第6章 优先级队列

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

- **考核大纲**
  - 二叉堆：二叉堆的存储实现、运算实现
  - 归并优先级队列，优先级队列的应用

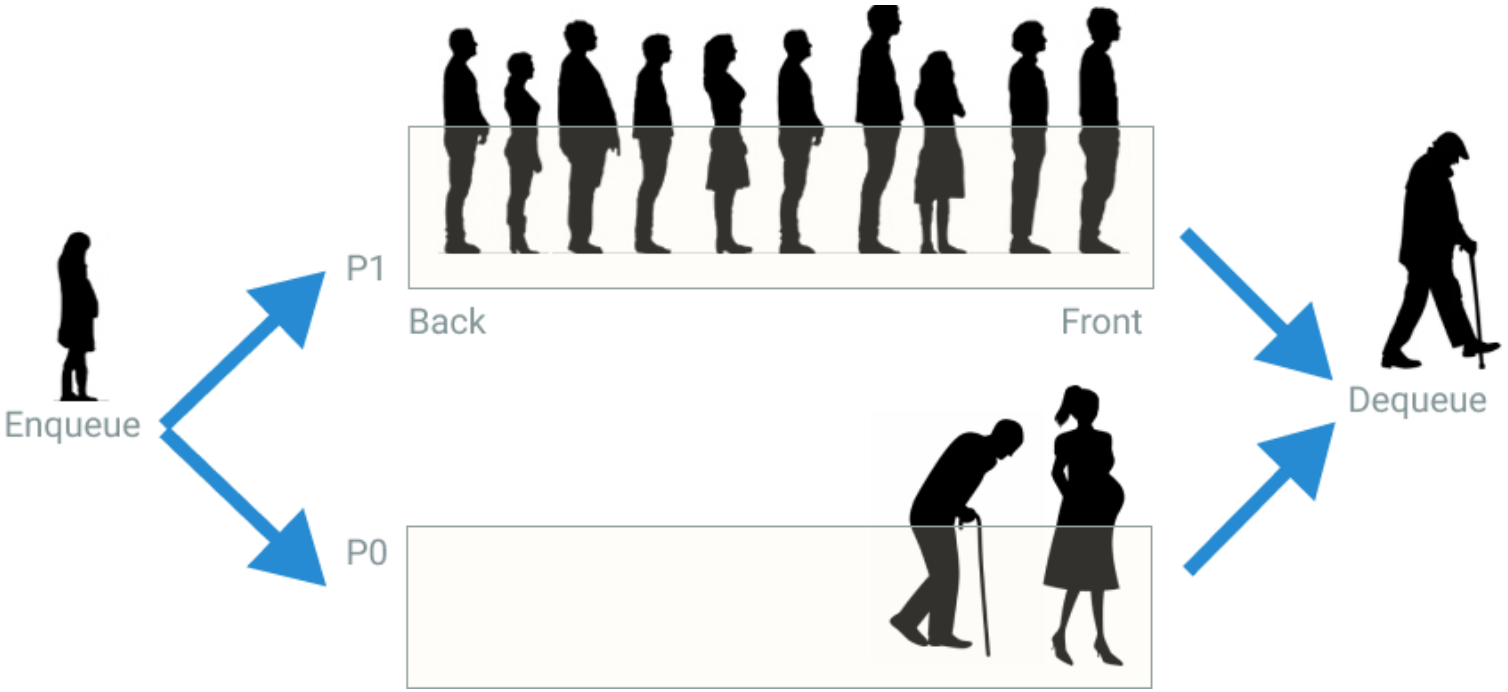
- **复习要点**
  - 熟练掌握二叉堆的实现（A）；了解 D 堆的概念（B）
  - 理解归并优先级队列（B）
  - 运用优先级队列求解经典问题（A）

提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

6.1 问题引入：带优先级的服务处理

- **问题**：第3章介绍的**队列**能够按先到先得的方式来处理，但实际问题中（例如医院或者银行）还有可能需要考虑加急的情况，更一般地说就是**优先级**。如何按优先级进行处理？
- **关键**：出队的顺序需要由元素本身的优先级来决定，而不是进队的顺序。



提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

6.2 优先级队列的定义

- **优先级队列**是一种特殊的队列。与普通队列相比，
  - **相同点**：都支持进队和出队操作
  - **不同点**：优先级队列的出队顺序按事先规定的优先级顺序进行

