



北京大学

第九章 外排序

邹磊

zoulei@pku.edu.cn

北京大学计算机科学技术研究所

引言

- 内存是一种有限的存储资源
- 现实世界数据是海量的（Big Data），需要外部存储器（外存）支持
- 外存文件的排序本质上还是需要借助内排序完成，需要多次的内外存之间的数据交换
- 与内存排序具有明显不同的特点

本章内容

➤ 9.1 主存储器和外存储器

➤ 9.2 文件的组织与管理

➤ 9.3 外排序

9.1 计算机存储器

- **主存储器（简称“内存”或“主存”）**
 - 随机访问存储器（Random Access Memory, 即RAM）
 - 高速缓存、视频存储器等
- **外存储器（简称“外存”）**
 - 硬盘、软盘、磁带等
- **主存、外存具有不同的组织方式和访问特点**

外存储器的特点

- 优点：永久存储能力、便携性
 - ➡ 外存可以永久存储，而内存却易失
 - ➡ 外存便携
- 缺点
 - ➡ 访问时间长
 - ➡ 访问外存(10^{-3} 秒) 比访问内存(10^{-9} 秒)慢5~6个数量级
- 讨论外存数据结构及其操作时，遵循的原则
 - ➡ 尽量减少访外存次数！

外存数据的访问方式

- 外存访问分为**定位和存取**两个阶段
- 外存被划分为长度固定的存储空间
- 外存的数据访问**以块(Block)为单位进行**，从而减少外存的定位次数，进而减少外存读写的时间耗费

9.2 文件的组织与管理

- 文件(file)是存储在外存上的数据结构，是由大量性质相同的记录组成的集合。
- 所谓记录，就是具有独立逻辑意义的数据块，是文件的基本数据单位。
 - 简单的记录可以是字符或者二进制序列
 - 复杂的记录可由若干字段或域(field)的数据项组成

文件分类——按记录类型分

➤ 操作系统文件

- ➡ 是一组连续的字符序列，没有明显结构。
- ➡ 用户也可以将文件划分成若干逻辑记录，以便存取和使用

➤ 数据库文件

- ➡ 是有结构记录集合，每条记录都由一个或多个数据项组成，而每个数据项是不可再分的基本数据单元

数据库文件示例

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

文件的分类(续)——按记录信息长度分

➤ 定长文件

- ➡ 文件中每条记录均含有相同的信息长度

➤ 不定长文件

- ➡ 文件中记录长度不相等

9.2.1 文件组织

- 操作系统**以文件的方式组织数据**，完成文件的逻辑结构到外存的物理结构的映射
- 文件逻辑组织
 - ➡ 定长记录、变长记录和按关键码存取的记录
- 文件物理结构
 - ➡ 顺序文件、散列文件、索引文件、倒排文件

9.2.2 C++的流文件

- 文件流是以外存文件为输入输出对象的数据流
- 文件流与文件不是同一个概念，文件流不是由若干个文件组成的流。
- 文件流本身不是文件，而只是以文件为输入输出对象的流。

标准输入\输出流类

➤ 包括 `istream`, `ostream` 和 `iostream` 类

➡ `Istream`: 通用输入流和其它输入流的基类

➡ `Ostream`: 通用输出流和其它输出流的基类

➡ `Iostream`: 通用输入输出流和其它输入输出流的基类

用于文件操作的文件类

- **ifstream类**：从**istream**类派生，用来支持从磁盘文件的输入
- **ofstream类**：从**ostream**类派生，用来支持向磁盘文件的输出
- **fstream类**：从**iostream**类派生，用来支持对磁盘文件的输入和输出

fstream类的主要成员函数

#include<fstream.h>

//fstream=ifstream+ofstream

void fstream::open(char* name, openmode mode); //打开文件进行处理

void fstream::close(); //处理结束后关闭文件

fstream::read(char* ptr, int numbytes); //从文件当前位置读入一些字节

fstream::write(char* ptr, int numbytes); //向文件当前位置写入一些字节

fstream::seekg(int pos); //移动指针到当前读取位置

fstream::seekg(int pos, ios::curr);

fstream::seekp(int pos); //移动指针到当前写出位置

fstream::seekp(int pos, ios::end);

9.3 外排序

- 根据内存的大小，将外存中的数据文件划分成若干段，每次把其中一段读入内存并用内排序方法进行排序。
- 这些已排序的段或子文件称为**顺串或归并段**。
- 顺串写回外存等待进一步处理，让出内存空间处理文件的其它未排序的段。

外排序的基本过程

I. 置换选择排序

➡ **目的：** 把外存文件初始化为尽可能长的有序顺串集；

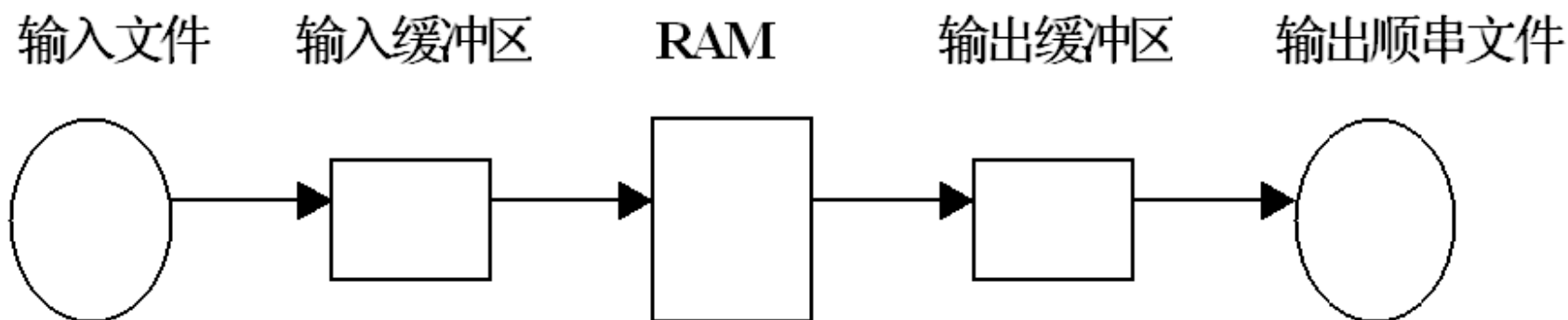
II. 归并排序

➡ **目的：** 把顺串集合逐趟归并排序，形成全局有序的外存文件

外排序的时间组成

- I. 产生初始顺串的内排序所需时间**
- II. 初始化顺串和归并过程所需的读写 (I/O) 时间**
- III. 内部归并所需要的时间**
- **减少外存信息的读写 (I/O) 次数是提高外部排序效率的关键**

9.3.1 置换选择排序



- **目的：** 将文件生成若干初始顺串（顺串越长越好，个数越少越好）
- **实现：** 借助在RAM中的堆来完成

置换选择算法

1. 初始化最小堆：目的是提高RAM中排序的效率

(a) 从缓冲区读M个记录放到数组RAM中

(b) 设置堆尾标志： $LAST = M - 1$

(c) 建立一个最小值堆

置换选择算法(续)

2. 重复以下步骤，直至堆空 (**结束条件**) (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
 - i. 如果R的关键码**不小于**刚输出的关键码值，则把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 (**等待下一顺串处理**)，
设置 **$LAST = LAST - 1$**
 - (c)重新排列堆，筛出根结点

