数据结构与算法分析

华中科技大学软件学院

2017年秋

大纲

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

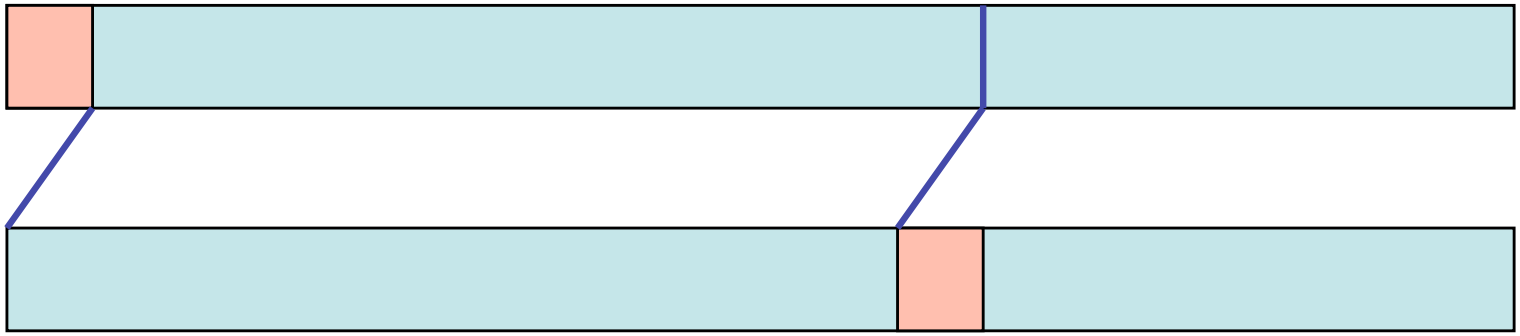
实验五

- 首先需要定位根节点：先序序列遍历的第一个元素
- 树根把中序序列分为3部分 {左子树列}树根{右子树列}
- 以上分别给出了先序序列的一个划分
- 使用左子树、右子树的遍历结果来重建子树

定位根结点

```
static int find_root (char a[], int n, char name)
{
    int i;

    for (i = 0; i < n; i++)
    {
        if (a[i] == name)
            return (i);
    }

    /* not found */
    return (-1);
}
```

树的划分



构建二叉树

```
static NODE *construct (char a[], char b[], int n)
{
    NODE *root;
    int k, l;

    /* trivial case: tree is empty */
    if (n == 0)
        return (NULL);

    root = (NODE *) malloc (sizeof (NODE));
    if (root == NULL)
        return (root);

    root->name = a[0];
    k = find_root (b, n, a[0]);
    if (k < 0)
    {
        printf ("node␣%c␣not␣found␣in␣in-order␣traversal\n", a[0]);
        return (NULL);
    }

    root->left = construct (&a[1], b, k);
    root->right = construct (&a[k + 1], &b[k + 1], n - k - 1);

    return (root);
}
```

复杂度分析

- 平凡情况：$T(0) = 1$
- 递归式：$T(n) = T(k) + T(n-k-1) +$ 非递归运算
  - 最佳情况：只有右子树，
    $T(n) = T(n-1) + O(1)$，$O(n)$
  - 最坏情况：只有左子树，
    $T(n) = T(n-1) + O(n)$，$O(n^2)$
  - 平衡情况：$T(n) = 2T(n/2) + O(n)$，$O(n \log n)$

课程计划

- 已经学习了
  - 树的表示
  - 二叉树与遍历
  - BST的操作
  - AVL树

课程计划

- 已经学习了
  - 树的表示
  - 二叉树与遍历
  - BST的操作
  - AVL树
- 即将学习
  - 散列
  - 冲突的处理
  - 再散列

Roadmap

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

为什么需要散列

- 对序列使用二进制查找用时log n
- 但若考虑数组访问：A[i]
  - 给定i，输入地址，例如，&A[0]+i*4 为整型
  - 然后访问在&A[0]+i*4 的值
- 常数 + 常数 = 常数时间
- 如果元素的键(keys)是整型，那么将元素存储在A[key]
- 可在常数时间内按键(key)查找

