

对数据（元素/Object）要解决的问题?

对数据在哪里进行存储?

存储结构如何? 该结构涉及的相关操作怎么样?

### 背景:

Heap 也是 ADT, Heap 是个性质 (max heap, min heap)

PQ 可以用 heap 来实现 (改 priority, 按 priority 排序)

Heap 可以用树来实现, 也可用指针/数组来实现

Lecture 2—研究堆 Heap 的两个应用:

1. 排序: 堆排序 (Heap Sort)
2. 优先队列 (Priority Queue)

优先队列(PriorityQueue) 是一种抽象数据结构 ADT, 队列内的元素都有对应的优先级, 并根据其优先级出队。

### Priority Queue 优先队列能解决的问题:

优先队列是一种队列, 它满足优先级别最高的先出队列。它的一个重要的应用时海量数据的排序, 当我们需要找出10亿数据中的Top10项, 不实际的解决方案是对所有数据排序, 取Top10项。使用优先队列可以在很小的辅助空间内找出Top10项。具体思路如下:

- (1) 建立大小为k的容器
- (2) 每次从大数据集中读取一个数据项
- (3) 如果容器没有满, 则把当前数据项加入容器
- (4) 如果容器已经满, 找出容器中最小的元素 (利用优先队列特性) 如果当前数据项大于最小元素, 则最小元素出队, 当前数据项入队
- (5) 重复 (2) 到 (3) 直到读取完大数据集

上述的方案可以以并行的方式运行, 最后合并结果。

优先度列的五种实现方式, 分别如下

1. 无序数组
2. 有序数组
3. 链表
4. 堆
5. 二叉搜索树

#### **【优先队列的数组实现】**

用数组实现优先队列: (insert 已经进行有序操作。pop 操作即为 peekMax, remove 操作)

## 无序数组的实现

无序数组实现方式的入队操作，直接把入队元素加到数组尾部。出队需要遍历数组，找出优先级别最高的出队，空缺的位置由后面元素依次补上。因此，入队的时间复杂度为 $O(1)$ ，出队为 $O(N)$ 。

## 有序数组的实现

由于要求数组有序，因此在插入的时候需要保存有序，插入操作需要找到适合的位置，然后在该位置插入，位置后面的元素依次往后移动，时间复杂度为 $O(n)$ 。而出队，由于序列是有序，可以在 $O(1)$ 内出队。

相关 java code 参见笔记

The image contains two screenshots of a presentation slide. Both screenshots have a green header bar with the text "Data Structures for Priority Queues: Lists".

**Unsorted List:**

- $Insert(PQ, x, p)$ :  $\Theta(1)$
- $FindMax(PQ)$ :  $\Theta(n)$
- $ExtractMax(PQ)$ :  $\Theta(n)$
- $IncreaseKey(PQ, x, k)$ :  $\Theta(1)$

**Sorted List (by priorities):**

- $Insert(PQ, x, p)$ :  $\Theta(n)$
- $FindMax(PQ)$ :  $\Theta(1)$
- $ExtractMax(PQ)$ :  $\Theta(1)$
- $IncreaseKey(PQ, x, k)$ :  $\Theta(n)$

CSC263 | University of Toronto 4

csc263 lecture2 ppt pg4-5

## 优先队列的堆实现

(二叉) 堆是一个数组 (Array)，它可以看成一个近似的完全二叉树 (Complete Binary Tree)。树上的每一个结点对应数组中的一个元素。除了最底层外，该树是完全充满的，而且是从左向右填充的。

该数组 (Array) 是个特殊的数组 (有一定的大小比较，但并不是完全 sort)，是用来表示堆的以完全二叉树来展现一个最大堆/最小堆 (相对而言，完全二叉树表示的更为直观形象贴切)，用数组 Array 来实现最大堆/最小堆

## 6.1 堆

如图 6-1 所示，(二叉)堆是一个数组，它可以被看成一个近似的完全二叉树(见 B. 5.3 节)。树上的每一个结点对应数组中的一个元素。除了最底层外，该树是完全充满的，而且是从左向右填充。表示堆的数组  $A$  包括两个属性： $A.length$ (通常)给出数组元素的个数， $A.heap\_size$  表示有多少个堆元素存储在该数组中。也就是说，虽然  $A[1..A.length]$  可能都存有数据，但只有  $A[1..A.heap\_size]$  中存放的是堆的有效元素，这里， $0 \leq A.heap\_size \leq A.length$ 。树的根结点是  $A[1]$ ，这样给定一个结点的下标  $i$ ，我们很容易计算得到它的父结点、左孩子和右孩子的下标：

```

PARENT(i)
1 return  $\lfloor i/2 \rfloor$ 

LEFT(i)
1 return  $2i$ 

RIGHT(i)
1 return  $2i+1$ 