ADT PriorityQueue {
数据对象：
　元素取自全集 *U* 的可重集合 *E*，表示优先级队列中包含的元素。
数据关系：
　全集 *U* 中的元素须满足严格弱序。
基本操作：
　（省略初始化、销毁、清除内容、判断为空、查询元素个数等操作）
　Insert(*pq*, *x*)：在优先级队列 *pq* 中插入元素 *x*。
　ExtractMin(*pq*)：从优先级队列 *pq* 中删除优先级最高（也就是值最小）的元素，并返回。
　PeekMin(*pq*)：返回优先级队列 *pq* 中优先级最高的元素（元素仍然保留在优先级队列中）。
}

6.2 优先级队列的定义

- 优先级队列**可以**用线性表来实现
  - 然而，使用线性表实现时出队操作需要将当前优先级队列中的所有元素都检查一遍，从而找到优先级最高的元素
  - 假设优先级队列中元素个数为 n，这种实现下出队操作的复杂度是 O(n)，效率较低
- 一般会使用**二叉堆**等更加高效的数据结构来实现优先级队列。
  - 查询优先级最高的复杂度为O(1)

提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

6.3 二叉堆

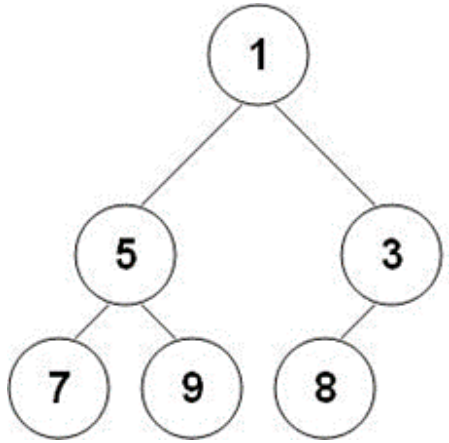
- **二叉堆**是一种常见的堆，常用来实现优先级队列。
- 二叉堆最早由 J. W. J. Williams 于 1964 年提出，作为支持堆排序的一种数据结构。
  - 一种基于**完全二叉树**的高效数据结构
  - 核心特性是“堆序性”，常用于实现优先级队列
  - 支持快速获取 / 删除最值元素
  - 查看最值：最高优先级出队的复杂度为O(1)

6.3.1 二叉堆的定义

- 二叉堆是父结点元素和子结点元素满足一定大小关系的完全二叉树。根据条件不同，可分为最小堆和最大堆。
  - **最小堆**：如果完全二叉树 T 中的所有父子结点对都有父结点的元素不大于子结点的元素，则称 T 为最小堆。
  - **最大堆**：如果完全二叉树 T 中的所有父子结点对都有父结点的元素不小于子结点的元素，则称 T 为最大堆。
  - 最小堆和最大堆的区别只在于父子结点元素之间的大小关系
- 为简便起见，本章中的二叉堆（也包括其他种类的堆）都以最小堆为例进行讲解，最大堆的情况可类推得到。

6.3.1 二叉堆的定义

- 注意到二叉堆是一棵完全二叉树，可以将其保存在一个数组中（使用5.3.4节的约定，根结点是下标为1的元素），并具有以下性质：
  - 结点 i 的左右子结点（如果存在）的下标分别为 2i 和 2i+1。
  - 结点 i 的父结点（如果存在）的下标为 $\lfloor i/2 \rfloor$。
- 如图展示了一个最小堆。其中结点 1 的子结点为结点 5 和结点 3，结点 3 的子结点只有结点 8。
  - 其中任意一个结点上的元素都不大于其子结点元素。



结点	1	5	3	7	9	8
下标	1	2	3	4	5	6

6.3.2 二叉堆的操作

- 二叉堆的基本操作是堆元素的**上调**和**下调**。

（这里的“上”和“下”是指用一般习惯画出二叉堆的树表示后元素在调整过程中的走向）

- 在上调和下调操作的基础上，可实现堆元素的
  - 插入
  - 删除
  - 建堆操作
- 设数组 h.data[] 中保存着二叉堆中的元素。
  - 在对二叉堆进行操作的过程中，可能会出现不满足二叉堆性质的时刻，为表述方便，仍用堆来称呼此时的状态。