算法分析

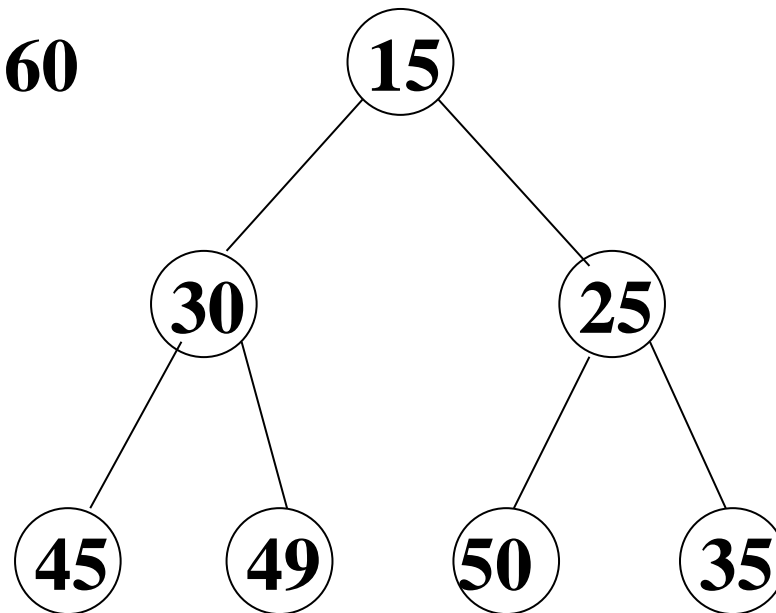
- 算法结束后，RAM中也填满了不能处理的数据，直接建成堆，留待下一顺串来处理
- 大小是M的堆，最小顺串的长度为M的记录
 - ➡ 至少原来堆中的那些记录将成为顺串的一部分
 - ➡ 最好情况下，有可能一次就把整个文件生成为一个顺串（何种情况下？）
 - ➡ 平均长度 $2M$, M 种形态，每种期望输出

算法过程示例

27

16

60



置换选择算法

//A是从外存读入n个元素后所存放的数组

```
template <class Elem>
```

```
void ReplacementSelection(Elem * A, int n, const char * in,  
const char * out){
```

```
    Elem mval;                //存放最小堆的最小值
```

```
    Elem r;                   //存放从输入缓冲区中读入的元素
```

```
    FILE * iptF;              //输入文件句柄
```

```
    FILE * optF;              //输出文件句柄
```

```
    Buffer<Elem> input;         //输入 buffer
```

```
    Buffer<Elem> output;        // 输出buffer
```


置换选择算法(续)

//初始化输入输出文件

initFiles(inputFile, outputFile, in, out);

//初始化堆的数据，读入n个数据

initMinHeapArray(inputFile, n, A);

//建立最小值堆

MinHeap<Elem> H(A, n, n);

//初始化inputbuffer，读入部分数据

initInputBuffer(input, inputFile);

置换选择算法实现(续)

```
for(int last = n-1; last >= 0;){  
    mval = H.heapArray[0]; //堆的最小值  
    sendToOutputBuffer(input,output,iptF,optF, mval);  
    input.read(r); //从输入缓冲区读入一个记录  
    if(!less(r, mval))    H.heapArray[0] = r;  
    else { //否则用last位置记录代替根结点, 把r放到last  
        H.heapArray[0] = H.heapArray[last];  
        H.heapArray[last] = r;  
    }  
}
```

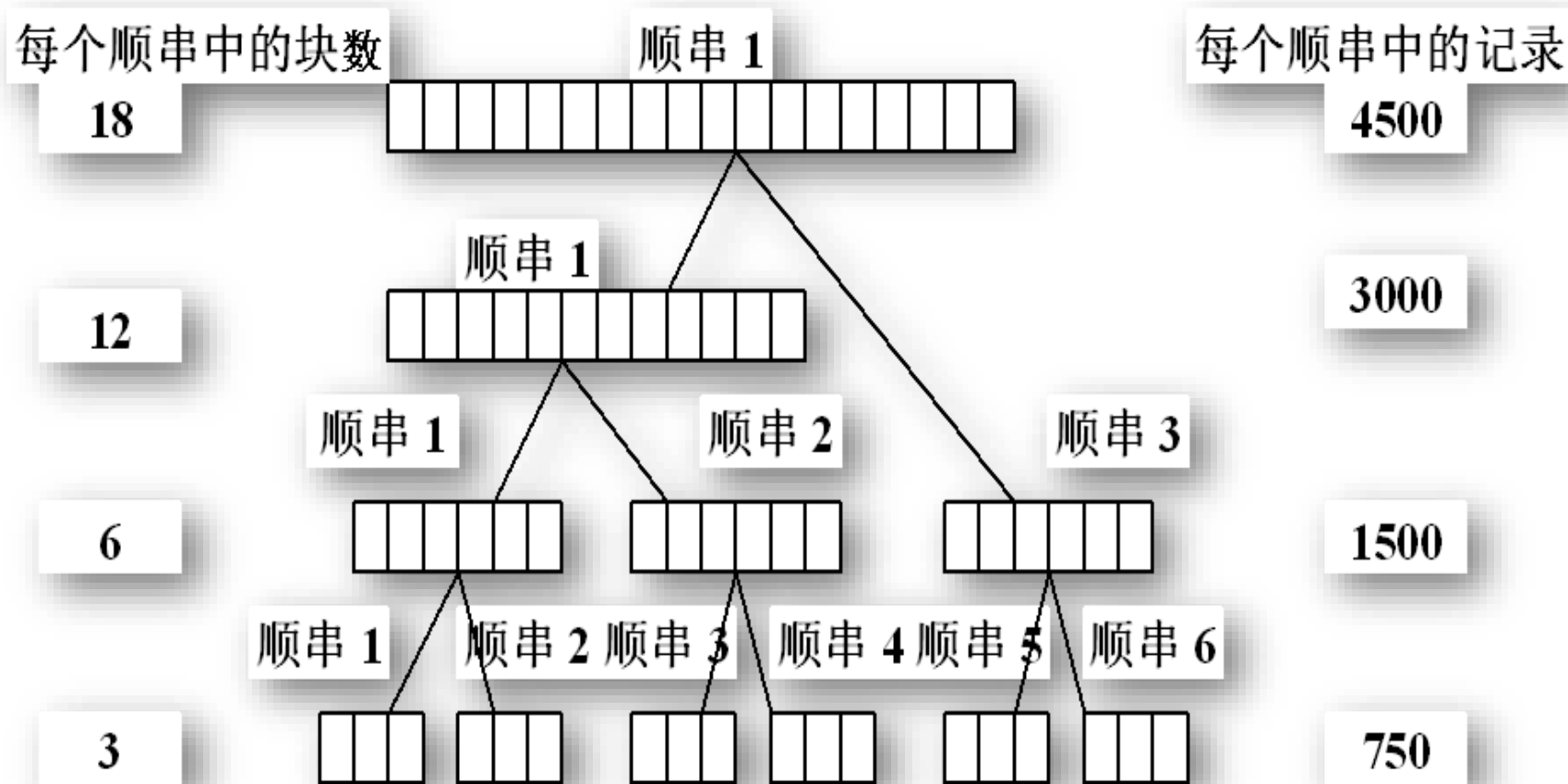
置换选择算法实现(续)

```
H.setSize(last);  
last--;  
}  
if (last!=0)  
    H.SiftDown(0); //堆调整  
}  
endUp(output,inputFile,outputFile); //处理输出缓冲区  
}
```

得到的顺串长度并不相等，平均长度2M。

9.3.2 归并排序

➤ 产生顺串 → 归并排序



归并性质

➤ 二路归并

➡ m 为顺串的个数

➡ 合并树高 $\lceil \log_2 m \rceil + 1$ ，进行 $\lceil \log_2 m \rceil$ 遍扫描

➡ 2个输入缓冲区，1个输出缓冲区

归并排序的趟数

- 所需读写外存次数与归并趟数有关系
- 假设有 m 个初始顺串，每次对 k 个顺串进行归并，归并趟数为 $\lceil \log_k m \rceil$
- 为了减少归并趟数，可以从两个方面着手
 - ➡ 减少初始顺串的个数 m
 - ➡ 增加同时归并的顺串数量 k

