堆的数据结构

2023年7月28日 10:05

优先队列：

- 优先队列（Priority Queue）：特殊的“队列”，取出元素的顺序是依照元素的优先权（关键字）大小，而不是元素进入队列的先后顺序。

屏幕剪辑的捕获时间: 2023/7/28 10:18

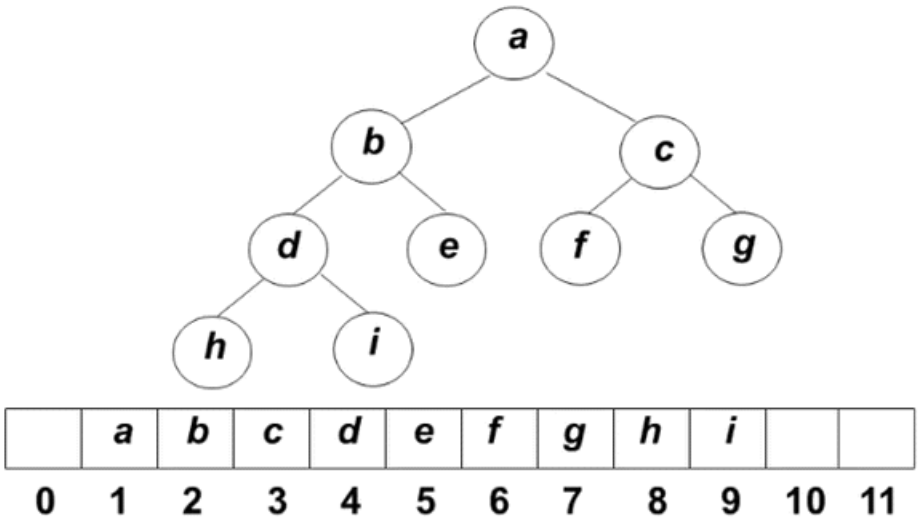
若采用数组或链表实现优先队列

- 数组：
  - 插入 — 元素总是插入尾部 ~ Θ(1)
  - 删除 — 查找最大（或最小）关键字 ~ Θ(*n*)
    从数组中删去需要移动元素 ~ O(*n*)
- 链表：
  - 插入 — 元素总是插入链表的头部 ~ Θ(1)
  - 删除 — 查找最大（或最小）关键字 ~ Θ(*n*)
    删去结点 ~ Θ(1)
- 有序数组：
  - 插入 — 找到合适的位置 ~ O(*n*) 或 $O(\log_2 n)$
    移动元素并插入 ~ O(*n*)
  - 删除 — 删去最后一个元素 ~ Θ(1)
- 有序链表：
  - 插入 — 找到合适的位置 ~ O(*n*)
    插入元素 ~ Θ(1)
  - 删除 — 删除首元素或最后元素 ~ Θ(1)

屏幕剪辑的捕获时间: 2023/7/28 10:19

如果用树来存储：

优先队列的完全二叉树表示



- 堆的两个特性
  - 结构性：用数组表示的完全二叉树；
  - 有序性：任一结点的关键字是其子树所有结点的最大值(或最小值)
    - “最大堆(MaxHeap)”,也称“大顶堆”：最大值
    - “最小堆(MinHeap)”,也称“小顶堆”：最小值

屏幕剪辑的捕获时间: 2023/7/28 10:19

堆的抽象数据类型描述

类型名称：最大堆（MaxHeap）

数据对象集：完全二叉树，每个结点的元素值不小于其子结点的元素值

操作集：最大堆H ∈ MaxHeap，元素item ∈ ElementType，主要操作有：

- MaxHeap Create(int MaxSize)：创建一个空的最大堆。
- Boolean IsFull(MaxHeap H)：判断最大堆H是否已满。
- Insert(MaxHeap H, ElementType item)：将元素item插入最大堆H。
- Boolean IsEmpty(MaxHeap H)：判断最大堆H是否为空。
- ElementType DeleteMax(MaxHeap H)：返回H中最大元素(高优先级)。