6.3.2 二叉堆的操作

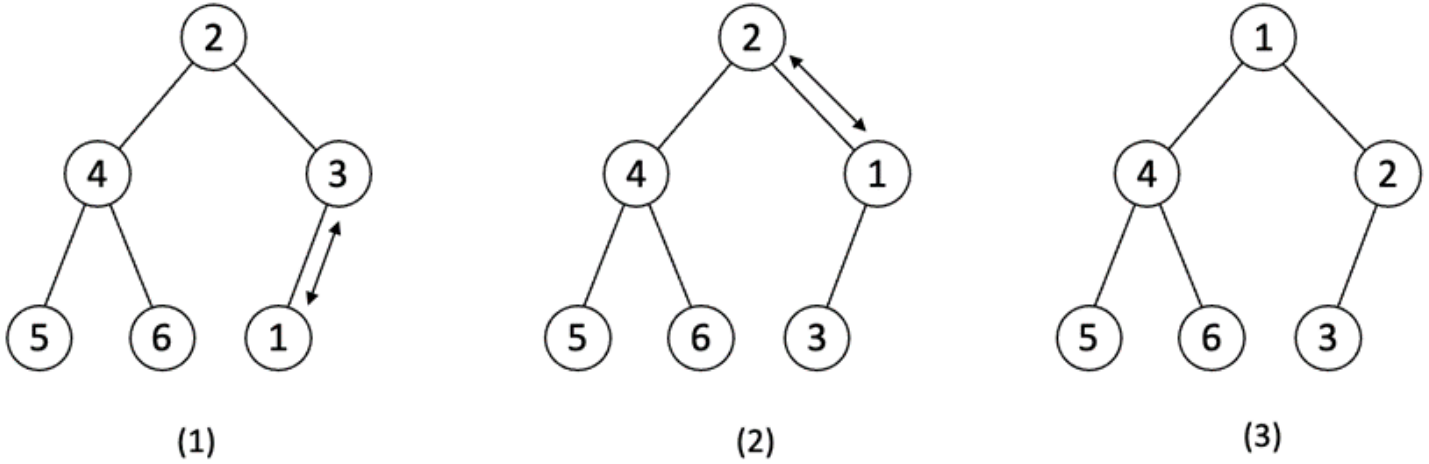
- 常见操作
  - 上调
  - 下调
  - 插入
  - 删顶

```
typedef int HElemSet;
typedef int Position; /* 数组下标即为元素位置 */
typedef struct BinaryHeapNode *MaxHeap;
typedef struct BinaryHeapNode *MinHeap;
struct BinaryHeapNode {
        HElemSet *data; /* 数据元素数组 */
        int size; /* 规模 */
        int capacity; /* 容量 */
};
```

```
void InitPQueue(MinHeap pq, int capacity);
void SiftUp(MinHeap h, Position i);
void SiftDown(MinHeap h, Position i);
void Insert(MinHeap h, HElemSet x);
HElemSet ExtractMin(MinHeap h);
```

6.3.2 二叉堆的操作

- 二叉堆的上调操作
  - 如果堆中某结点 i 小于其父结点 p，此时可以交换结点 i 和结点 p 的元素，也就是把结点 i 沿着堆的这棵树往“上”调整。
  - 此时，再看新的父结点与它的大小关系。重复该过程，直到结点 i 被调到根结点位置或者和新的父结点大小关系满足条件。



如图演示了一次二叉堆的上调操作的过程，元素1从开始的叶结点位置一直调整到了根结点。

6.3.2 二叉堆的操作

- 二叉堆的上调操作
  - 在实现时，可以通过以下方法避免交换，从而减少赋值操作的次数。
    SiftUp 先将结点 i 的元素保存在临时变量中，随着调整将父结点的元素往“下”移动，最后再将原来结点 i 的元素填入合适的位置。

算法6-1： 二叉堆的上调操作 SiftUp(*h*, *i*)
输入： 堆 *h* 和上调起始位置 *i*
输出： 上调后满足堆性质的*h*

```
1.  elem← h.data[i]
2.  while i > 1 且 elem < h[i / 2] do  //当前结点小于其父结点
3.  | h.data[i] ← h.data[i / 2]  //将i的父结点元素下移
4.  | i ← i / 2  //i指向原结点的父结点，即向上调整
5.  end
6.  h.data[i] ← elem
```

