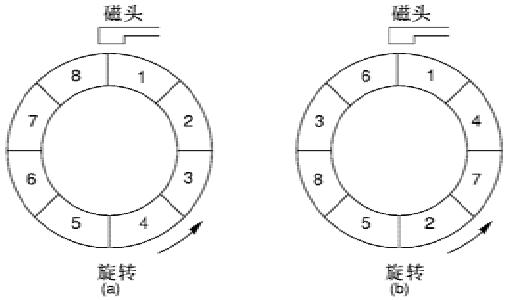
磁盘盘片的组织



磁盘存取步骤

- 选定某个盘片组
- 选定某个柱面
  - 这需要把磁头移动到该柱面，这个移动过程称为寻道(seek)
- 确定磁道
- 确定所要读写的数据在磁盘上的准确位置
  - 这段时间一般称为旋转延迟(rotational delay 或者rotational latency)
- 真正进行读写

磁盘磁道的组织（交错法）



（a）没有扇区交错；（b）以3为交错因子

磁盘访问时间估算

- 磁盘访问时间主要由寻道时间、旋转延迟时间和数据传输时间组成。
- 寻道时间（Seek time）：是移动磁盘臂，定位到正确磁道所需的时间。
- 旋转延迟时间：是等待被存取的扇区出现在读写头下所需的时间。

总存取时间(1)

（1）数据连续存放，而且给出了平均寻道时间。

总存取时间 = [平均寻道时间]
+ [第一道读取时间]
+ (总磁道数–1)×[(第二次寻道时间)+(读取整道的时间)]
= [平均寻道时间]
+ [(0.5圈延迟+交错因子) × 每圈所花时间]
+ (总磁道数–1)
× [磁道转换时间+(0.5圈延迟+交错因子)×每圈所花时间]

总存取时间(2)

（2）数据随机存放。

总存取时间 = 簇数 × {[平均寻道时间]+[旋转延迟]+[读一簇时间]}

= 簇数 × {[平均寻道时间]

+ [0.5圈延迟×每圈所花时间]

+ [交错因子×（每簇扇区数/每道扇区数）×每圈时间]}

- 例8.1 假定一个磁盘总容为16.8GB，分布在10个盘片上。每个盘片上有13085个磁道，每个磁道中包含256个扇区，每个扇区512个字节，每个簇8个扇区。扇区的交错因子是3。磁盘旋转速率是5400 rpm，磁道转换时间是2.2 ms，随机访问的平均寻道时间是9.5 ms。

- 如果读取一个1MB的文件，该文件有2048个记录，每个记录有512字节（1个扇区）。对于以下两种情况，估算进行文件处理的总时间。
  - （1）假定所有记录在8个连续磁道上；
  - （2）假定文件簇随机地散布在磁盘上。

解：我们先总结出一些共有的特性

- 每个簇为4KB
  - 8个扇区 = 512 byets × 8 = 4KB
- 每个磁道有32簇
  - 256个扇区 = 256 扇区 ÷8（扇区/簇） = 32个簇
- 每个盘片有1.68GB
  - 16.8GB ÷ 10 = 1.68GB
- 每转一圈的时间为11.1ms
  - (60×1000)ms/5400圈 = 11.1（ms/圈）= 11.1（ms/道）
  - 一个磁道是一个整圈，因此，下面计算公式中，所用的单位“圈”等同于“道”

- 数据连续存放，而且给出了平均寻道时间。

总存取时间 = [平均寻道时间]

+ [(0.5圈延迟+交错因子) × 每圈所花时间]

+ (总磁道数−1)

× [磁道转换时间+(0.5圈延迟+交错因子)×每圈所花时间]

= [9.5ms（平均寻道时间）]

+ [(0.5（半圈旋转延迟） + 3（交错因子）)×11.1ms/圈

+（8-1）个其他磁道

× [2.2ms磁道转换时间+(0.5圈延迟+3交错因子)×11.1ms/圈]

= 9.5ms + 38.9ms + 7 × 41.1ms

= 336.1ms

- 数据随机存放
- 总簇数：8个磁道 × 32(簇/磁道) = 256簇

总存取时间 = 簇数 × {[平均寻道时间]

+ [0.5圈延迟×每圈所花时间]

+ [交错因子×（每簇扇区数/每道扇区数）×每圈时间]}

= 256 ×{ [9.5ms（平均寻道时间）]

+ [0.5（半圈旋转延迟）×11.1ms/圈]