```

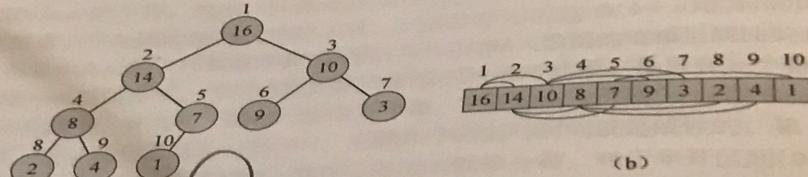


图 6-1 以(a)二叉树和(b)数组形式展现的一个最大堆。每个结点圆圈内部的数字是它所存储的数据。结点上方的数字是它在数组中相应的下标。数组上方和下方的连线显示的是父-子关系：父结点总是在它的孩子结点的左边。该树的高度为 3，下标为 4(值为 8)的结点的高度为 1

在排序算法和优先队列中应用最大堆/最小堆

增删改查的某些相关操作会破坏最大/最小堆的性质

删除很特殊，衍生到删除最大/最小

- **堆构建 (建堆) - Build Heap ( $O(n)$ )**

(先将数据源的数据元素单纯按照完全二叉树从左至右依次填入，在按照最大堆/最小堆性质进行调整-Bubble Down, Bubble Up, Build-Max-Heap 这个过程调用了 Max-Heapify)

### 3. 最小堆的构建

初始数组为：9,3,7,6,5,1,10,2

按照完全二叉树，将数字依次填入。

填入后，找到最后一个结点（本示例为数字2的节点），从它的父节点（本示例为数字6的节点）

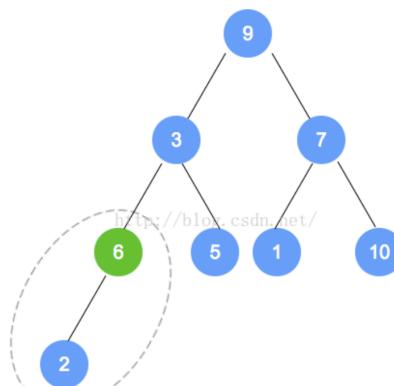
开始调整。根据性质，小的数字往上移动；至此，第1次调整完成。

注意，被调整的节点，还有子节点的情况，需要递归进行调整。

第二次调整，是数字6的节点数组下标小1的节点（比数字6的下标小1的节点是数字7的节点），

用刚才的规则进行调整。以此类推，直到调整到根节点。

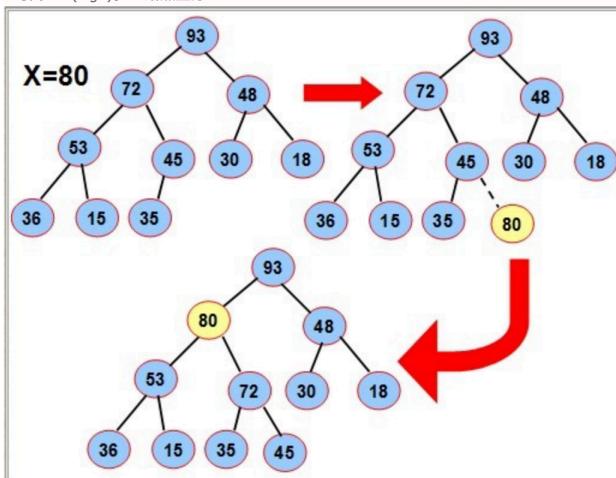
以下是本示例的图解：



- 插入 - Insert( $O(\lg n)$ )

(1) 最大堆的插入

由于需要维持完全二叉树的形态，需要先将要插入的结点x放在最底层的最右边，插入后满足完全二叉树的特点；然后把x依次向上调整到合适位置满足堆的性质，例如下图中插入80，先将80放在最后，然后两次上浮到合适位置。时间： $O(\lg n)$ 。“结点上浮”



- 维护堆的性质：Max-Heapify( $A, i$ ) ( $O(\lg n)$ )