屏幕剪辑的捕获时间: 2023/7/28 10:20

堆的创建、插入与删除

2023年7月28日　10:20

1.创建堆时，从数组1位置开始存储，而0位置设置为**哨兵**（作用后面再讲）

最大堆的操作

☞ 最大堆的创建

```
typedef struct HeapStruct *MaxHeap;
struct HeapStruct {
        ElementType *Elements; /* 存储堆元素的数组 */
        int Size;              /* 堆的当前元素个数 */
        int Capacity;          /* 堆的最大容量     */
};
```

```
MaxHeap Create( int MaxSize )
{       /* 创建容量为MaxSize的空的最大堆 */
  MaxHeap H = malloc( sizeof( struct HeapStruct ) );
  H->Elements = malloc( (MaxSize+1) * sizeof(ElementType));
  H->Size = 0;
  H->Capacity = MaxSize;
  H->Elements[0] = MaxData;
     /* 定义“哨兵”为大于堆中所有可能元素的值，便于以后更快操作 */
  return H;
}
```

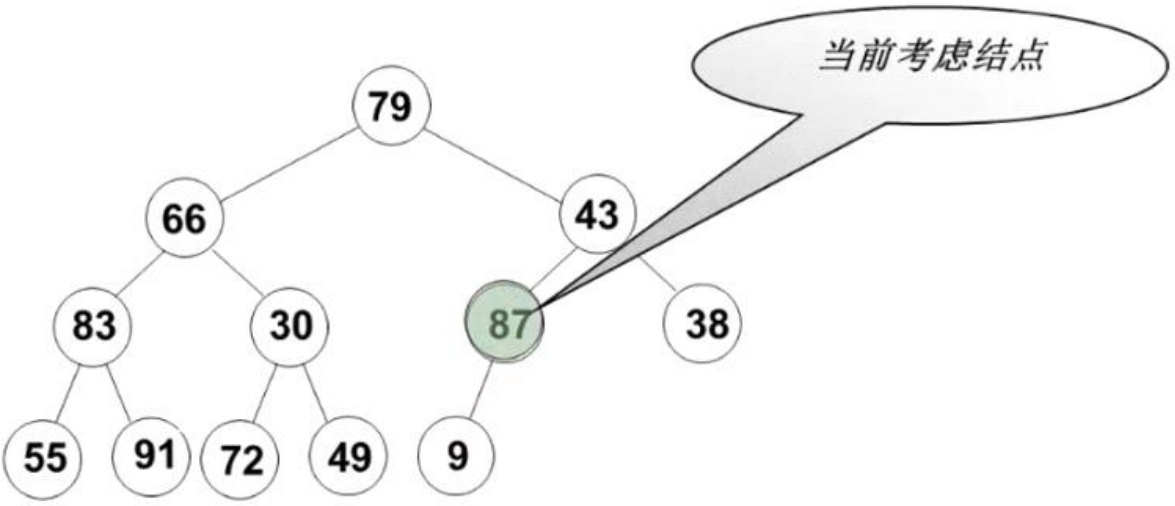
屏幕剪辑的捕获时间: 2023/7/28 12:20

在创建最大、最小堆时，如果按照自顶向下建堆，时间复杂度就是O(NlogN)，**但是还有一种O(N)方法**：

把所有元素顺序放入树中，此时还不是堆，然后从后往前找出第一个有儿子的结点，将其调整为堆（利用了左儿子是一个堆，右儿子是一个堆，然后向两个堆中插入元素使其形成堆的思路），之后连续地调整这个结点之前的各个结点，使其均成为堆，最终得到了自下向上的建堆过程，可以证明其时间复杂度为**O(N)**

※已经自行证明了，考虑堆为h高的满二叉树情况（h=0,1,2...），利用N=2^(h+1)-1即可。

最终结果是T=N-(log(N+1)+1)，甚至比N还要再小一些，调整过程比插入数据还要省时间XD



屏幕剪辑的捕获时间: 2023/8/11 15:25

2.堆的插入操作:

❖ 算法：将新增结点插入到从其父结点到根结点的有序序列中

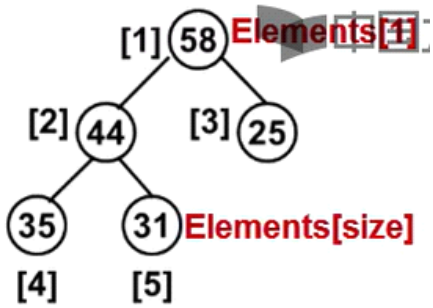
```
void Insert( MaxHeap H, ElementType item )
{ /* 将元素item 插入最大堆H，其中H->Elements[0]已经定义为哨兵 */
    int i;
    if ( IsFull(H) ) {
        printf("最大堆已满");
        return;
    }
    i = ++H->Size; /* i指向插入后堆中的最后一个元素的位置 */
    for ( ; H->Elements[i/2] < item; i/=2 )
        H->Elements[i] = H->Elements[i/2]; /* 向下过滤结点 */
    H->Elements[i] = item; /* 将item 插入 */
}
```

屏幕剪辑的捕获时间: 2023/7/28 12:21

观察上述循环，注意到如果这个元素要放入根部时，还要再与Emelent[0]位置进行判断，也即上面创建堆时提到的哨兵，所以哨兵的作用就是控制循环退出，而无需再每次判断i的值来控制循环，提高了程序效率

3.堆的删除操作:

```
ElementType DeleteMax( MaxHeap H )
{   /* 从最大堆H中取出键值为最大的元素，并删
    int Parent, Child;
    ElementType MaxItem, temp;
    if ( IsEmpty(H) ) {
        printf("最大堆已为空");
        return;
    }
    MaxItem = H->Elements[1]; /* 取出根结点最大值 */
    /* 用最大堆中最后一个元素从根结点开始向上过滤下层结点 */
    temp = H->Elements[H->Size--];
    for( Parent=1; Parent*2<=H->Size; Parent=Child ) {
        Child = Parent * 2;
        if( (Child!= H->Size) &&
            (H->Elements[Child] < H->Elements[Child+1]) )
           Child++;  /* Child指向左右子结点的较大者 */
        if( temp >= H->Elements[Child] ) break;
        else  /* 移动temp元素到下一层 */
           H->Elements[Parent] = H->Elements[Child];
    }
    H->Elements[Parent] = temp;
    return MaxItem;
```