+ [3(交错因子)×8(扇区/簇/)÷256(扇区/道) ×11.1ms/圈]}

≈ 256 × {9.5 + 5.55 + 1.04}ms

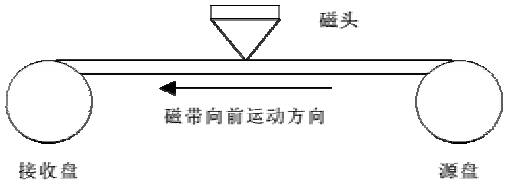
≈ 4119.04ms

- 读一簇的时间
  - [3(交错因子)×8(扇区/簇/)÷256(扇区/道) ×11.1ms/圈]
    ≈ 1.04ms
- 读一个扇区的时间
  - [9.5ms(平均寻道时间)]+[0.5(半圈旋转延迟)×11.1ms/圈+[1扇区÷256(扇区/道) ×11.1ms/圈]
    ＝ 15.1ms
- 读一个字节的时间
  - [9.5ms(平均寻道时间)]+[0.5(半圈旋转延迟)×11.1ms/圈]
    ＝ 15.05ms
- “一次访外”操作
  - 磁盘一次I/O操作访问一个扇区，通常称为访问“一页”（page），或者“一块”(block)

磁带

- 主要优点 ：使用方便、价格便宜、容量大、所存储的信息比磁盘和光盘更持久
- 缺点：速度较慢，只能进行顺序存取

磁带运行示意图



磁带卷在一个卷盘上，运行时磁带经过读写磁头，把磁带上的信息读入计算机，或把计算机中的信息写在磁带上。

磁带发展史

- 手工阶段
- 自动化磁带库系统
- 虚拟磁带技术

外存文件的组织

职工号	姓名	性别	职务	婚姻状况	工资
156	张东	男	程序员	未婚	7800
860	李珍	女	分析员	已婚	8900
510	赵莉	女	程序员	未婚	6900
950	陈萍	女	程序员	未婚	6200
620	周力	男	分析员	已婚	10300

- 文件是一些记录的汇集，其中每个记录由一个或多个数据项组成。
- 数据项有时也称为字段，或者称为属性。例如，一个职工文件里的记录可以包含下列数据项：职工号，姓名，性别，职务，婚姻状况，工资等。

文件组织

- 逻辑文件
  - 对高级程序语言的编程人员而言，存储在磁盘中可以随机访问的文件被当作一段连续的字节，而且可以把这些字节结合起来构成记录，这种文件被称为逻辑文件(logical file)。
- 物理文件
  - 实际存储在磁盘中的物理文件(physical file)通常不是一段连续的字节，而是成块地分布在整个磁盘中。
- 文件管理器
  - 是操作系统的一部分，当应用程序请求从逻辑文件中读解数据时，它把这些逻辑位置映射为磁盘中具体的物理位置。

文件的逻辑结构

- 文件是记录的汇集。
  - 当一个文件的各个记录按照某种次序排列起来时，各纪录间就自然地形成了一种线性关系。
- 因而，文件可看成是一种线性结构。

文件的存储结构

- 顺序结构——顺序文件
- 计算寻址结构——散列文件
- 带索引的结构——带索引文件

文件上的操作

- 检索：在文件中寻找满足一定条件的记录
- 修改：是指对记录中某些数据值进行修改。若对关键码值进行修改，这相当于删除加插入。
- 插入：向文件中增加一个新记录。
- 删除：从文件中删去一个记录 。
- 排序 ：对指定好的数据项，按其值的大小把文件中的记录排成序列，较常用的是按关键码值的排序。

C＋＋的流文件

- C++程序员对随机访问文件的逻辑视图是一个单一的字节流，即字节数组。
- 程序员需要管理标志文件当前位置的文件指针。
- 常见的三个基本文件操作，都是围绕文件指针进行的：
  - 把文件指针设置到指定位置（移动文件指针）
  - 从文件的当前位置读取字节
  - 向文件中的当前位置写入字节