折半查找与数组

- 数组访问比折半查找还快
- 折半查找假定所有元素已被排序
- 如果所有的关键字都在范围之内，为什么不对应地存入数组？这样就可以在常数时间里访问
- 问题在于：
  - 通常，关键字可能超出存储空间的范围
  - 有时关键字也不是数字类型，例如，可能是字符串
- 引入Hashing解决上述问题

Hashing

- 另一种用于字典探测的数据结构：基于带有给定的数字键的项，支持快速插入、删除、查找
- 不支持：
  - 快速地 findMin, findMax, print-in-sorted-order
- 哈希表： array[0..n-1] + 哈希函数
  - 函数将项x映射为[0..n-1]
  - 插入：将x存入 h(x)
  - 查找：找到pos h(x)中的x

Hashing的效果

- 数组访问是常数时间
- 若数组大小 >= key计数，则纳入所有(room for all)
- 访问算法:
  - 计算 h(x) - 常数时间
  - 计算 h(x)地址 - 常数时间
  - 访问存储的元素 A[h(x)] - 常数时间
- 数组访问是 O(1)
  - 非常快
  - 优于 logn

选择哈希函数

- 目的：平等使用所有数组项
  - 哈希函数向四周传播输出
  - 输入是“哈希的” —— 经排序的(“hashed” - scrambled up)
- 输入是有限的，键是无限的
- 数组是有一定长度的
- 每个数组单元格多个项目
  - 尝试均匀划分
  - 每个单元格 = 关联的列表
  - 访问时间 = O(键计数/数组长度) = O(1)
- 现在：选择哈希函数

哈希函数

- 将整数键映射到 [0..size-1]
- 显然的想法: 模数(modulus), h(x) = x % size
- 例: 散列价格以美分为单位来调整1000表的size
  19.95 → 995, 29.95 → 995, 39.95 → 995
- 举例, 若键值是size的乘积, 则全部映射到A[0]
- 问题: 键中的结构仍保持散列
- 可为每个h(x)调用随机数生成器, 但随后h(x)肯定会出现相同项
- 更优(简单)解: 选size = 大素数 , 如 1007
  19.95 → 998, 29.95 →981, 39.95→ 974

设计哈希函数

- 一个好的哈希函数应有如下特征
  - 低成本
  - 均匀性
  - 确定性，连续性/离散性
- 除表的查找以外的其他应用
  - 密码学
    - 无冲突
    - 易于计算
    - 难以插入
  - 数据完整性
    - 信息摘要
    - 校验和
  - 电子签名

哈希函数的特性

- 压缩 - h 将一个任意bit长度的输入x映射为固定数量的bits h(x)
- 易于计算 - 给定 h 和 x, 易算出 h(x)
- 前像抗性(Pre-image resistance) - 给定y, 在计算上不可能找到这样的x,使h(x) = y
- 冲突抗性(Collision resistance) - 不可能找到2个不同的x和x', 使得h(x) = h(x')

一些哈希函数

Name	Bitlength	Rounds × Steps per round	Relative speed
MD4	128	3 × 16	1.00
MD5	128	4 × 16	0.68
RIPEMD-128	128	4 × 16 twice (in parallell)	0.39
SHA-1	160	4 × 20	0.28
RIPEMD-160	160	5 × 16 twice (in parallel)	0.24

冲突的频率

- 有多少冲突?
  - 尽可能保持量少
  - 很难完全避免
- 生日问题: n个人中有 >= 2人生日相同的概率是多少?
- 计算所有n都是不同的概率并相减

$$\frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{(366-n)}{365} = \frac{365!/(366-n)!}{365^n}$$