  - 对于上调操作而言，循环的次数不会超过树的高度，因此时间复杂度为 O(log n)。

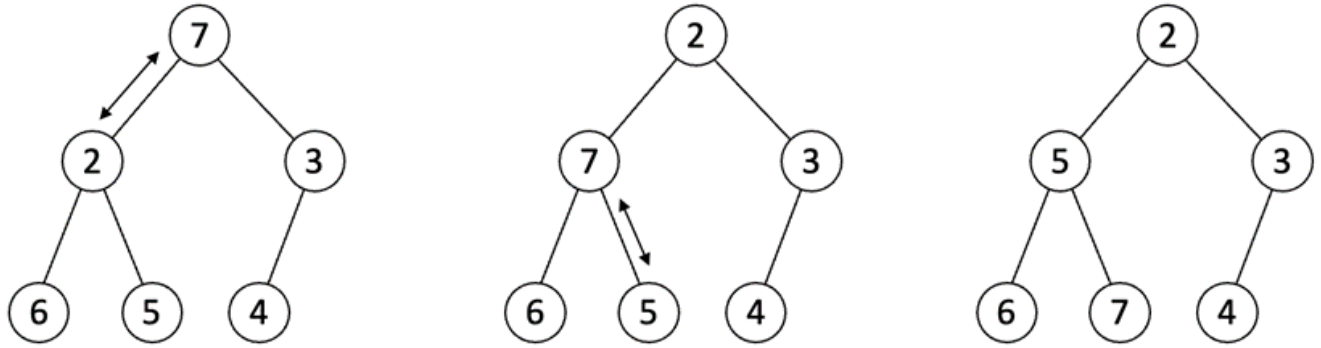
6.3.2 二叉堆的操作

- 二叉堆的上调操作

```
void SiftUp(MinHeap h, Position i)
{
    HElemSet elem;
    elem = h->data[i];
    while (i>1 && elem<h->data[i>>1]) { /* 当前结点小于其父结点 */
        h->data[i] = h->data[i>>1]; /* 将i的父结点元素下移 */
        i >>= 1; /* i指向原结点的父结点，即向上调整 */
    }
    h->data[i] = elem;
}
```

6.3.2 二叉堆的操作

- 二叉堆的下调操作
  - 如果堆中某结点 i 大于其子结点，则要将其向“下”调整。调整时需要注意，对于有两个子结点的情况，如果两个子结点均小于结点 i，交换时应选取它们中的较小者，只有这样才能保证调整之后三者的关系能够满足堆的性质。
  - 重复该过程，直到结点 i 被调到叶结点位置或者大小关系满足条件。
  - 如图演示了一次二叉堆的下调操作的过程，元素7从开始的根结点位置一直调整到了[illegible]素进行交换。



6.3.2 二叉堆的操作

- 二叉堆的下调操作
  - 下调操作同样可以使用上调操作的方法来避免交换操作
  - 对于下调操作，循环次数不会超过树的高度，因此时间复杂度为O(log n)。

算法6-2： 二叉堆的下调操作 SiftDown(*h*, *i*)
输入： 堆 *h* 和下调起始位置 *i*
输出： 下调后满足堆性质的*h*

```
1.  last ← h.size  //这是最后一个元素的位置
2.  elem ← h.data[i]
3.  while true do
4.  | child ← 2i  //child当前是i的左孩子的位置
5.  | if child<last 且h.data[child+1]<h.data[child] then //如果i有右孩子并且右孩子更小
6.  | | child ← child + 1  //child更新为i的右孩子的位置
7.  | else if child>last  //如果i是叶子结点
8.  | | break  //已经调整到底，跳出循环
9.  | end
10. | if h.data[child] < elem then  //若较小的孩子比elem小
11. | | h.data[i] ← h.data[child]  //将较小的孩子结点上移
12. | | i ← child  //i指向原结点的孩子结点，即向下调整
13. | else  //若所有孩子都不比elem小
14. | | break  //则找到了elem的最终位置，跳出循环
15. | end
16. end
17. h.data[i] ← elem
```

```
void SiftDown(MinHeap h, Position i)
{
	Position last, child;
	HElemSet elem;
	last = h->size; /* 这是最后一个元素的位置 */
	elem = h->data[i];
	while (1) {
		child = i<<1; /* child当前是i的左孩子的位置 */
		if (child<last && h->data[child+1]<h->data[child]) { /* 如果i有右孩子并且右孩子更小 */
			child++; /* child更新为i的右孩子的位置 */
		}
		else if (child > last) { /* 如果i是叶子结点 */
			break; /* 已经调整到底，跳出循环 */
		}
		if (h->data[child] < elem) { /* 若较小的孩子比elem小 */
			h->data[i] = h->data[child]; /* 将较小的孩子结点上移 */
			i = child; /* i指向原结点的孩子结点，即向下调整 */
		}
		else { /* 若所有孩子都不比elem小 */
			break; /* 则找到了elem的最终位置，跳出循环 */
		}
	}
	h->data[i] = elem;
}
```