C＋＋操作二进制文件的常用方式——fstream类

- fstream类提供函数操作可随机访问的磁盘文件中的信息。
- fstream类的主要成员函数包括open，close，read，write，seekg，seekp。

fstream类的主要函数成员

```
#include<fstream.h>//fstream=ifstream+ofstream

// 打开文件进行处理。
// 模式示例: ios::in | ios::binary
void fstream::open(char* name, openmode mode);

// 处理结束后关闭文件。
void fstream::close();
```

ftream类的主要函数成员（续1）

```
// 从文件当前位置读入一些字节。
// 随着字节的读入，文件当前位置向前移动。
fstream::read(char* ptr, int numbytes);

// 向文件当前位置写入一些字节
// (覆盖已经在这些位置的字节)。
// 随着字节的写入，文件当前位置向前移动。
fstream::write(char* ptr, int numbtyes);
```


ftream类的主要函数成员（续2）

```
// seekg和seekp：在文件中移动当前位置，
// 这样就可以在文件中的任何一个位置
// 读出或写入字节。
// 实际上有两个当前位置，
//一个用于读出，另一个用于写入。
// 函数seekg用于改变读出位置，
// 函数seekp用于改变写入位置
fstream::seekg(int pos);  // 处理输入
fstream::seekg(int pos, ios::curr);
fstream::seekp(int pos);  // 处理输出
  fstream::seekp(int pos, ios::end);
```


缓冲

- 目的：减少磁盘访问次数的
- 方法：缓冲(buffering)或缓存(caching)
  - 在内存中保留尽可能多的块
  - 可以增加待访问的块已经在内存中的机会，因此就不需要访问磁盘

缓冲区和缓冲池

- 存储在一个缓冲区中的信息经常称为一页(page)，往往是一次I/O的量
- 缓冲区合起来称为缓冲池(buffer pool)

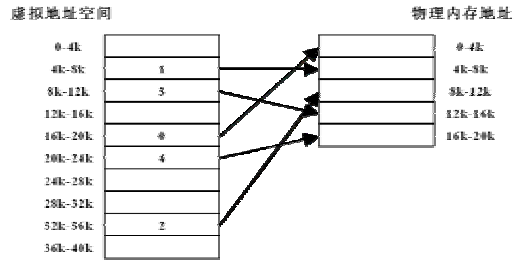
替换缓冲区块的策略

- “先进先出”(FIFO)
- “最不频繁使用”(LFU)
- “最近最少使用”(LRU)

虚拟存储 (virtual memory)

- 虚拟存储使得程序员能够使用比实际内存更大的内存空间。
- 磁盘中存储虚拟存储器的全部的内容，根据存储器的访问需要把块读入内存缓冲池。

虚拟存储



- 系统运行到某一时刻，虚拟地址空间中有5个页被读入到物理内存中，现在如果需要对地址为24k-28k的页进行访问，就必须替换掉当前物理内存中的一个页框。

外排序

- 外排序(external sort)
  - 外存设备上(文件)的排序技术
  - 由于文件很大，无法把整个文件的所有记录同时调入内存中进行排序，即无法进行内排序
- 外部排序算法的主要目标是尽量减少读写磁盘的次数

关键码索引排序

- 索引文件(index file)中
  - 关键码与指针一起存储
  - 指针标识相应记录在原数据文件中的位置
- 对索引文件进行排序，所需要的I/O操作更少
  - 索引文件比整个数据文件小很多。
  - 在一个索引文件中，只针对关键码排序(key sort)。

磁盘排序过程

- 顺串：用某种有效的内排序方法对文件的各段进行初始排序，这些经过排序的段通常称为顺串(run)，它们生成之后立即被写回到外存上。

磁盘排序过程：

- 文件分成若干尽可能长的初始顺串；
- 逐步归并顺串归，最后形成一个已排序的文件。

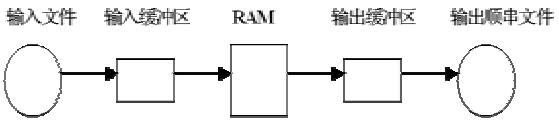
置换选择排序

- 采用置换选择算法，在扫描一遍的前提下，使得所生成的各个顺串有更大的长度。这样减少了初始顺串的个数，有利于在合并时减少对数据的扫描遍数。

置换选择算法

- 假设外排序算法可以使用一个输入缓冲区、一个输出缓冲区、一个能存放M个记录的RAM内存块（可以看成大小为M的连续数组）。排序操作可以利用缓冲区，但只能在RAM中进行。
- 平均情况下，这种算法可以创建长度为2M个记录的顺串。

置换选择方法图示



处理过程为：从输入文件读取记录（一整块磁盘中所有记录），进入输入缓冲区；然后在RAM中放入待排序记录；记录被处理后，写回到输出缓冲区；输出缓冲区写满的时候，把整个缓冲区写回到一个磁盘块。当输入缓冲区为空时，从磁盘输入文件读取下一块记录。