屏幕剪辑的捕获时间: 2023/7/28 12:24

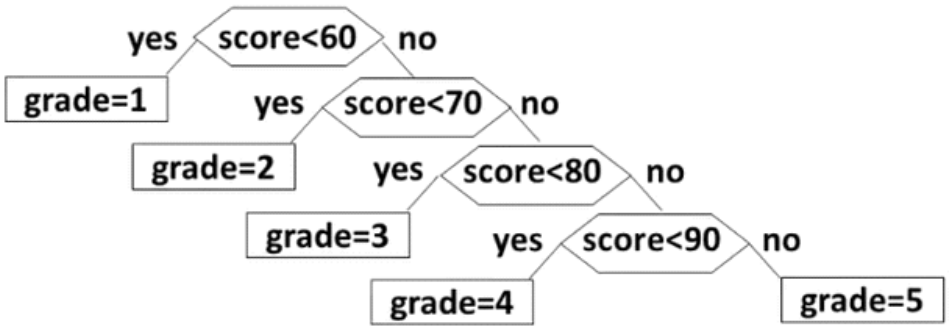
哈夫曼树的定义

2023年7月28日 12:25

- 如果考虑学生成绩的分布的概率：

分数段	0-59	60-69	70-79	80-89	90-100
比例	0.05	0.15	0.40	0.30	0.10

- 查找效率：0.05×1+0.15×2+0.4×3+0.3×4+0.1×4 = 3.15

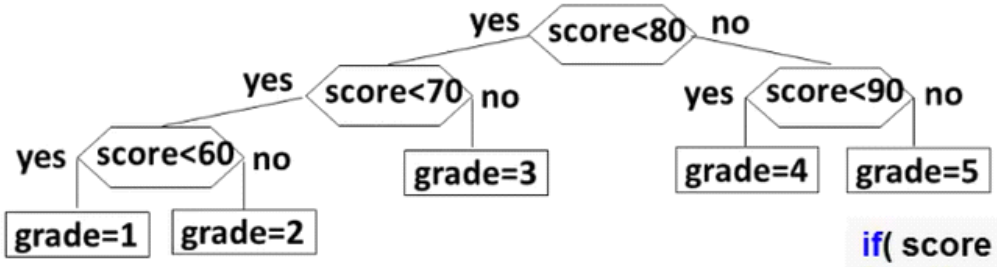


屏幕剪辑的捕获时间: 2023/7/28 12:26

- 如果考虑学生成绩的分布的概率：

分数段	0-59	60-69	70-79	80-89	90-100
比例	0.05	0.15	0.40	0.30	0.10

- 修改判定树：




```
if( score < 80 )
{     if( score < 70 )
          if( score < 60 ) grade =1;
          else grade = 2;
}else if( score < 90 ) grade =4;
else grade =5;
```

效率：0.05×3+0.15×3+0.4×2+0.3×2+0.1×2 = 2.2

如何根据结点不同的查找频率构造更有效的搜索树？

屏幕剪辑的捕获时间: 2023/7/28 12:26

❖ 哈夫曼树的定义

带权路径长度(WPL)：设二叉树有n个叶子结点，每个叶子结点带有权值 w_k，从根结点到每个叶子结点的长度为 l_k，则每个叶子结点的带权路径长度之和就是：$WPL=\sum_{k=1}^{n} w_k l_k$

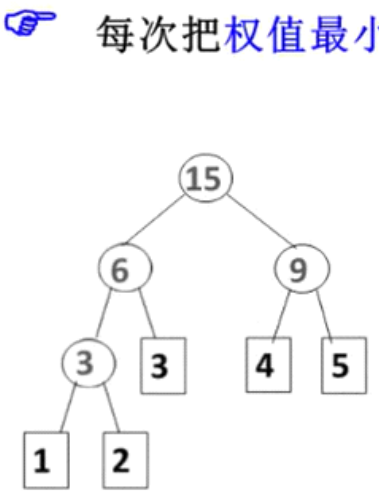
最优二叉树或哈夫曼树：WPL最小的二叉树

屏幕剪辑的捕获时间: 2023/7/28 12:28

哈夫曼树的构造

2023年7月28日　　12:28

哈夫曼树构造比较简单，每次将两个权重最小的二叉树合并即可



屏幕剪辑的捕获时间: 2023/7/28 12:33

C++实现:

```
typedef struct TreeNode *HuffmanTree;
struct TreeNode{
  int Weight;
  HuffmanTree Left, Right;
}
HuffmanTree Huffman( MinHeap H )
{    /* 假设H->Size个权值已经存在H->Elements[]->Weight里*/
     int i;  HuffmanTree T;
     BuildMinHeap(H); /*将H->Elements[]按权值调整为最小堆*/
     for (i = 1; i < H->Size; i++) { /*做H->Size-1次合并*/
          T = malloc( sizeof( struct TreeNode) ); /*建立新结点*/
          T->Left = DeleteMin(H);
                         /*从最小堆中删除一个结点，作为新T的左子结点*/
          T->Right = DeleteMin(H);
                        /*从最小堆中删除一个结点，作为新T的右子结点*/
          T->Weight = T->Left->Weight+T->Right->Weight;
                        /*计算新权值*/
          Insert( H, T ); /*将新T插入最小堆*/
     }
     T = DeleteMin(H);
     return T;
}
```