6.3.2 二叉堆的操作

- 二叉堆的插入操作
  - 堆的插入操作就可以实现为向堆中追加待插入的元素（通常放在顺序表最后位置），然后用上调操作将其调整到合适的位置来完成堆的调整

算法6-3： 二叉堆的插入操作 Insert(*h*, *x*)
输入： 堆 *h* 和待插入元素 *x*
输出： 将元素插入后的堆

```
1. h.size ← h.size + 1
2. last ← h.size
3. h.data[last] ← x  //暂时将x放入最后一个元素的位置
4. SiftUp(h, last)  //从最后一个位置上调
```

  - 时间复杂度O(log n)

6.3.2 二叉堆的操作

- 二叉堆的插入操作
  - 堆的插入操作就可以实现为向堆中追加待插入的元素（通常放在顺序表最后位置），然后用上调操作将其调整到合适的位置来完成堆的调整

```
void Insert(MinHeap h, HElemSet x)
{
        Position last;
        if (h->size==h->capacity) { /* 堆已满 */
                printf("错误：堆已满，无法插入。\n");
        }
        else {
                h->size++;
                last = h->size;
                h->data[last] = x; /* 暂时将x放入最后一个元素的位置 */
                SiftUp(h, last);
        }
}
```

6.3.2 二叉堆的操作

- 二叉堆的删除操作
  - 删除操作所要提取的最小元素就是堆中的第一个元素，然后把堆中的最后一个元素挪到第一个位置，并通过下调操作将这个元素调整到合适的位置

算法6-4： 二叉堆的删顶操作 ExtractMin(*h*)
输入： 堆 *h*
输出： *h*中的最小元，以及删除了最小元素后的堆*h*

```
1. min_key ← h.data[1]  //这是将要返回的最小元
2. last ← h.size  //这是删除前最后一个元素的位置
3. h.size ← h.size - 1
4. h.data[1] ← h.data[last]  //暂时将删除前最后一个元素放入根的位置
5. SiftDown(h, 1)  //从根结点下调
6. return min_key
```

  - 时间复杂度是O(log n)。删除操作算法如下所示。

6.3.2 二叉堆的操作

- 二叉堆的删除操作

```
HElemSet ExtractMin(MinHeap h)
{
    HElemSet min_key;
    Position last;
    min_key = h->data[1]; /* 这是将要返回的最小元 */
    last = h->size; /* 这是删除前最后一个元素的位置 */
    h->size--;
    h->data[1] = h->data[last]; /* 暂时将删除前最后一个元素放入根的位置 */
    SiftDown(h, 1); /* 从根结点下调 */
    return min_key;
}
```

6.3.2 二叉堆的操作

- 朴素**建堆**操作
  - 对于任意一组元素，可以通过逐个上调的方式把它们转化为一个堆，相当于依次插入堆中

算法 6-5： 二叉堆的朴素建堆操作 MakeHeapUp(*h*)
输入： 存储在 *h* 中的数据
输出： 满足堆性质的堆*h*

```
1.  last ← h.size  //这是最后一个元素的位置
2.  for i ← 2 to last do
3.  |  SiftUp(h, i)
4.  end
```

```
void MakeHeapUp(MinHeap h)
{
        Position last;
        int i;
        last = h->size; /* 这是最后一个元素的位置 */
        for (i=2; i<=last; i++) { /* 相当于顺次插入元素 */
                SiftUp(h, i);
        }
}
```