置换选择算法

1. 初始化堆：

(a) 从磁盘读M个记录放到数组RAM中。

(b) 设置堆末尾标准LAST ＝ M － 1。

(c) 建立一个最小值堆。

置换选择算法（续）

2. 重复以下步骤，直到堆为空（即LAST < 0）：

(a) 把具有最小关键码值的记录(根结点)送到输出缓冲区。

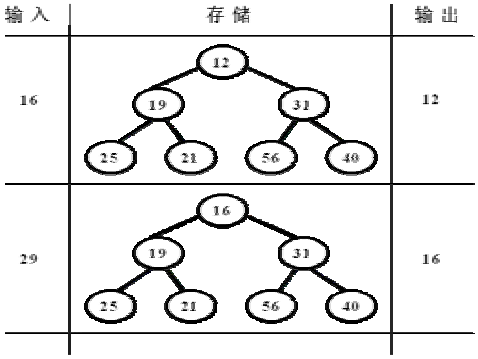
(b) 设R是输入缓冲区中的下一条记录。如果R的关键码值大于刚刚输出的关键码值……

i．那么把R放到根结点。

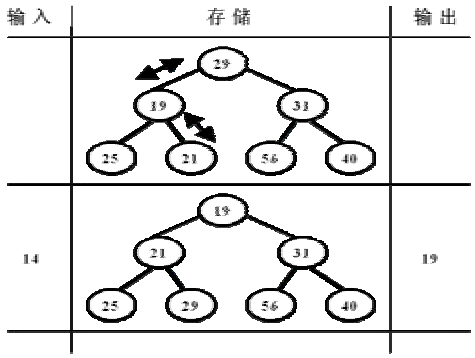
ii．否则使用数组中LAST位置的记录代替根结点，然后把R放到LAST位置。设置LAST ＝ LAST － 1。

(c)重新排列堆，筛出根结点。

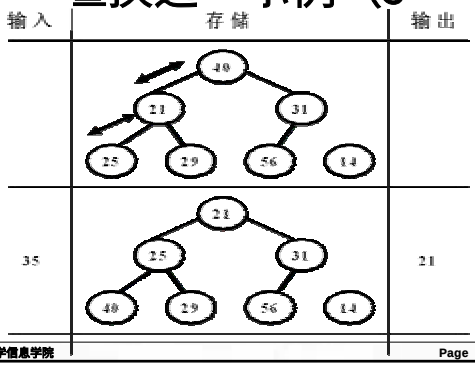
置换选择示例（1）



置换选择示例（2）



置换选择示例（3）



置换选择算法的实现

```
////置换选择算法
//模板参数Elem代表数组中每一个元素的类型
//A是从外存读入n个元素后所存放的数组，n是数组元素的个数
//in和out分别是输入和输出文件名
template <class Elem>
void replacementSelection(Elem * A, int n, const char * in, const char * out){
```


置换选择算法的实现（续1）

```
    Elem mval;  //存放最小值堆的最小值
    Elem r;  //存放从输入缓冲区中读入的元素
    //输入、输出文件句柄
    FILE * inputFile;
    FILE * outputFile;
   //输入、输出buffer
    Buffer<Elem> input;
    Buffer<Elem> output;
  //初始化输入输出文件
    initFiles(inputFile, outputFile, in, out);
```


置换选择算法的实现（续2）

```
/********** 算法主体部分 ***********/

//初始化堆的数据，
// 从磁盘文件读入n个数据置入数组A
initMinHeapArry(inputFile, n, A);

//建立最小值堆
  MinHeap<Elem> H(A, n, n);
```


置换选择算法的实现（续3）

```
//初始化input buffer，读入一部分数据
  initInputBuffer(input, inputFile);

for(int last = (n-1); last >= 0;){
//堆不为空，就做这个循环
     mval = H.heapArray[0];  //堆的最小值
//把mval送到输出缓冲区，
//同时处理因缓冲区空或满造成的各种情形
  sendToOutputBuffer(input, output, inputFile,
  outputFile, mval);
```