- 概率(Probability) = 1 - $365!/(366-n)!/365^n$

Roadmap

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

哈希表中的冲突

- 假定散列函数(hash)将项(item)均匀地映射到单元格(cells)
- 每m个 单元格(cells) 得到 项(item) 的1/m
  - 插入第一项 x1 到 h(x1), 现在得到第二个
  - 位置不同的概率 = (m-1)/m
  - 第三个单元格不同的概率: (m-2)/m
- 全部不同的概率:
  $(m-1)/m*(m-2)/m*...*(m+1-n)/m = (m!/(m-n!))/m^n$
- 可表示: $n = 1.177 * \sqrt{m} \rightarrow Pr(collision) > \frac{1}{2}$
- $m = 1,000,000$, $n = 1177 \rightarrow Pr(collision) > 0.5$

分离链路法

- 对于所有的哈希函数，选择将冲突最小化的一个
- 其他问题：如何处理冲突
- 简单想法：每个元素都是一个关联的列(linked list)
  - 插入：插入到单元格的关联列(linked list)
  - 查找：找到单元格，遍历它的列
- 大表格，妙函数 $\rightarrow$ 短列集

哈希表

- H(x) = x % 7, 插入 11, 12, 13, 14, 15, 16, 17, 18 , 19, 20

哈希表

- H(x) = x % 7，插入 11，12，13，14，15，16，17，18 ，19，20

0	->14	-> ∥	
1	->15	-> ∥	
2	->16	-> ∥	
3	->17	-> ∥	
4	->11	->18	-> ∥
5	->12	->19	-> ∥
6	->13	->20	-> ∥

装填因子

- 负载系数 λ= 比率(ratio)e（元素计数）/（表格大小）= 列的平均长度
- 不成功的查找：λ 对比
- 一个成功列的理想运行时间：$1 + \lambda/2$
  - 每次匹配(match)各1
  - 到达合适位置之前对列扫描的平均长度：$\lambda/2$

开放地址法

- 无独立的数据结构
  - 没有创建新节点
  - 另一方面：需要更小的 λ <0.5
- 使用辅助功能：如果 h(x) 是满的，用 h1(x) = h(x)+f(1) % 表的大小，h2(x)=h(x)+f(2) % 表的大小，等等，直到找到一个空闲单元
- 需要使用等价检测正确的那个
  - f = “冲突解决策略”
  - 最简策略，线性探测：f(i) = i
- Eg：插入 89, 18, 49, 58, 69 到一个 size-10 的表

线性探测

- $h_i(x) = h(x) + i$, where $h(x) = x \% 10$
- 插入 89, 18, 49, 58, 69

线性探测

- $h_i(x) = h(x) + i$, where $h(x) = x \% 10$
- 插入 89, 18, 49, 58, 69

49
58
69
18
89

平方探测

- 我们可以证明：对于二次探查，期望时间更少 $f(i) = i^2$
- 用这种方式插入 89, 18, 49, 58, 96
- 即使没有填充表，二次探测也可能无法插入一个键
- 例如，在一个大小为7的表中，插入0，7，14，21和28，我们将得到重复的位置序列来查询
- 对于 n = 7*q + r (r = 0..6)，对于 n^2 % 7 只有4个可能的值：0，1，2，4

平方探测定理

Theorem

只要表大小是素数，则表至少有一半是空的($\lambda < 0.5$)，恒成立

证明：对于所有 0 < i，j < 1/2 层(表的大小)，$H(x) + i^2 = H(x) + j^2$(% 表的大小)to $i^2 - j^2 = 0$ (% 表的大小) $\rightarrow$ (i - j)*(i + j) = 0(% 表的大小) $\rightarrow$ i = j，给定的表大小是素数 且 i+j < 表的大小. 也就是说，每个x可能被放置在1/2的(表大小)位置(i=0、1、…、1/2(表大小)都导致不同的探测位置)，而这不能全部填在一张表上 $\lambda < 0.5$.