  - 使用上述方法进行建堆，堆中后一半的元素进行上调时都可能需要O(log n)的时间，因此总的时间复杂度是O(n log n)。

6.3.2 二叉堆的操作

- **快速建堆**操作
  - 也可以用逐个下调的方式把它们转化为一个堆。由于可以把叶结点跳过，因此是从最后一个有叶子的结点（大概是一半的位置）开始操作

算法6-6： 二叉堆的快速建堆操作 MakeHeapDown(*h*)
输入： 存储在 *h* 中的数据
输出： 满足堆性质的堆*h*

```
1.  last ← h.size  //这是最后一个元素的位置
2.  for i ← last/2 downto 1 do  // last/2是最后一个元素的父结点的位置
3.  |  SiftDown(h, i)
4.  end
```

  - 和MakeHeapUp相比，这样做的好处是在MakeHeapDown中有将近一半结点的下调操作只需要O(1)的时间，因此更加高效。具体分析如下：

$$\sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} O(k) = O\left(n \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \right) = O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k} \right) = O(n)$$

6.3.2 二叉堆的操作

- **快速建堆**操作

```
void MakeHeapDown(MinHeap h)
{
        Position last;
        int i;
        last = h->size; /* 这是最后一个元素的位置 */
        for (i=last>>1; i>0; i--) {
                SiftDown(h, i); /* 自底向上调整 */
        }
}
```

提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

6.4 多叉堆

- **多叉堆定义**
  - 也称为 d 堆（d-ary heap 或 d heap），是二叉堆的推广形式
  - 除边界情况外，二叉堆的每个结点有两个子结点，而多叉堆有 d 个子结点
- **多叉堆存储**
  - 多叉堆可以用数组来保存堆中的元素，元素的下标之间存在数学关系
  - 从数学公式的简洁优雅考虑，多叉堆用数组表示时下标一般从0开始
  - 在多叉堆的数组表示中，元素 0 是根结点，元素 1~d 是根结点的子结点，而紧接着的 d^2 个元素是根结点的孙子结点，以此类推。
- **多叉堆结点之间的下标关系如下：**
  - 结点 i 的 d 个子结点的下标分别为 di+j，其中1⩽j⩽d
  - 结点 i 的父结点的下标为 $\lfloor (i-1)/d \rfloor$

6.4 多叉堆

- **多叉堆的上调操作**
  - 可以根据以上性质扩展二叉堆的 SiftUp 操作，使得它支持多叉堆

算法6-7： 多叉堆的上调操作 SiftUpD(*h*, *d*, *i*)
输入： 堆 *h*、堆的分叉数 *d* 和上调起始位置 *i*
输出： 上调后满足*d*叉堆性质的*h*

```
1.  elem ← h.data[i]
2.  while i > 0 and elem < h.data[(i−1) / d] do
3.  |  h.data[i] ← h.data[(i−1) / d]
4.  |  i ← (i−1) / d
5.  end
6.  h.data[i] ← elem
```

  - 时间复杂度 $O(\log_d n)$

6.4 多叉堆

- **多叉堆的上调操作**
  - 可以根据以上性质扩展二叉堆的 SiftUp 操作，使得它支持多叉堆

```
void SiftUpD(MinHeap h, int d, Position i)
{
      HElemSet elem;
      elem = h->data[i];
      while (i>0 && elem<h->data[(i-1)/d]) { /* 当前结点小于其父结点 */
            h->data[i] = h->data[(i-1)/d]; /* 将i的父结点元素下移 */
            i = (i-1)/d; /* i指向原结点的父结点，即向上调整 */
      }
      h->data[i] = elem;
}
```