置换选择算法的实现（续4）

```
input.read(r); //从输入缓冲区读入一个记录
if(!less(r, mval)){
//若r的关键码值大于等于刚才输出缓冲区记录的关键码值，
   //把r放到根结点
              H.heapArray[0] = r;
        }
else {  //否则用last位置的记录代替根结点，把r放到last位置
              H.heapArray[0] = H.heapArray[last];
              H.heapArray[last] = r;
              H.setSize(last);
              last--;
        }
```


置换选择算法的实现（续5）

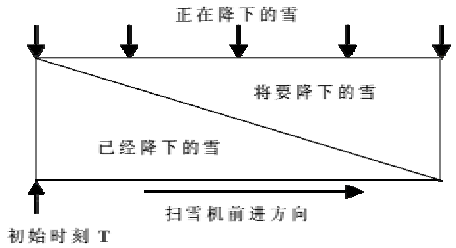
```
  H.SiftDown(0);   //把根结点记录下降到合适的位置
  }  //for
//算法结束工作：
//处理输出缓冲区，输入/输出文件
  endUp(output, inputFile, outputFile);
}
```

置换选择算法的效果

如果堆的大小是M

- 一个顺串的最小长度就是M个记录
  - 至少原来在堆中的那些记录将成为顺串的一部分
- 最好的情况下，例如输入已经被排序，有可能一次就把整个文件生成为一个顺串。

扫雪机模型

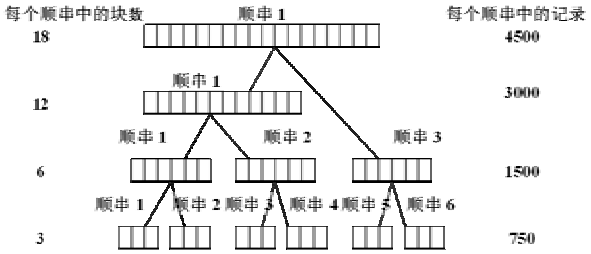


- 扫雪机模型显示扫雪机在环绕一圈过程中的行为。根据“扫雪机”模型的分析，可以预计平均情况下顺串总长度是数组长度的两倍。

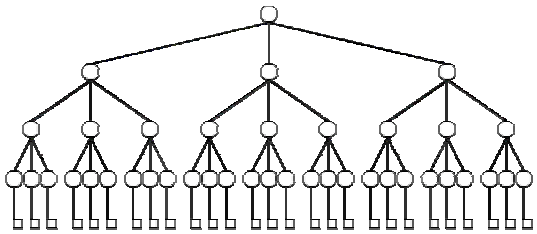
二路外排序

- 归并原理：把第一阶段所生成的顺串加以合并(例如通过若干次二路合并)，直至变为一个顺串为止，即形成一个已排序的文件。

二路归并外排序



多路归并



三路归并

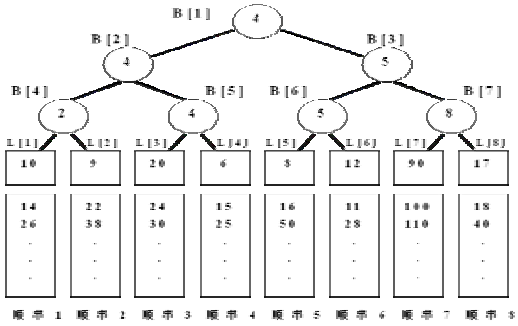
选择树

- 在K路归并中，最直接的方法就是作K-1次比较来找出所要的记录，但这样做花的代价较大，我们采用选择树的方法来实现K路归并。
- 选择树是完全二叉树，有两种类型：赢者树和败方树。

赢者树

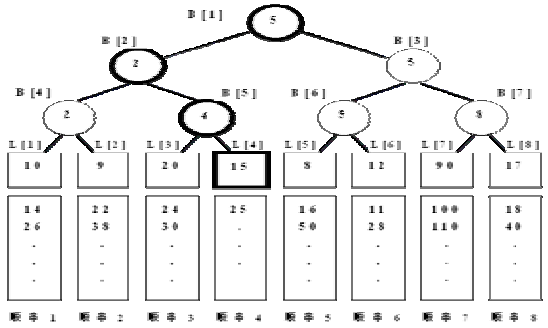
- 叶子节点用数组L表示
  - 代表各顺串在合并过程中的当前记录（图中标出了它们各自的关键码值）；
- 分支节点用数组B表示
  - 每个分支节点代表其两个儿子结点中的赢者(关键码值较小的)所对应数组L的索引。
- 因此，根结点是树中的最终赢者的索引，即为下一个要输出的记录结点。