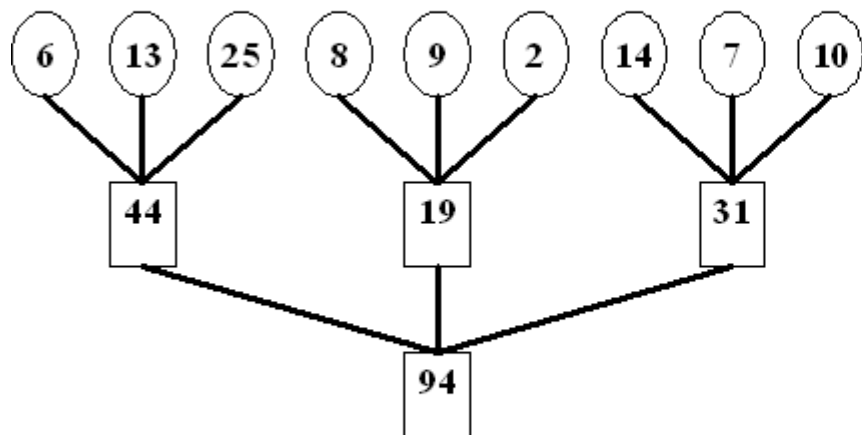
算法分析

- 为一个待排文件创建尽可能大的初始顺串，可以大大减少扫描遍数和外存读写次数。
- 归并顺序的安排也能影响读写次数，把初始顺串长度作为权，其实质就是Huffman树最优化问题

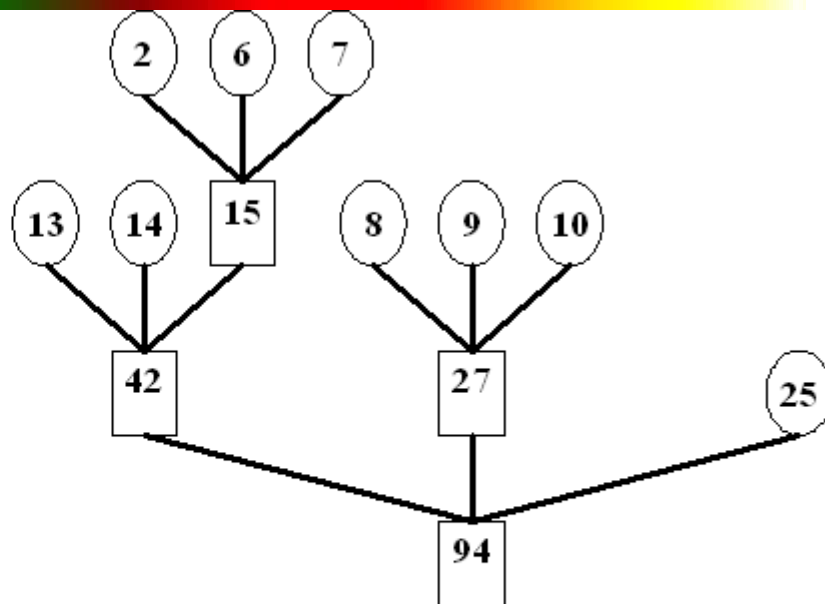
最佳归并树

- 进行多路归并时，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

最佳归并树



(a) 一棵普通的归并树



(b) 最佳归并树

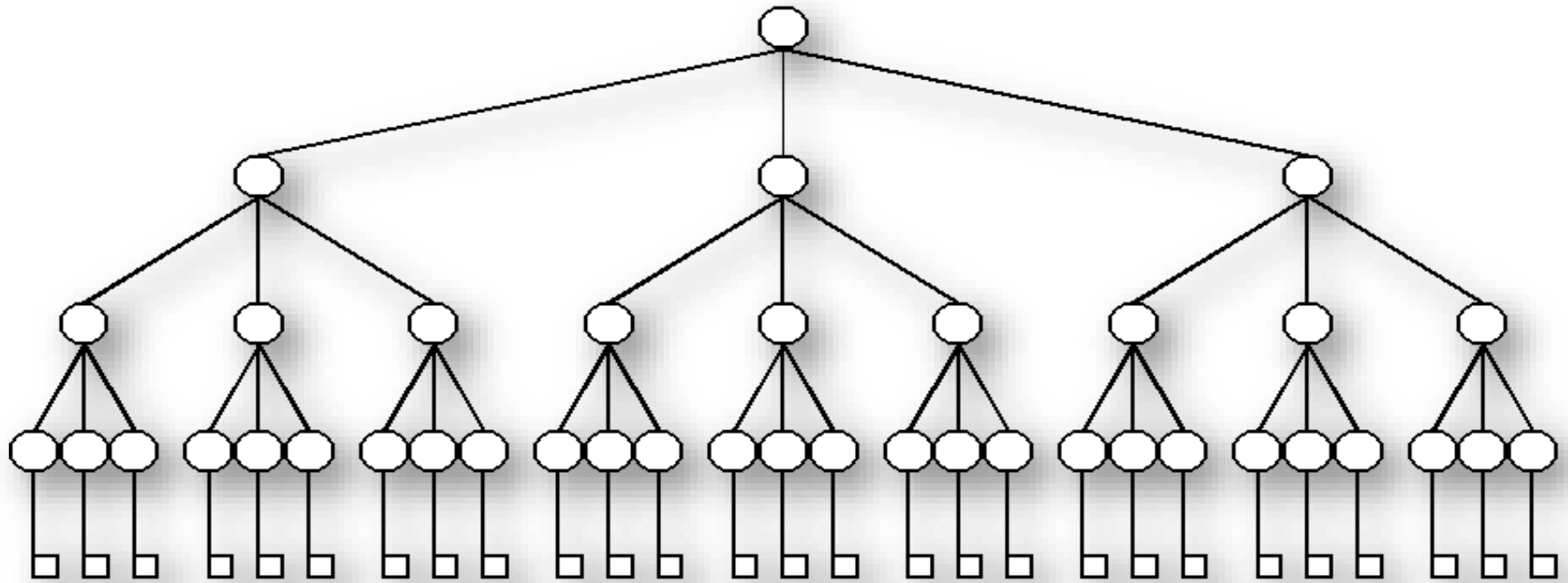
➤ (a) 访外总次数为 $(6+13+25+8+9+2+14+7+10) \times 2 \times 2 = 376$

➤ (b) 访外总次数为

$$(2+6+7) \times 3 \times 2 + (13+14) \times 2 \times 2 + (8+9+10) \times 2 \times 2 + 25 \times 2 = 356$$

9.3.3 多路归并树

➤ k 路归并指每次将 k 个顺串合并成一个顺串



多(3)路归并外排序

多路归并树分析

- 在做 k 路归并的时候，每次都需要做 $k-1$ 次比较才能找出所要的记录，代价较大
- 选择树
 - ➡ 赢者树
 - ➡ 败者树
- 目的
 - ➡ 提高在 k 个归并串的当前值中找到最小值的效率