(Max-Heapify 是用于维护 Max Heap 性质的重要过程，它的输入是一个 Array 数组和一个 index 下标  $i$ ，过程 Build-Max-Heap 对树中的其他结点都调用一次 Max-Heapify)

- 伪代码
- 执行过程，实现原理
- time-complexity

```

MAX-HEAPIFY (A, i)
1   l = LEFT (i)
2   r = RIGHT (i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY (A, largest )
    
```

图 6-2 图示了 MAX-HEAPIFY 的执行过程。在程序的每一步中，从  $A[i]$ 、 $A[\text{LEFT}(i)]$  和  $A[\text{RIGHT}(i)]$  中选出最大的，并将其下标存储在  $\text{largest}$  中。如果  $A[i]$  是最大的，那么以  $i$  为结点的子树已经是最大堆，程序结束。否则，最大元素是  $i$  的某个孩子结点，则交换  $A[i]$  和  $A[\text{largest}]$  的值。从而使  $i$  及其孩子都满足最大堆的性质。在交换后，下标为  $\text{largest}$  的结点是原来的  $A[i]$ ，于是以该结点为根的子树又有可能会违反最大堆的性质。因此，需要对该子树递归调用 MAX-HEAPIFY。

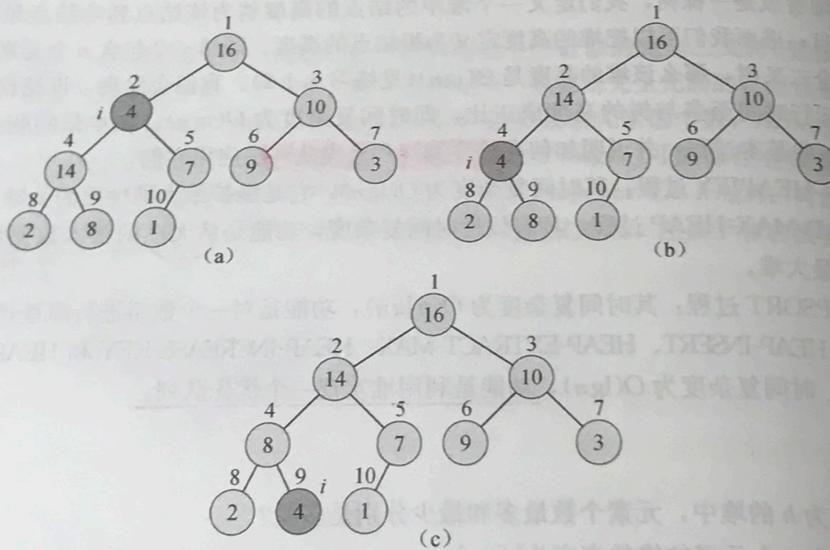


图 6-2 当  $A.\text{heap-size}=10$  时，MAX-HEAPIFY( $A, 2$ )的执行过程。(a)初始状态，在结点  $i=2$  处， $A[2]$  违背了最大堆性质，因为它的值不大于它的孩子。在(b)中，通过交换  $A[2]$  和  $A[4]$  的值，结点 2 恢复了最大堆的性质，但又导致结点 4 违反了最大堆的性质。递归调用 MAX-HEAPIFY( $A, 4$ )，此时  $i=4$ 。在(c)中，通过交换  $A[4]$  和  $A[9]$  的值，结点 4 的最大堆性质得到了恢复。再次递归调用 MAX-HEAPIFY( $A, 9$ )，此时不再有新的数据交换。

- 建堆：Build-Max-Heap( $A$ ) ( $O(n)$ )

- 伪代码
- 执行过程，实现原理
- time-complexity
- prove correctness

### 3 建堆

我们可以用自底向上的方法利用过程 MAX-HEAPIFY 把一个大小为  $n = A.\text{length}$  的数组  $[1..n]$  转换为最大堆。通过练习 6.1-7 可以知道，子数组  $A(\lfloor n/2 \rfloor + 1..n)$  中的元素都是树的叶结点。每个叶结点都可以看成只包含一个元素的堆。过程 BUILD-MAX-HEAP 对树中的其他结点都调用一次 MAX-HEAPIFY。

```
BUILD-MAX-HEAP( $A$ )
1  $A.\text{heap-size} = A.\text{length}$ 
2 for  $i = \lfloor A.\text{length}/2 \rfloor$  down to 1
3     MAX-HEAPIFY( $A, i$ )
```