6.4 多叉堆

- **多叉堆的下调操作**
  - 扩展二叉堆的 SiftDown 操作，要注意向下比较时要对比所有的子结点
  - 时间复杂度 $O(d\log_d n)$

算法6-8：多叉堆的下调操作 SiftDownD(*h, d, i*)
输入：堆 *h*、堆的分叉数 *d* 和下调起始位置 *i*
输出：将堆 *h* 中的元素下调以满足堆性质

```
1.  last ← h.size - 1  //这是最后一个元素的位置
2.  elem ← h.data[i]
3.  while true do
4.  | child ← i
5.  | for k ← 1 to d do  //找所有孩子中最小的
6.  | | if d×i+k ≤ last 且h.data[d×i+k]<h.data[child] then
7.  | | | child ← d×i+k  //child更新为更小的孩子的位置
8.  | | end
9.  | end
10. | if child = i then  //前面for循环未执行，i是叶结点
11. | | break  //已经调整到底，跳出循环
12. | end
13. | if h.data[child]<elem then  //若最小的孩子比elem小
14. | | h.data[i] ← h.data[child]  //将最小的孩子结点上移
15. | | i ← child  //i指向原结点的孩子结点，即向下调整
16. | else  //若所有孩子都不比elem小
17. | | break  //则找到了elem的最终位置，跳出循环
18. | end
19. end
20. h.data[i] ← elem
```

```
void SiftDownD(MinHeap h, int d, Position i)
{
    Position last, child;
    HElemSet elem;
    int k;
    last = h->size - 1; /* 这是最后一个元素的位置 */
    elem = h->data[i];
    while (1) {
        child = d*i+1; /* child初始化为第1个孩子 */
        for (k=2; child<=last && k<=d; k++) { /* 找所有孩子中最小的 */
            if ((d*i+k)<=last && h->data[d*i+k]<h->data[child]) {
                child = d*i+k; /* child更新为更小的孩子的位置 */
            }
        }
        if (child > last) { /* 前面for循环未执行，i是叶结点 */
            break; /* 已经调整到底，跳出循环 */
        }
        if (h->data[child] < elem) { /* 若最小的孩子比elem小 */
            h->data[i] = h->data[child]; /* 将最小的孩子结点上移 */
            i = child; /* i指向原结点的孩子结点，即向下调整 */
        }
        else { /* 若所有孩子都不比elem小 */
            break; /* 则找到了elem的最终位置，跳出循环 */
        }
    }
```

6.4 多叉堆

- 多叉堆的分析
  - SiftUpD 为 $O(\log_d n)$
  - SiftDownD 为 $O(d \log_d n)$
  - 与二叉堆的对应操作相比，多叉堆的上调操作性能会更好一些，而下调操作则会变差。
  - 对于建堆操作，可以通过数学证明其复杂度仍然为 O(n)。
  - 多叉堆可用于上调操作比下调操作频繁的应用场景，包括图论中的很多常用算法。此外，与二叉堆相比，多叉堆有更好的高速缓存特性，因此实际使用中能够在现代计算机系统结构上取得更好的运行效率。

提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

*6.5 可并堆

- 问题引入：云计算资源调度中，为了节能有时需要将一台物理机中正在运行的虚拟机迁移到另外一台物理机。由于这两台物理机原本都有任务的优先队列在调度，因此在迁移时就需要将两个优先队列合并，即归并（merge）
  - 二叉堆结构，合并操作的实现方式是将其中一个堆（一般是元素较少的那个）中的全部元素逐一插入到另一个堆中，或者直接将两个堆的元素连接在一起再执行一次建堆操作，复杂度都比较高O(n)
- 归并优先级队列（Merging Priority Queue)或可并堆：一种支持**高效合并操作**的**优先级队列**
  - 保留优先级队列的核心能力：O (1) 查看最值，O (log n) 插入 / 删除
  - 新增核心能力：**低复杂度合并**，可做到 O (log n) 或更低
  - 既能像普通优先级队列一样快速获取最值，又能高效合并多个同类队列

*6.5 可并堆

- 左堆（Leftist Heap，左偏堆）**是归并优先级队列的一种经典实现，**完美契合归并优先级队列“高效合并 + 优先级管理”的核心需求
  - **左堆是优先级队列：**一棵“堆序二叉树”（小顶堆或大顶堆），具备优先级队列的所有基础操作：
    - 查看最值：直接访问根节点，时间复杂度O(1)
    - 插入元素：将元素作为新节点与原堆合并，时间复杂度 O(log n)
    - 删除最值：删除根节点后，合并左右子堆，时间复杂度 O(log n)
  - **左堆的核心优势是“高效合并”：**左堆通过“**左偏性质**”（节点的左子树“零路径长度”≥右子树）保证合并操作的高效性：
    - 合并两个左堆时，通过递归将“根值较大的堆”与“根值较小的堆的右子堆”合并，再调整左偏性质。
    - 合并的时间复杂度为O(log n + log m)（n、m分别为两个堆的大小），远优于普通二叉堆 O(n+m)的合并复杂度，完全契合归并优先级队列“低复杂度合并”的核心要求。