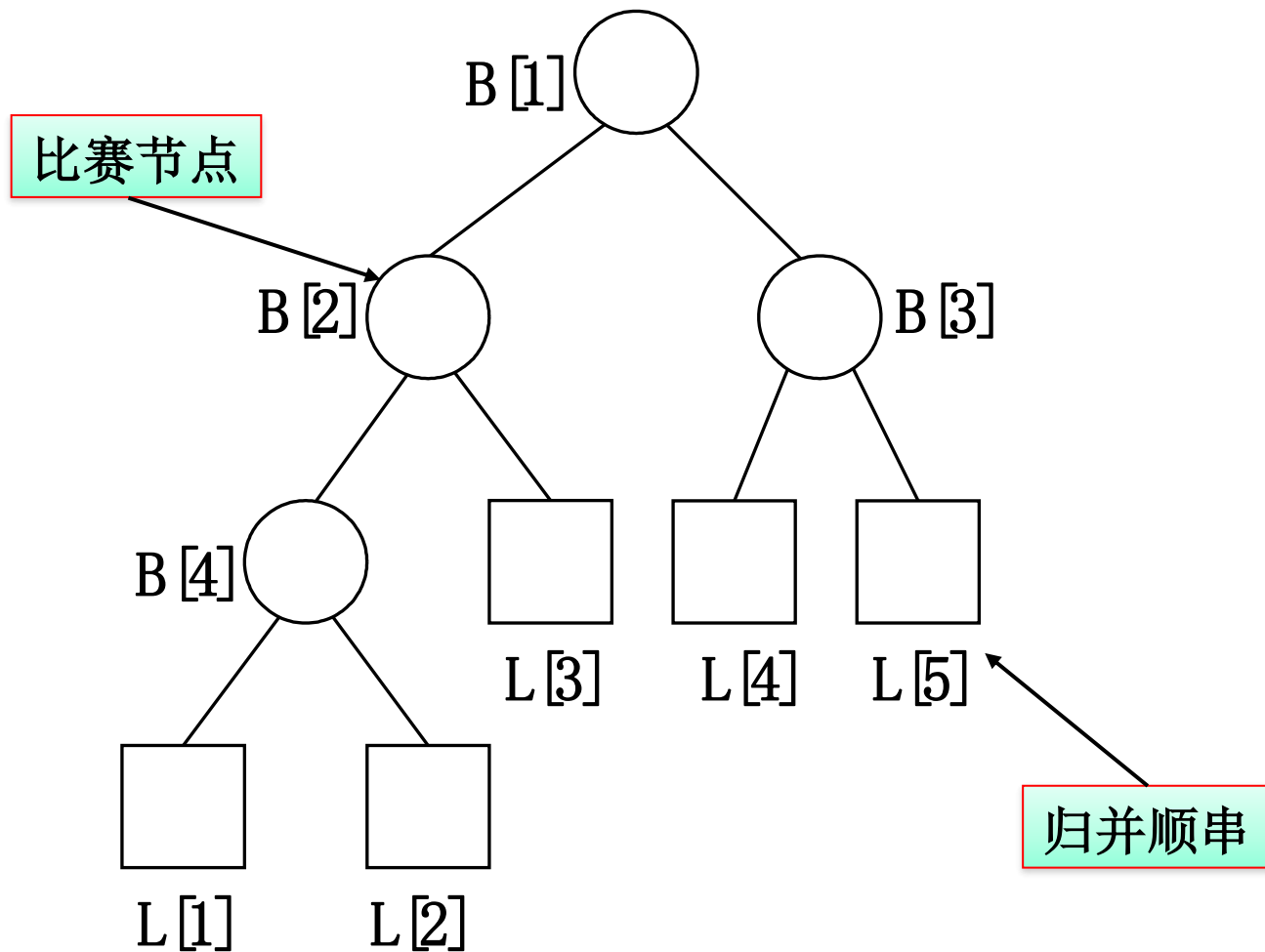
A. 赢者树

➤ 用完全二叉树作为存储结构

- ➡ 叶结点用数组 $L[1..n]$ 表示，内部结点用数组 $B[1..n-1]$ 表示
- ➡ 数组 B 中实际存放的是数组 L 的索引

赢者树的结构

n路归并，赢者树具有 $2n-1$ 个节点



外部结点L[i]与内部父结点B[p]关系

➤ 外部结点数为 n ，内部节点数为 $n-1$ ，比赛树深度 $s = \lceil \log_2 n \rceil - 1$

➤ 如右图，比赛树（只考虑内部节点）深度为 $s = \lceil \log_2 5 \rceil - 1 = 2$

➤ 最底层最左端内部结点编号为 2^s

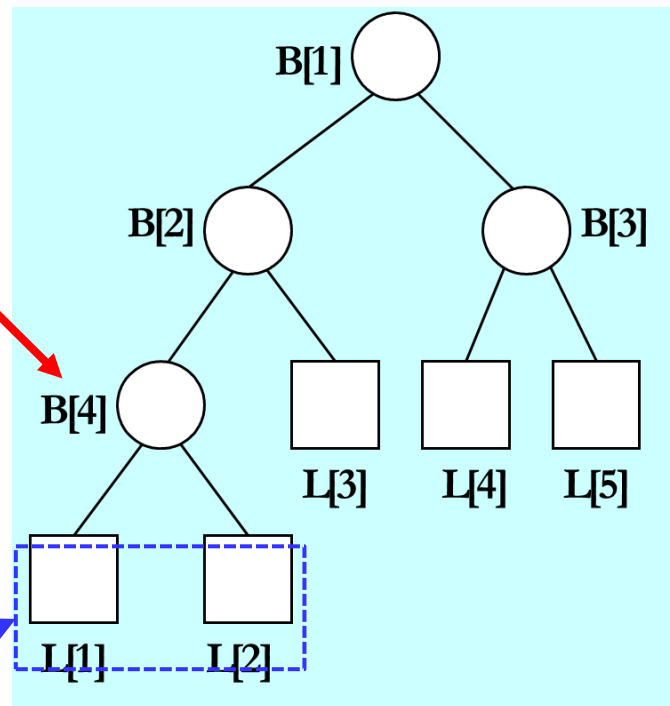
➤ 如右图， $B[4] = 2^2 = 4$

➤ 最底层的内部结点数为 $n - 2^s$

➤ 如右图， $n - 2^s = 5 - 4 = 1$

➤ 最底层的外部结点数（LowExt）为最底层内部结点数的2倍

➤ 即：LowExt = $2 * (n - 2^s) = 2 * 1 = 2$



外部结点L[i]与内部父结点B[p]关系

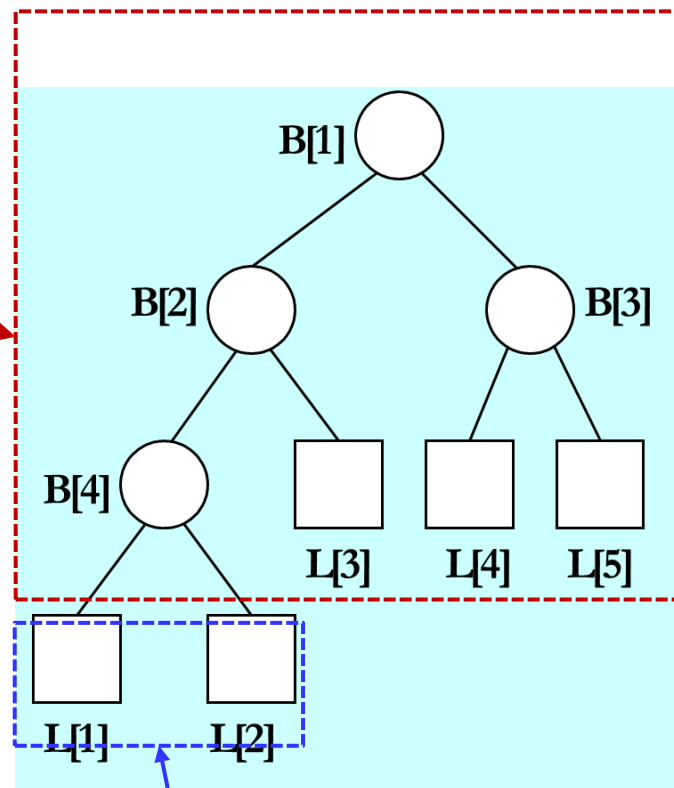
➤ 最底层外部结点之上的所有(内+外)

结点数目 (offset) , $offset =$

$$2^{(s+1)} - 1$$

➡ 如右图, $offset = 2^{(2+1)} - 1 = 7$

➤ 外部结点L[i]与内部父结点B[p]关系可以表示如下:



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

赢者树的特点

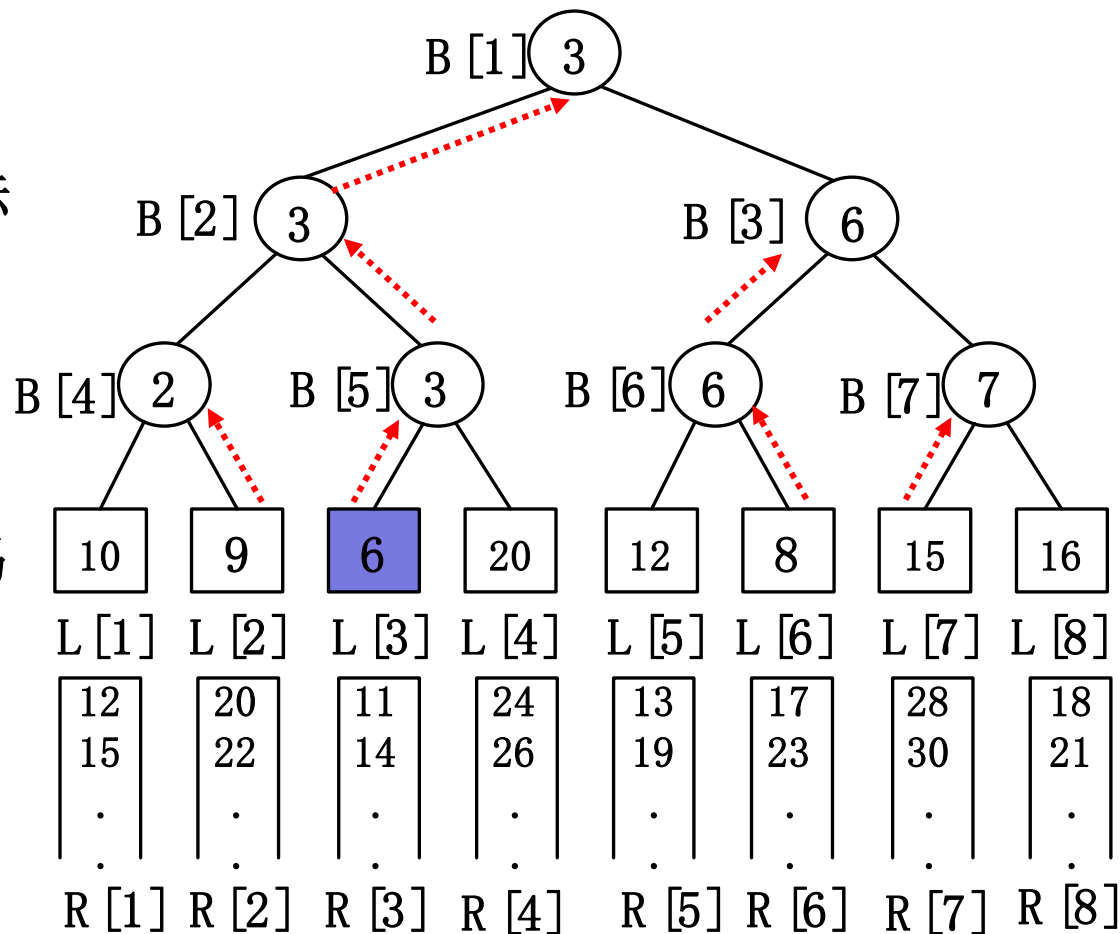
- 通过比较两个选手的分数确定一场比赛的赢家
 - ➡ 从树最底层开始，每两个树叶之间进行比赛，输的选手被淘汰，赢的继续向上进行竞赛，树根记录了整个比赛的胜者。
- 如果选手 $L[i]$ 的分值改变，可以修改这棵赢者树
 - ➡ 沿着从 $L[i]$ 到根结点的路径，和兄弟节点的值进行比较，根据比赛结果修改二叉树节点的值（赢着上），而不必改变树其他部分的比赛结果