赢者树示例（1）



根结点所指向的L[4]记录具有最小的关键码值6，它所指的记录是顺串4的当前记录，该记录即为下一个要输出的记录。

赢者树示例（2）

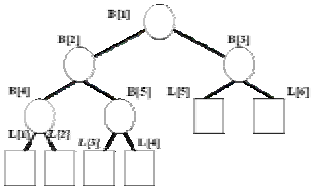


重构后的赢者树，改动的节点用较粗的框显示出来。为了重构这颗树，只须沿着从结点L[4]到根结点的路径重新进行比赛。

赢者树ADT

```
template<class T>
class WinerTree{
public:
Create();                    //创建一个空的赢者树
  Iitialize(T a[], int n);   //返回比赛的赢者
  Replay(i);         //选手i变化时，重建赢者树
};
```


赢者树与数组的对应关系



外部节点的数目为n，LowExt代表最底层的外部节点数目；offset代表最底层外部节点之上的所有节点数目。每一个外部节点L[i]所对应的内部节点B[p]，i和p之间存在如下的关系：

$$p = \begin{cases} (i + offset)/2 & i \le LowExt \\ (i - LowExt + n - 1)/2 & i > LowExt \end{cases}$$

赢者树的类定义

```
template<class T>
class WinnerTree{
  private:
      int MaxSize;  //允许的最大选手数
      int n;          //当前大小
      int LowExt;   //最低层的外部节点数
      int offset;   //2^(s+1)－1
      int * B;      //内部节点数组
      T * L;                //外部节点数组
void Play(int p, int lc, int rc, int(* winner)(T A[], int b, int c));
```

赢者树的类定义（续）

```
public:
WinnerTree(int Treesize = MAX);
  ~WinnerTree(){delete [] B;}

  void Initialize(T A[], int size,int (*winner)(T A[], int
  b, int c)); //初始化赢者树
  int Winner();//返回最终胜者的索引
  void RePlay(int i, int(*winner)(T A[], int b, int c));
  //位置i的外部选手改变后重构赢者树
};
```

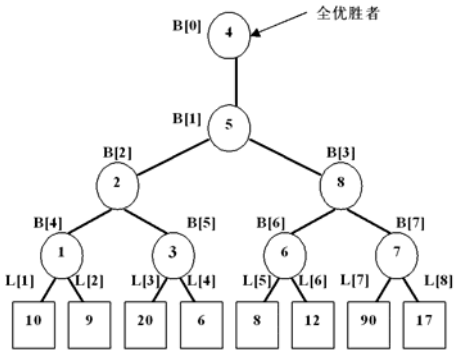

败方树

- 所谓败方树就是在选择 (比赛)树中，每个非叶结点均存放其两个子结点中的败方所对应叶子节点的索引值。
- 此外，还另加进一个结点，即结点B[0]，以代表比赛的全局获胜者的索引值。

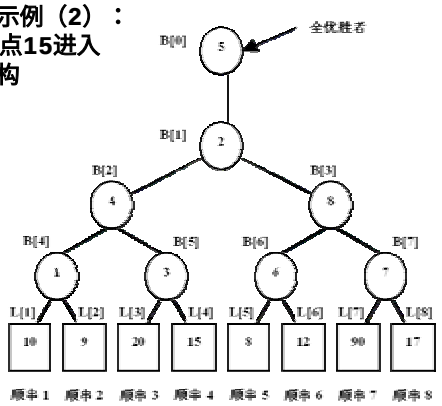
败方树比赛过程

- 将新进入树的结点与其父结点进行比赛
  - 把败者存放在父结点中
  - 而把胜者再与上一级的父结点进行比赛
- 这样的比赛不断进行，直到结点B[1]处比完
  - 把败者的索引放在结点B[1]
  - 把胜者的索引放到结点B[0]

败方树示例（1）



败方树示例（2）：L[4]节点15进入后做重构



败方树 ADT

```
template<class T>
class LoserTree{
  private:
      int MaxSize;  //最大选手数
      int n;   //当前选手数
      int LowExt;   //最底层外部节点数
      int offset;      //最底层外部节点之上的节点总数
      int * B;         //赢者树数组，实际存放的是下标
      T * L; //元素数组
      void Play(int p, int lc, int rc, int(* winner)(T A[], int b,
  int c));
```