平方探测中的删除

- 删除问题:如果删除临时数字，可能会在此之后丢失(冲突)数字，　懒惰删除
- 删除总是更困难

Roadmap

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

再散列

- 当表变得更满时，访问(所有方法)需要更长的时间
- 结果：最好创造更大的表
- 当速度太慢或插入失败时
  - 用矢量来做:创建双重大小的表，
  - 浏览原始表，计算(新)散列，复制到新表中的正确位置
- Amortized(摊还)： 对每个操作增加一个常数时间
- 摊还分析:随时间分摊成本
- 从16、13、15、2、27、34、66开始 列入 0..19

字符串作为关键字

- 字符串作为关键词是常见的情况
- 为整数定义的散列函数，因此必须将字符串 $\rightarrow$ 转化为整型数
- 一个简单的想法是:将一个字符串中的字符的ASCII值累加

哈希函数代码

```
int hash (char key[], int n, int size)
{
    int hash = 0;

    for (int i = 0; i < n; i++)
        hash += key[i];

    return (hash % size);
}
```

10 chars, about 128^{10} strings, but hash range:
0..1280 $\rightarrow$ < 1300 keys

改进的函数代码

$\sum_{i=0}^{n} s[n-i-1]*31^i$, recall:
$k0 + 31*k1 + 31^2*k2 = ((k2)*31 + k1)*31 + k0$

```
int hash (char key[], int n, int size)
{
    int hash = 0;

    for (int i = 0; i < n; i++)
        hash = 31*hash + key[i];
    hash %= size;
    if (hash < 0)
        hash += size;

    return (hash);
}
```

使用随机数

```
unsigned char Rand8[256];      // This array contains a random
                                  permutation from 0..255 to 0..255
int Hash(char *x) {            // x is a pointer to the first char;
     int h;                    //    *x is the first character
     unsigned char h1, h2;

     if (*x == 0) return 0;    // Special handling of empty string
     h1 = *x; h2 = *x + 1;     // Initialize two hashes
     x++;                      // Proceed to the next character
     while (*x) {
          h1 = Rand8[h1 ^ *x]; // Exclusive-or with the two hashes
          h2 = Rand8[h2 ^ *x]; //    and put through the randomizer
          x++;
     }                         // End of string is reached when *x=0
     h = ((int)(h1)<<8) |      // Shift h1 left 8 bits and add h2
          (int) h2 ;
     return h ;                // Hash is concatenation of h1 and h2
}
```