赢者树的示例

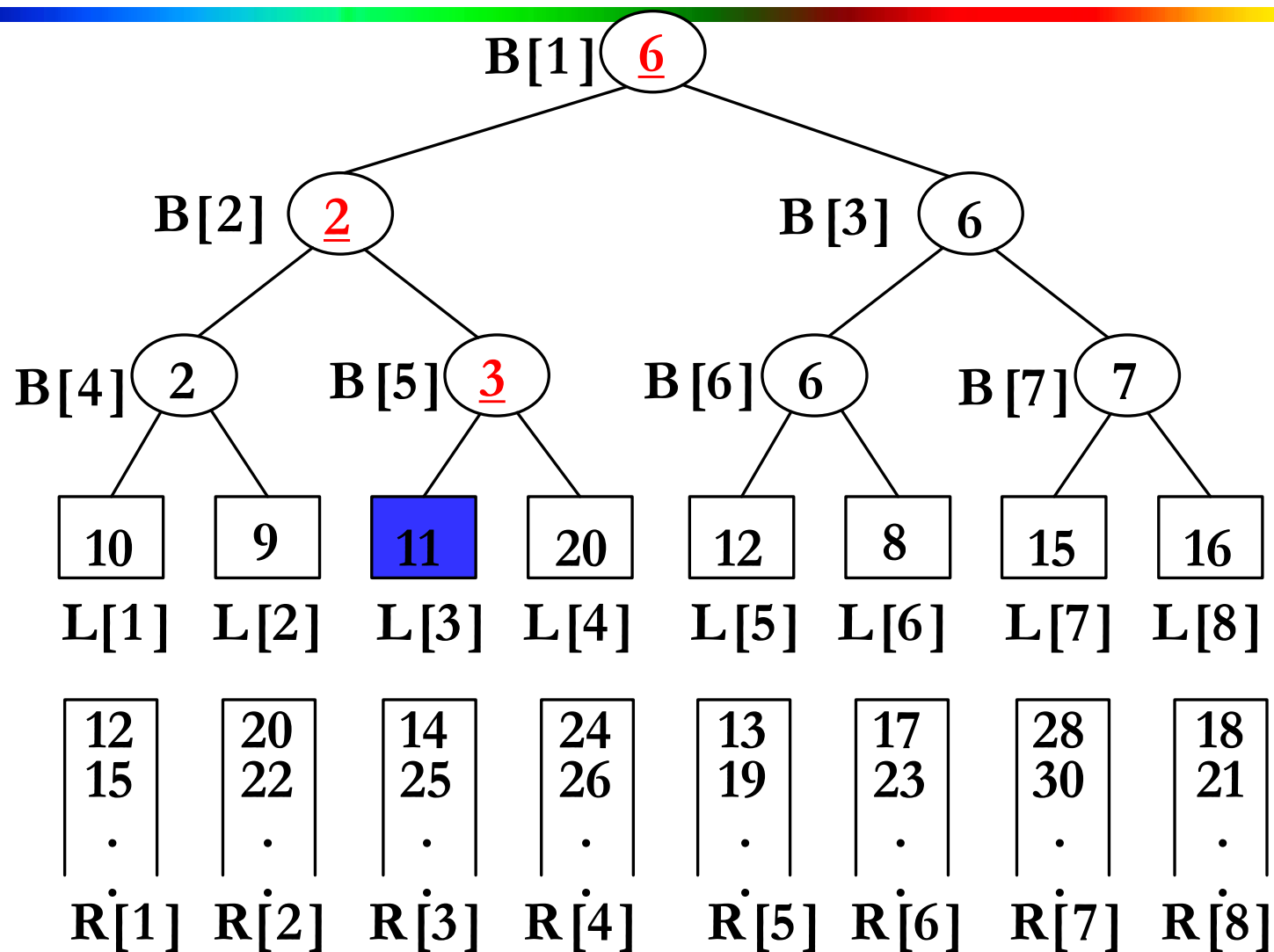
数组B中存储的是L的下标

当前进行比较的8个关键码

等待进行比较的8个顺串



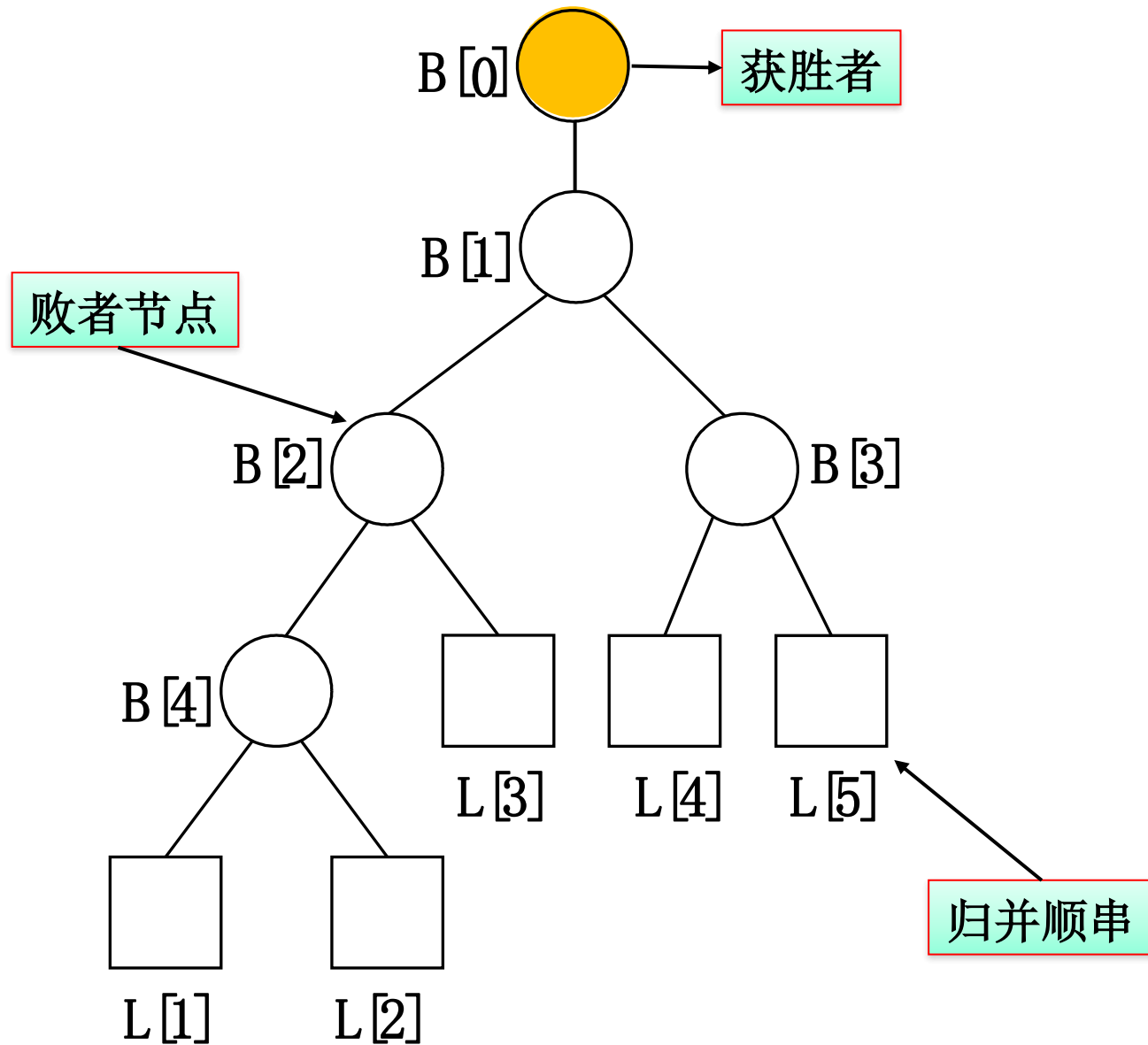
赢者树的重构



(2) 败者树

- 是赢者树的一种变体。
- 在败者树中，用父结点记录其左右子结点进行比赛的败者，而让获胜者去参加更高阶段的比赛。
- 新增根结点 $B[0]$ ，来记录整个比赛的最终胜者。
- 败者树是为了简化重构过程，树结构未变。