败方树ADT（续）

```
public:
LoserTree(int Treesize = MAX);
  ~LoserTree(){delete [] B;}
 void Initialize(T A[], int size,int (*winner)(T A[], int b,
  int c), int(*loser)(T A[], int b, int c));  //初始化败方树
  int Winner(); //返回最终胜者索引

 //位置i的选手改变后重构败方树
  void RePlay(int i, int(*winner)(T A[], int b, int c), int
  (*loser)(T A[], int b, int c));};
```


败方树的实现

```
//构造函数
template<class T>
LoserTree<T>::LoserTree(int TreeSize){
   MaxSize = TreeSize;
   B = new int[MaxSize];
   n = 0;
}
//析构函数
template<class T>
LoserTree<T>::~LoserTree(){
delete [] B;
}
```


败方树的实现（续1）

```
//成员函数Winner，返回最终胜者的索引
//在败方树中这个索引存放在B[0]中
template<class T>
int LoserTree<T>::Winner(){
   return (n)?B[0]:0;
}
```


败方树的实现（续2）

```
//成员函数Initilalize负责初始化败方树
template<class T>
void LoserTree<T>::Initialize(T A[], int size,
  int(*winner)(T A[], int b, int c), int(*loser)(T
  A[], int b, int c)) {
 //能否处理size个选手的数组a[]
 if(size > MaxSize || size < 2){
      cout<<"Bad Input!"<<endl<<endl;
      return;
  }
```


败方树的实现（续3）

```
//初始化成员变量
   n = size;
   L = A;

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

   LowExt = 2*(n-s);
   offset = 2*s-1;
```


败方树的实现（续4）

```
//最底层外部结点的比赛
 for(i = 2; i <= LowExt; i+=2)
      Play((offset+i)/2, i-1, i, winner, loser);
 //处理其余外部结点
 if(n%2){//n为奇数，内部结点和外部结点比赛
      //这里用L[LowExt+1]和它的父结点比赛
      //因为此时它的父结点中存放的是
      //其兄弟结点处的比赛胜者索引
      Play(n/2, B[(n-1)/2], LowExt+1, winner, loser);
      i = LowExt+3;
 }
 else i = LowExt+2;
```

败方树的实现（续5）

```
//剩余外部结点的比赛
  for(; i<=n; i+=2)
      Play((i-LowExt+n-1)/2, i-1, i, winner, loser);
}
```


败方树的实现（续6）

```
//成员函数Play负责在内部结点B[p]处开始比赛
template<class T>
void LoserTree<T>::Play(int p, int lc, int rc, int(* winner)(T A[], int b, int c), int(* loser)(T A[], int b, int c)){
  B[p] = loser(L, lc, rc);//败者索引放在B[p]中

    int temp1, temp2;
    temp1 = winner(L, lc, rc);//p处的胜者索引
```


败方树的实现（续7）

```
while(p>1 && p%2){
      //右孩子，需要沿路径继续向上比赛
      //和B[p]的父结点所标识的外部结点相比较
      //p的胜者和p的父结点比较，赢者暂存在temp2中
      temp2 = winner(L, temp1, B[p/2]);
      //败者索引放入B[p/2]
      B[p/2] = loser(L, temp1, B[p/2]);
      temp1 = temp2;
      p/=2;
  }//while
```


败方树的实现（续8）

```
//结束循环(B[p]是左孩子，或者B[1])之后，
//在B[p]的父结点写入胜者索引
  B[p/2] = temp1;

}
```


败方树的实现（续9）

```
//成员函数RePlay负责选手i的值改变后重新开始比赛
template<class T>
void LoserTree<T>::RePlay(int i, int (*winner)(T A[], int b, int c), int (*loser)(T A[], int b, int c)){
  if(i <= 0 || i > n){
      cout<<"Out of Bounds!"<<endl<<endl;
      return;
  }
```


败方树的实现（续10）

```
    int p;
    //确定父结点的位置
    if(i <= LowExt)
        p = (i+offset)/2;
    else
        p = (i-LowExt+n-1)/2;

    //B[0]中始终保存胜者的索引
    B[0] = winner(L, i, B[p]);
    //B[p]中保存败者的索引
    B[p] = loser(L, i, B[p]);
```

败方树的实现（续11）

```
//沿路径向上比赛
  for(; (p/2)>=1; p/=2){
      int temp;//临时存放赢者的索引
      temp = winner(L,B[p/2], B[0]);
      B[p/2] = loser(L,B[p/2], B[0]);
      B[0] = temp;
  }
}
```