Roadmap

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

伪随机数发生器

- 线性同余算法

$$X_t = (aX_{t-1} + c) \mod m, t = 1, 2, \ldots$$

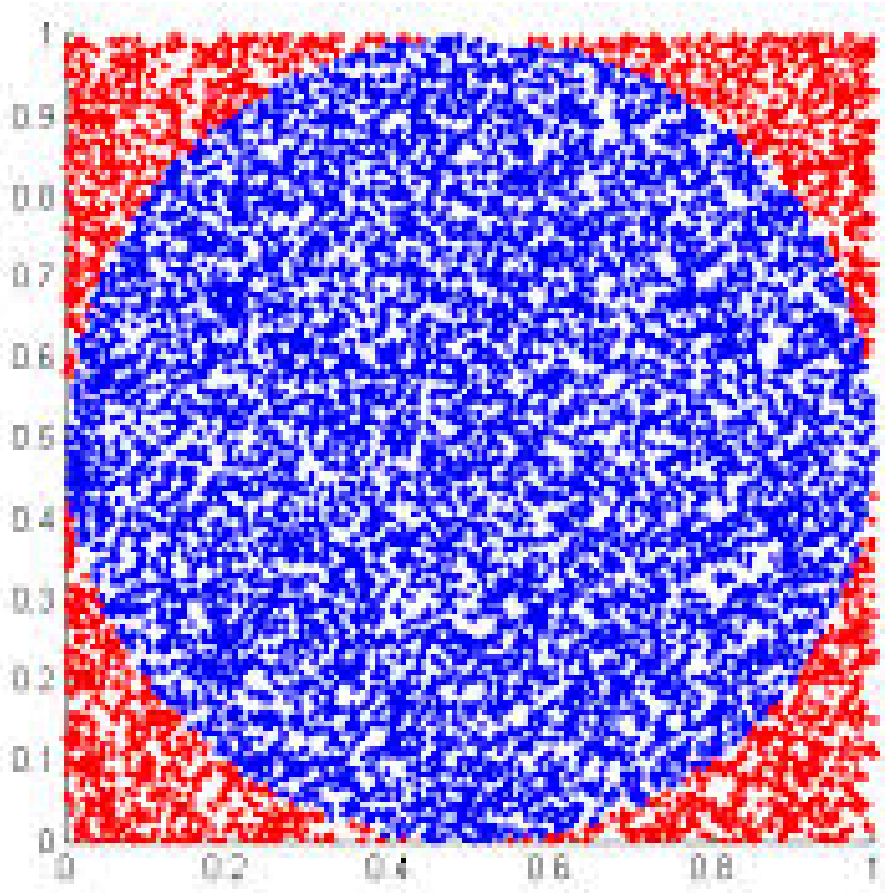
$$U_t = \frac{X_t}{m}$$

- 常数的选择:
  $a = 7^5 = 16807, c = 0, m = 2^{31} - 1 = 2147483647$

Monte Carlo方法

- 蒙特卡罗方法遵循一个典型的过程:
  - 定义一个可能的输入域
  - 在域上随机生成一个概率分布的输入
  - 对输入执行确定性计算
  - 累计结果

圆周率计算



字符串匹配

- 给定字符串文本,搜索字符串模式情况，模式可以是一个正则表达式，但假设只是另一个字符串
- 举个例子:
  - 在莎士比亚的著作中寻找 "to be or not to be" 这句完整的话
  - 在网页/ Word文档中寻找单词
  - 在基因组中找到DNA片段
- 不同于字典问题-链表/数组/ BST
  - 字符串不是离散的集合
  - 这种模式在字符串中出现*多个*字符

匹配算法

- naive算法:

```
for i = 0 to n-1
    for j = 0 to 模式大小-1
        看到 t[i], p[j]
```

时间复杂度: n*m

- Rabin-Karp 算法:

```
for i = 0 to n-1-len(p)
    观察 hash(t[i..i+len(p)-1]), hash(p)
        如果匹配, 则查看字符串
```

Rabin-Karp

- 似乎更好了，但如何获得hash(t[i..i+len(p)-1])s? naive算法：顺序遍历，复制字符，每次都发送给hash-time len(p)，总复杂度：n*m!
- 更好的方法:选择散列以便可以增量地计算。 如果是对字符的求和，那么从一个到下一个：减去第一个，添加下一个
- 如果是幂的和，使幂递减， 减去第一项，乘以质数(“左移”)， 添加下一个
- 每种转化的复杂度都为0(1)
- 现在的总复杂度： 0(n)

Rabin-Karp

- 从 “to be or not to be” 中寻找 “not to”
  - 原文：116 111 32 98 101 32 111 114 32 110 111 116 32 116 111 32 98 101
  - 模式：110 111 116 32 116 111
  - 模式的和：596

116 111 32 98 101 32 = 490
111 32 98 101 32 111 = 490 - 116 + 111 = 485
32 98 101 32 111 114 = 485 - 111 + 114 = 488
98 101 32 111 114 32 = 488 - 32 + 32 = 488
101 32 111 114 32 110 = 488 - 98 + 110 = 500
32 111 114 32 110 111 = 500 - 101 + 111 = 510
111 114 32 110 111 116 = 510 - 32 + 116 = 594
114 32 110 111 116 32 = 594 - 111 + 32 = 515
32 110 111 116 32 116 = 515 - 114 + 116 = 517

110 111 116 32 116 111 = 515 - 32 + 111 = 596

小结

- 数组访问的复杂度为O(1)
- 许多可能的关键词 $\rightarrow$ 用哈希函数映射到数组的位置， 把 x 储存在 A[h(x)]
- 冲突是可能的 $\rightarrow$ 一定要用确定的策略解决
  - 分离链路法(Separate chaining)
  - 开放地址法(Open addressing)
- 非整数关键词必须映射到整数中

装填因子

- 装填因子 λ = 比例:
  - 填入表中的元素个数 / 哈希表的长度
  - = 平均列表长度
- 预计总访问时间: $1 + \lambda/2$
- 对于固定元素计数, 1 + 元素标号/2*size = O(1)
- 对于链路: 保持 $\lambda <= 1$
- 对于二次探测: 保持 $\lambda <= 1/2$, 因为如果 $\lambda > 1/2$ 则可能失败