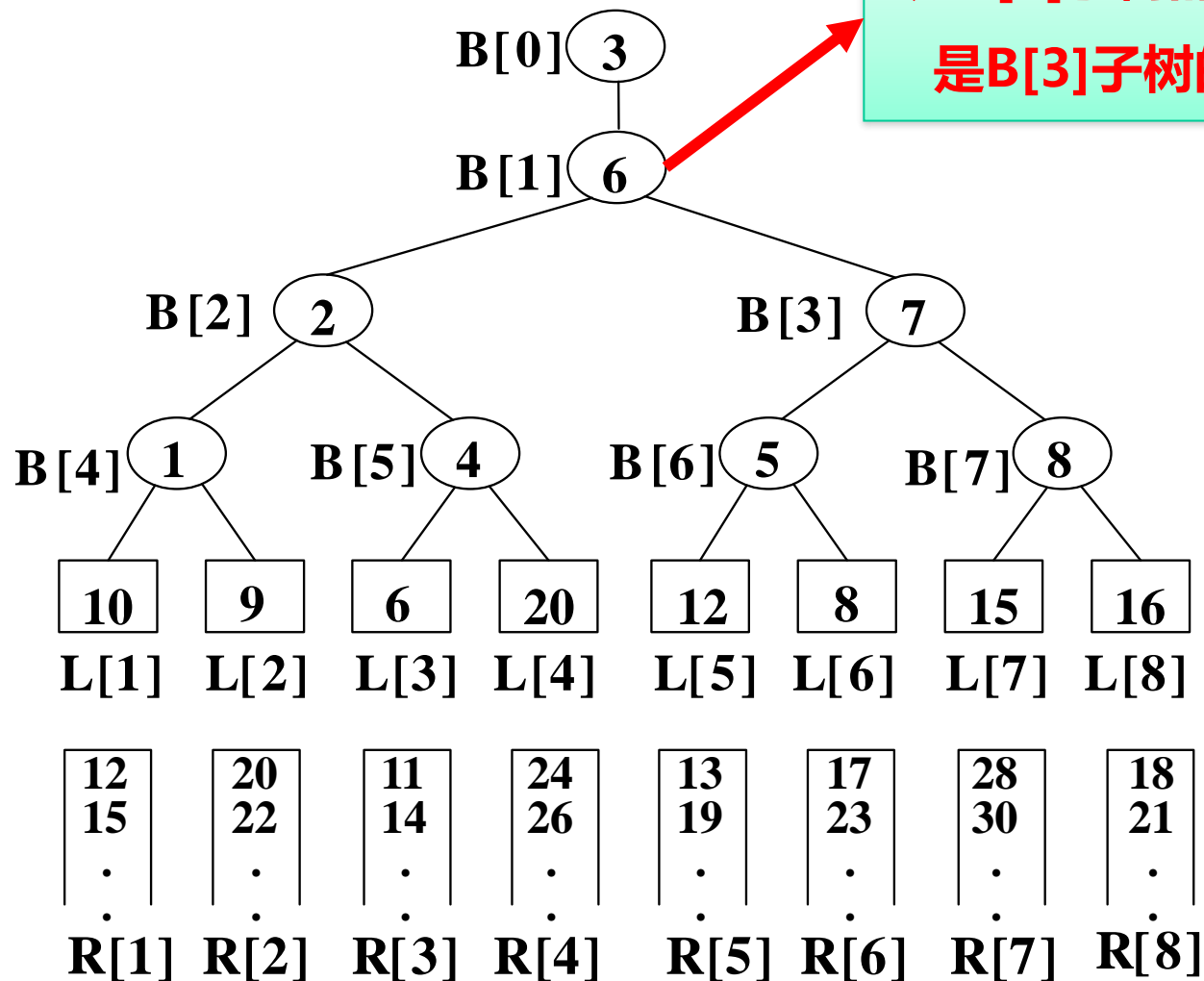
败者树的结构



比赛过程

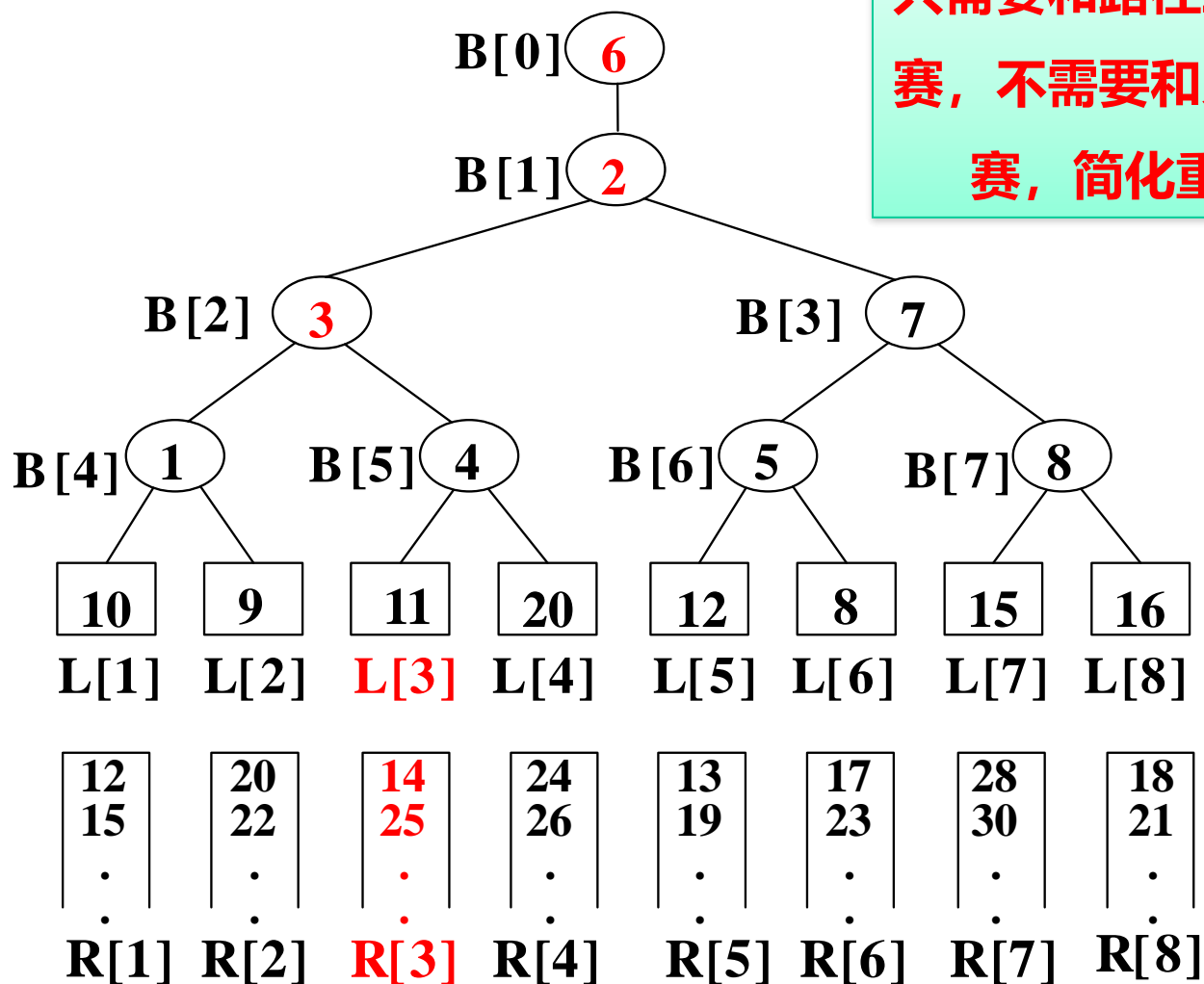
- 将新进入树的结点与其父结点进行比赛
 - ➡ 把败者存放在父结点中
 - ➡ 而把胜者再与上一级的父结点进行比赛
- 这样的比赛不断进行，直到结点B[1]
 - ➡ 把败者的索引放在结点B[1]
 - ➡ 把胜者的索引放到结点B[0]

败者树的示例



败者树的重构

只需要和路径上的节点比赛，不需要和兄弟节点比赛，简化重构过程



败者树算法

```
template<class T>
class LoserTree{
    private:
        int MaxSize;    // 最大选手数
        int n;          // 当前选手数
        int LowExt;     // 最底层外部结点数
        int offset;     // 最底层外部结点之上的结点总数
        int *B;         // 败者树数组，实际存放的是下标
        T *L;           // 元素数组
        // 在内部结点从右分支向上比赛
        void Play(int p, int lc, int rc, int(*winner)(T A[], int
b, int c), int(*loser)(T A[], int b, int c));
```




public:

LoserTree(int Treesize = MAX);

~LoserTree(){delete [] B;}

// 初始化败者树

void Initialize(T A[], int size, int (*winner)(T A[], int b, int c), int (*loser)(T A[], int b, int c));

int Winner(); // 返回最终胜者索引

// 重构败者树

void RePlay(int i, int (*winner)(T A[], int b, int c), int (*loser)(T A[], int b, int c));

};

(1) 初始化败者树

```
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)) {
```

```
    int i,s;
```

```
    n = size;
```

// 初始化成员变量

```
    L = A;
```

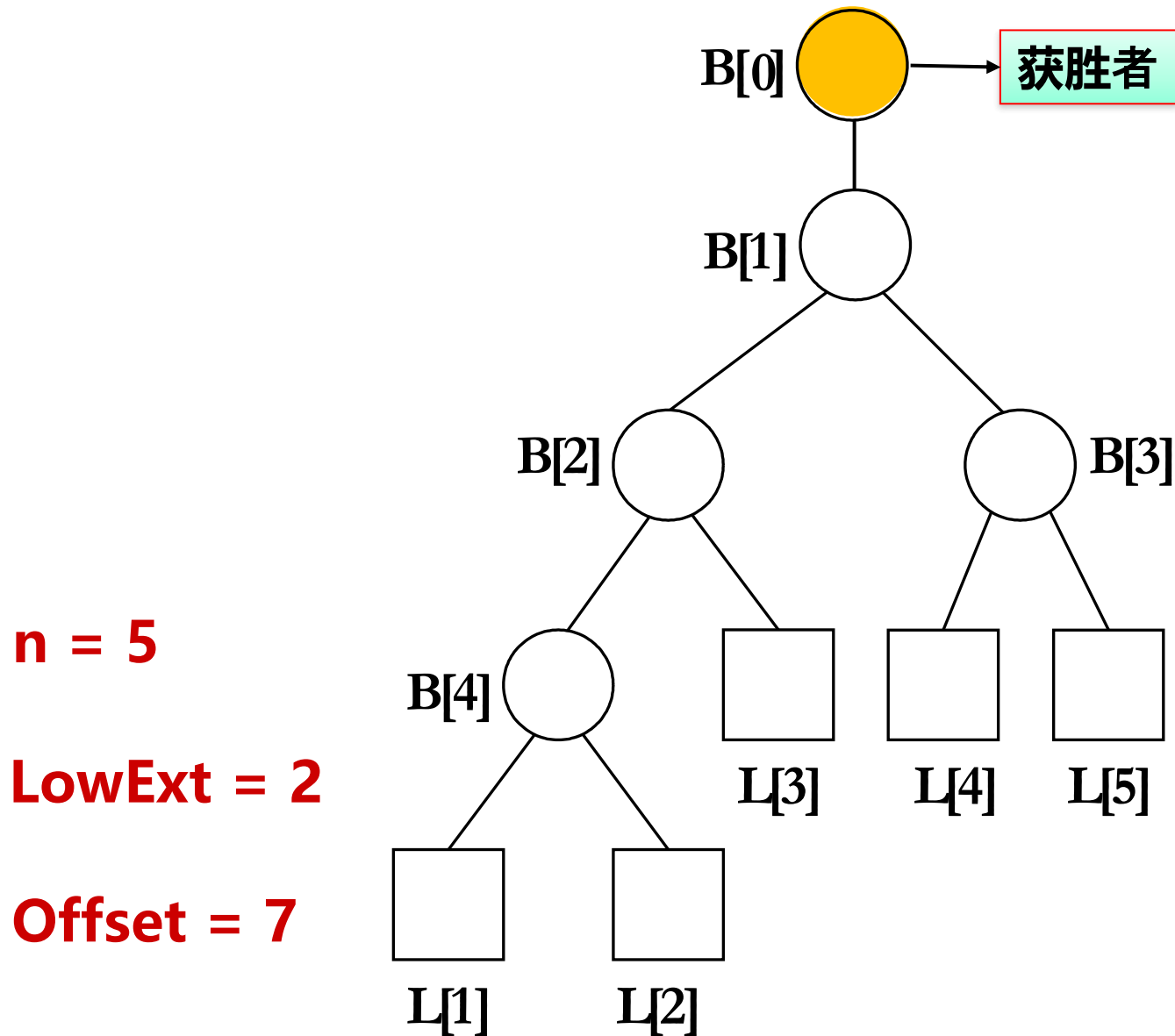
```
    for (s = 1; 2*s <= n-1; s += s);
```