*6.5 可并堆

- **归并优先级队列是 “抽象概念”**
  - 描述一类 “支持高效合并的优先级队列”，定义了核心能力（合并、优先级操作），但未限定具体实现。
- **左堆是 “具体实现”：是归并优先级队列的典型实例之一**
  - 常见的可并堆有**左堆**、**斜堆**、**二项堆**、斐波那契堆等实现
  - 左堆因结构简单、合并逻辑清晰，是最常用的归并优先级队列实现之一
- **多数可并堆会将合并操作作为最基本的操作**
  - 插入操作可由原有堆与待插入元素本身构成的单元素堆的合并操作来完成
  - 删除操作则是将原有堆删除结点后所产生的所有分离的子树进行合并

提纲

- 6.1 问题引入：带优先级的服务处理
- 6.2 优先级队列的定义
- 6.3 二叉堆
- 6.4 多叉堆
- *6.5 可并堆
- 6.6 优先级队列应用：Huffman树的构建

6.6 优先级队列应用：Huffman树的构建

- 在构建哈夫曼树的过程中，算法CreateHuffmanTree会地
  - 从一个二叉树集合中取出带权路径长度最小和次小的两棵二持续叉树
  - 把合并后的树加回到集合里
  - 时间复杂度：$O(n^2)$
- 优先级队列的典型应用，可以将二叉树的带权路径长度定为优先级（带权路径长度越小优先级越高）
  - ExtractMin：取出最小、次小
  - Insert：完成从集合以及加回到集合的操作
  - 时间复杂度： O(n log n)
- 为了提高算法的效率，通常使用**二叉堆**作为优先级队列的实现

小结

- **优先级队列定义**
  - 核心约束：元素取自全集U，需满足**严格弱序**（保证优先级可比）；与普通队列的区别是按元素优先级（非进队顺序）出队
  - 核心操作：Insert（插入元素）、ExtractMin（删并返最值）、PeekMin（查最值，不删除），需高效维护优先级
- **二叉堆**
  - 性质：满足“堆序性”的完全二叉树（最小堆：父≤子；最大堆：父≥子），用数组存储（下标从 1 开始，节点i左子2i、右子2i+1、父($\lfloor \frac{i}{2} \rfloor$)）
  - 核心操作：Insert（插入元素）、ExtractMin（删并返最值）、PeekMin（查最值，不删除），需高效维护优先级

小结

- **可并堆（归并优先级队列）**
  - 需求：解决二叉堆合并效率低O(n)的问题，适用于合并多队列场景
  - 左堆实现：满足“堆序性 + 左偏性质”，合并复杂度O(log n+log m)，插入 / 删顶基于合并实现O(log n)
  - 归并优先级队列为抽象概念，左堆是典型实现
- **应用：Huffman 树构建**
  - 优化：用二叉堆实现优先级队列
  - ExtractMin取最小 / 次小树
  - Insert加回合并树
  - 总复杂度从O(n^2)降至O(n log n)

优先级队列高频必刷题（LeetCode）

分类	题号	题目	考察点（高频原因）	难度
优先级队列（二叉堆实现）	703	Kth Largest Element in a Stream	小顶堆维护固定大小元素，二叉堆的插入与删顶操作，数据流场景下优先级队列核心应用，对应 PDF 二叉堆操作考点	Easy
	215	Kth Largest Element in an Array	小顶堆维护前 K 大元素，二叉堆的插入、ExtractMin 操作，高频 Top K 问题，贴合 PDF 二叉堆高效找最值特性	Medium
	347	Top K Frequent Elements	哈希表统计频率 + 小顶堆筛选前 K 高频元素，优先级队列与哈希表结合应用，覆盖 PDF 优先级队列核心操作	Medium
优先级队列应用（哈夫曼树）	1167	Minimum Cost to Connect Sticks	哈夫曼贪心思想（合并最小权值对），二叉堆实现高效取最小元素，直接对应 PDF 中“优先级队列应用：Huffman 树构建”考点	Medium
	1733	Minimum Number of People to Teach	哈夫曼贪心思想应用（合并最小冲突语言对），优先级队列优化决策过程，关联 PDF 哈夫曼树的贪心逻辑	Medium
优先级队列综合应用（合并场景）	23	Merge k Sorted Lists	小顶堆维护多链表最小节点，模拟可并堆的合并逻辑，体现 PDF 中“可并堆”的合并需求，优先级队列在多数据源合并中的典型应用	Hard