散列操作的复杂度

- 搜索/插入/删除
  - 平均情形：O(1)
  - 最坏情形：O(n)
- 使用一个好的哈希函数，最坏的情况是几乎不可能出现的
- 一般情况下，得到常数非常小的O(1)

Roadmap

1. 散列
2. 冲突的处理
3. 再散列
4. 随机数算法
5. 堆和优先队列

优先队列

- 我们希望将任务的优先级分配到队列中，以实现单一目标：将总成本降至最低
- 想想看，你站在餐厅的一条线上，每个人都花不同的时间做订单
- 直观上看，优先级对应于花费时间的倒数

优先队列的操作

- 两个典型操作： 插入，删除最小元（优先级最高者）
- 直接实现:使用链表来保存数据、查找最小元并删除它
- 这需要O(n)时间，比必要的时间长
- 那使用BST如何呢,查找最小元只需要log(n)

二叉堆

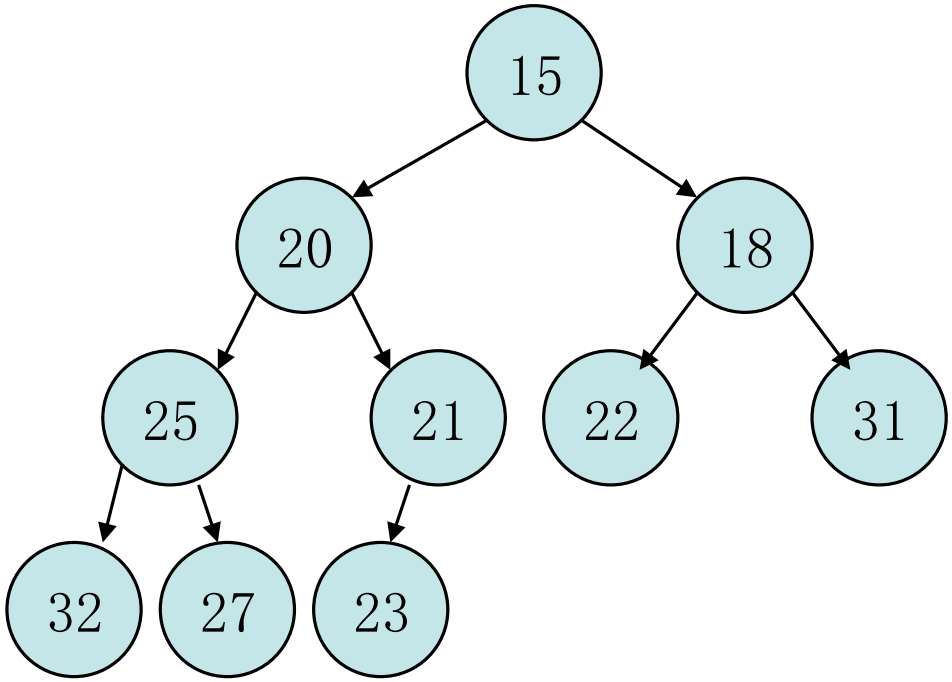
- 我们需要O(1)插入和O(log n)删除操作
- 两个属性:一个完整的二叉树和堆顺序
- 完整的二叉树:一棵满树，唯一的例外在底部
- 堆顺序:最小元素总是在根上
- 所有操作不应违反上述属性!