利用败方树的多路归并排序算法

```
//输入参数分别是：
// lt－败方树，racer－最初的竞赛者，
// bufferPool－缓冲池，f－输入/输出文件句柄数组
//这里输出文件句柄是f[0]，输入文件句柄是f[1]到
//f[MAX]，MAX为输入文件的数目
//NO_MEANING宏代表一个极大值
template <class T>
void multiMerge(LoserTree<T> & lt, T * racer,
  Buffer<T> * bufferPool, FILE * * f){
  int winner; //最终胜者索引
  T read;     //读出的数据置放在里面
```


利用败方树的多路归并排序算法（续1）

```
//初始化败方树
 lt.Initialize(racer, MAX, Winner, Loser);

//以下处理f[1]到f[MAX]所代表的
 //MAX个输入顺串，
 //并把结果输出到f[0]所代表的输出顺串中

 //取得最终胜者索引
 winner = lt.Winner();
```


利用败方树的多路归并排序算法（续2）

```
while(racer[winner] != NO_MEANING){
//循环退出时胜者为NO_MEANING，
//所有的输入顺串都已经处理完毕
//把胜者插入到输出缓冲区中
//输出缓冲区满，flush到磁盘文件去
      if(bufferPool[0].isFull())
            bufferPool[0].flush(f[0]);
      bufferPool[0].insert(racer[winner]);
```


利用败方树的多路归并排序算法（续3）

```
//从输入缓冲区读入一个新的竞赛者
if(bufferPool[winner].isEmpty()==false)
      //输入缓冲区不为空
      //从缓冲区读入值放进racer[winner]
      bufferPool[winner].read(racer[winner]);
else{//输入缓冲区为空
      if(!feof(f[winner])){
        //如果对应的输入文件还有数据
        //从输入文件读入一批数据到输入缓冲区
        fillBuffer(f[winner], bufferPool[winner]);
```


利用败方树的多路归并排序算法（续4）

```
      //从缓冲区读数据放进racer[winner]
      bufferPool[winner].read(racer[winner]);
} //if
 else{//对应的输入文件没有数据
  //在racer[winner]位置放NO_MEANING
      racer[winner] = NO_MEANING;
  }//else
}//else
```

利用败方树的多路归并排序算法（续5）

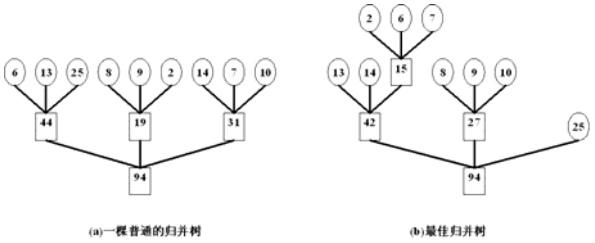
```
//重新进行比赛，取得胜者索引
lt.RePlay(winner, Winner<int>, Loser<int>);
winner = lt.Winner();
}//while

//把输出缓冲区中剩余的数据写进磁盘文件
  bufferPool[0].flush(f[0]);
}
```


多路归并的效率

假设对k个顺串进行归并。

- 原始方法：找到每一个最小值的时间是Θ(k)，产生一个大小为n的顺串的总时间是Θ (k·n)
- 败方树方法：初始化包含k个选手的败方树需要Θ (k)的时间；读入一个新值并重构败方树的时间为Θ (*log* k)。故产生一个大小为n的顺串的总时间为Θ (k+n·*log* k)。

最佳归并树



(a)一棵普通的归并树　　(b)最佳归并树

- （a）访外总次数为(6+13+25+8+9+2+14+7+10)×2×2=376
- （b）外存读/写块的次数为 (2+6+7)×3×2+(13+14)×2×2+(8+9+10)×2×2+25×2=356

最佳归并树

- 如果在进行多路归并的时候，各初始顺串的长度不同，对外存扫描的次数，即执行时间会产生影响。
- 把所有初始顺串的块数作为树的叶结点，如果是K路归并则建立起一棵K-叉Huffman树。这样的一棵Huffman树就是最佳归并树。
- 通过最佳归并树进行多路归并可以使对外存的I/O降到最少，提高归并执行效率。

本章总结

- 主存和外存的主要区别
- 磁盘的物理结构和工作方式
- 外存文件组织
- 缓冲区和缓冲池
- 外排序思想
- 置换选择排序
- 选择树（赢者树和败方树）
- 多路归并的效率