- 堆排序：HeapSort( $A$ ) ( $O(n \lg n)$ )

- 伪代码
- 执行过程，实现原理
- time-complexity

## HEAPSORT( $A$ )

```

1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  down to 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap\_size = A.heap\_size - 1$ 
5      MAX-HEAPIFY( $A$ , 1)

```

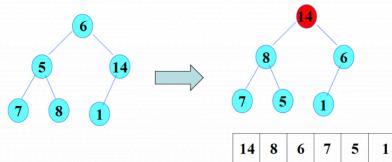


HIT  
CS&E

### Heapsort Algorithm

- Basic idea

6	5	14	7	8	1
---	---	----	---	---	---

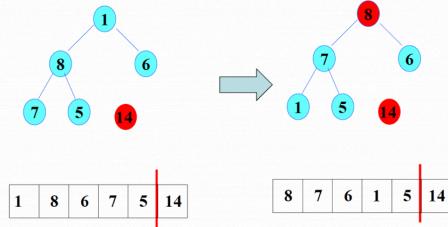


现在要完成6个整数{6, 5, 14, 7, 8, 1}的大根堆排序;

这里左侧是建立完全二叉树,然后调整为大根堆结构; 开始的位置是第一个非叶节点及其子节点;



HIT  
CS&E

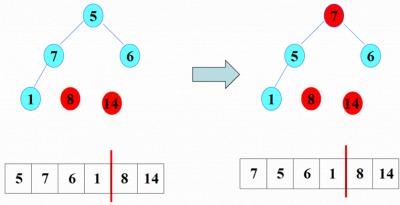


上图是要排序时,将大根堆的根节点(这里是14)从二叉树中去掉,然后用最后一个叶子节点(这里是1)填充,变成了上图左侧的结构,然后再调整为大根堆,就是上图右侧二叉树;

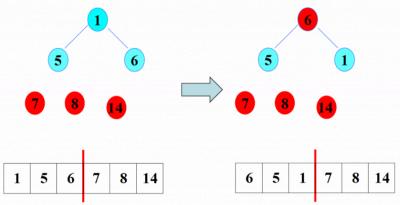
下面几幅图操作相同.

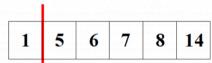
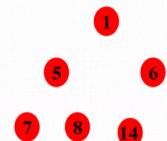
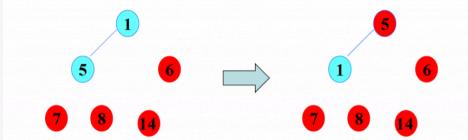


HIT  
CS&E



HIT  
CS&E





通过这个例子不难看出，主要思想是将大根堆的堆顶取出，放到数组的最后一个位置，将最后一个位置原来的数放到堆顶，然后对堆顶做调整；

## 2. Heap 堆如何基于最大堆 MaxHeap 实现最大优先队列

Heap-Maximum(A): **big-theta(1)**  
 Heap-Extract-Max(A): **O(lgn)**  
 Heap-Increase-Key(A,i,key): **O(lgn)**  
 Max-Heap-Insert(A,key): **O(lgn)**

### HEAP-MAXIMUM(A)

1 **return A[1]**

过程 HEAP-EXTRACT-MAX 实现 EXTRACT-MAX 操作。  
循环体部分(第 3~5 行)很相似。

### HEAP-EXTRACT-MAX(A)

```

1 if A. heap-size < 1
2     error "heap underflow"
3 max = A[1]
4 A[1] = A[A. heap-size]
5 A. heap-size = A. heap-size - 1
6 MAX-HEAPIFY(A, 1)
7 return max

```

HEAP-INCREASE-KEY( $A$ ,  $i$ ,  $key$ )

1   **if**  $key < A[i]$

2           **error** "new key is smaller than current key"

3    $A[i] = key$

4   **while**  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$

5           exchange  $A[i]$  with  $A[\text{PARENT}(i)]$

6            $i = \text{PARENT}(i)$

MAX-HEAP-INSERT( $A$ ,  $key$ )

1    $A.\text{heap-size} = A.\text{heap-size} + 1$

2    $A[A.\text{heap-size}] = -\infty$

3   HEAP-INCREASE-KEY( $A$ ,  $A.\text{heap-size}$ ,  $key$ )