// 计算倒数第2层节点数

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

```
    offset = 2*s-1;
```

败者树的结构



for (i = 2; i <= LowExt; i += 2) // 最底层外部结点比赛

Play((offset+i)/2, i-1, i, winner, loser);

// 处理其余外部结点

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

if (n%2) { //n为奇数, 内部结点和外部结点比一次

// 暂存在父中的左胜者与外部右子结点比

Play(n/2, B[n], LowExt+1, winner, loser);

i = LowExt+3;

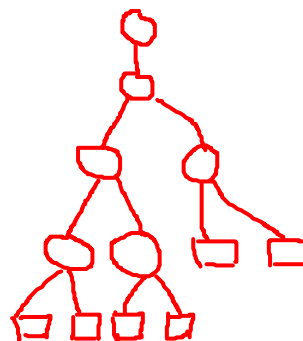
}

else i = LowExt+2;

for (; i <= n; i += 2) // 剩余外部结点的比赛

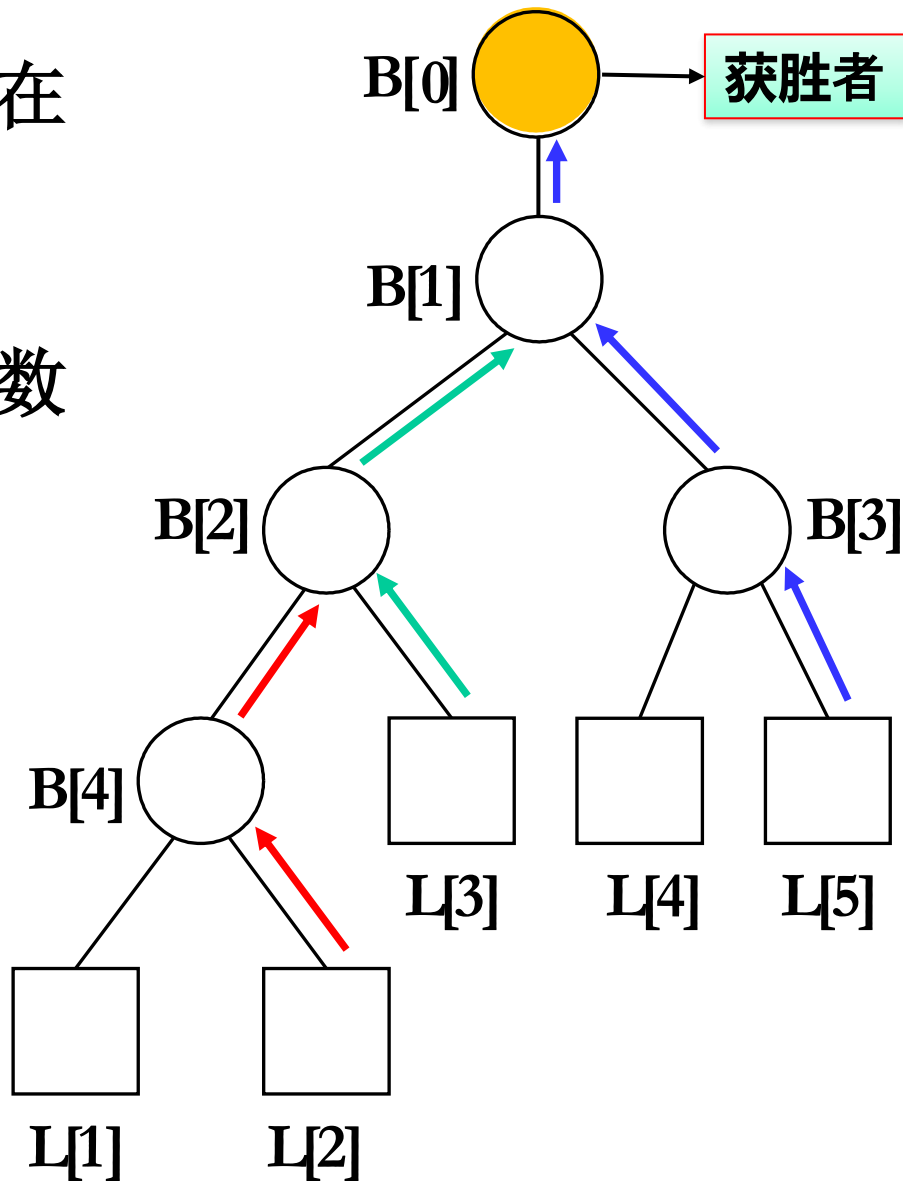
Play((i-LowExt+n-1)/2, i-1, i, winner, loser);

}



(2) 生成败者树

- 沿叶节点（由小到大）所在子树的右分支向上比赛
- 右分支路径节点编号为奇数



生成败者树算法

```
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);
    while (p>1 && p%2) { //p为奇数向上比赛
        temp2 = winner(L, temp1, B[p/2]); //胜者与父比较
        B[p/2] = loser(L, temp1, B[p/2]); //败者留下
        temp1 = temp2; //胜者放入temp1
        p/=2; //p向上指向父结点
    } // while
    B[p/2] = temp1; //B[p]是左孩子或者p=1
}
```

(3) 重构败者树

```
void LoserTree<T>::RePlay(int i, int (*winner)(T A[], int b, int c),
    int (*loser)(T A[], int b, int c)) {

    int p;                //用于计算父结点索引的临时变量


    if (i <= LowExt)       //确定父结点的位置

        p = (i+offset)/2;

    else p = (i-LowExt+n-1)/2;

    B[0] = winner(L, i, B[p]); // B[0]中保存胜者的索引

    B[p] = loser(L, i, B[p]);  // B[p]中保存败者的索引
```



```
for (; (p/2) >= 1; p/=2) { // 沿路径向上比赛
```

```
    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;
```

```
}
```

```
}
```


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 $\Theta(k)$ ，产生一个大小为 n 的顺串的总时间是 $\Theta(k \cdot n)$
- 败方树方法
 - ➡ 初始化包含 k 个选手的败方树需要 $\Theta(k)$ 的时间
 - ➡ 读入一个新值并重构败方树的时间为 $\Theta(\log k)$
 - ➡ 故产生一大小为 n 的顺串的总时间为 $\Theta(k + n \cdot \log k)$

思考题

- 用堆作为选择树进行归并排序可以吗？
- 赢者树和败者树和堆相比有什么优缺点？
请对比分析。



再见…

联系信息：

电子邮件：`zoulei@pku.edu.cn`

电 话：82529643