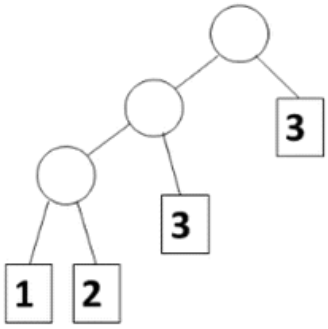
屏幕剪辑的捕获时间: 2023/7/28 12:33

Huffman树的特点：

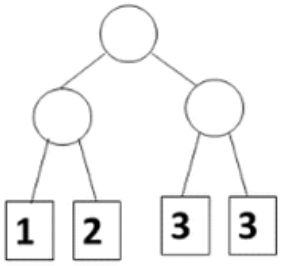
❖ 哈夫曼树的特点：

- 没有度为1的结点；
- n个叶子结点的哈夫曼树共有2n-1个结点；
- 哈夫曼树的任意非叶节点的左右子树交换后仍是哈夫曼树；
- 对同一组权值$\{w_1, w_2, \ldots\ldots, w_n\}$，是否存在不同构的两棵哈夫曼树呢？

对一组权值{ 1, 2 , 3, 3 }，不同构的两棵哈夫曼树：



WPL = 18



WPL = 18

屏幕剪辑的捕获时间: 2023/7/28 13:16

哈夫曼编码

2023年7月28日　13:14

不等长编码：

怎么进行不等长编码？

如何避免二义性？

前缀码*prefix code*：任何字符的编码都不是另一字符编码的前缀

- 可以无二义地解码

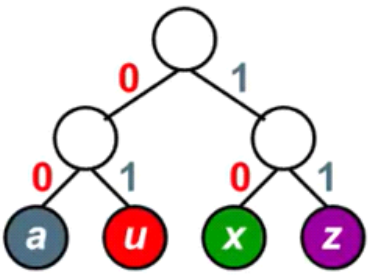
a：1
e：0
s：10

屏幕剪辑的捕获时间: 2023/7/29 7:15

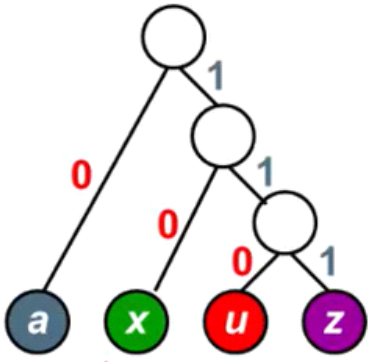
❖二叉树用于编码

用二叉树进行编码：
（1）左右分支：0、1
（2）字符只在叶结点上

四个字符的频率: a:4, u:1, x:2, z:1



Cost (aaaxuaxz → 0000001001001011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16



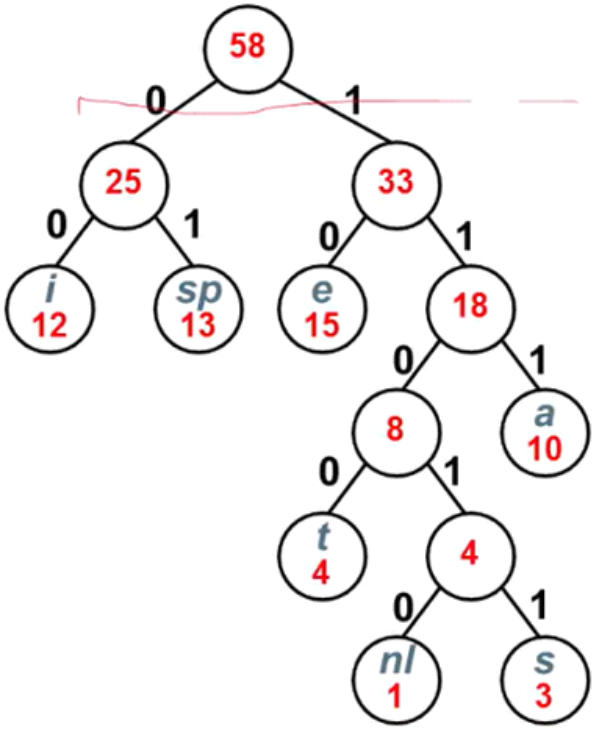
Cost (aaaxuaxz → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

屏幕剪辑的捕获时间: 2023/7/29 7:16

所有字符都在二叉树叶结点上，可以保证都是前缀码

〖例〗哈夫曼编码

C_i	a	e	i	s	t	sp	nl
f_i	10	15	12	3	4	13	1



屏幕剪辑的捕获时间: 2023/7/29 7:16

集合

2023年7月29日 7:16