插入

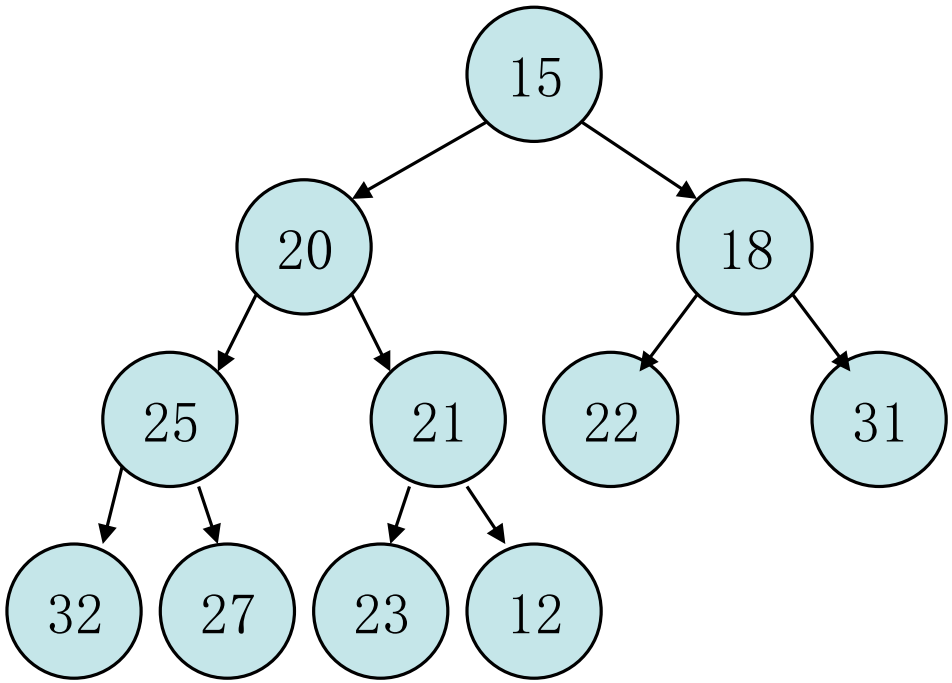
- 保持一棵树完整：
  - 在根部创建一个新节点
  - 将新元素的值赋给该节点
- 但是堆顺序可能已经被破坏了
- 补救方法：如果后面的元素比较小，则与父元素反复交换插入的元素
- 一个新元素将被推到子树的顶部，且它是最小值

上滤

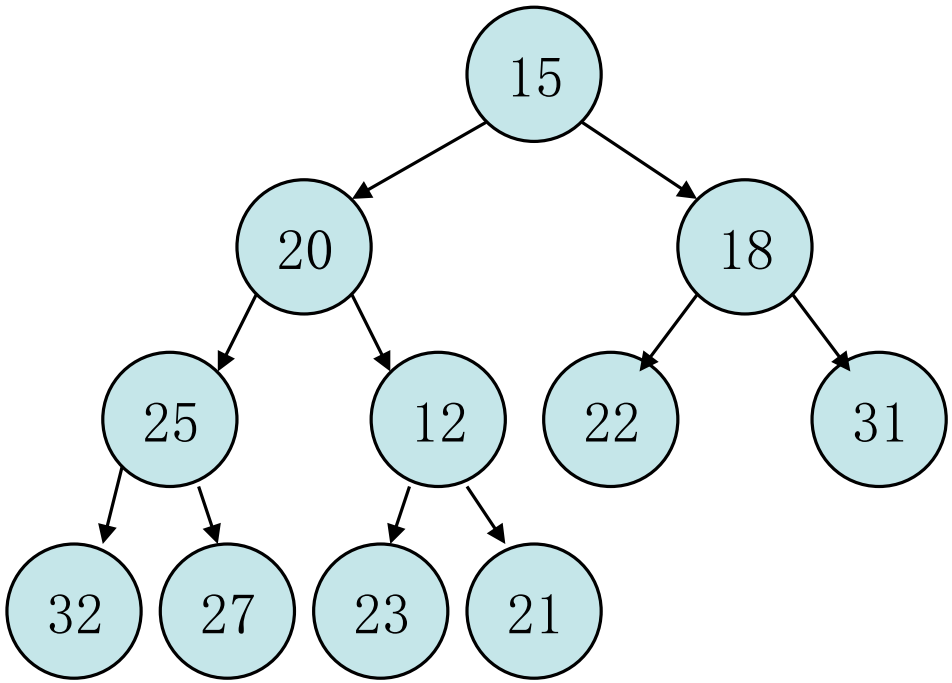
在堆中插入12



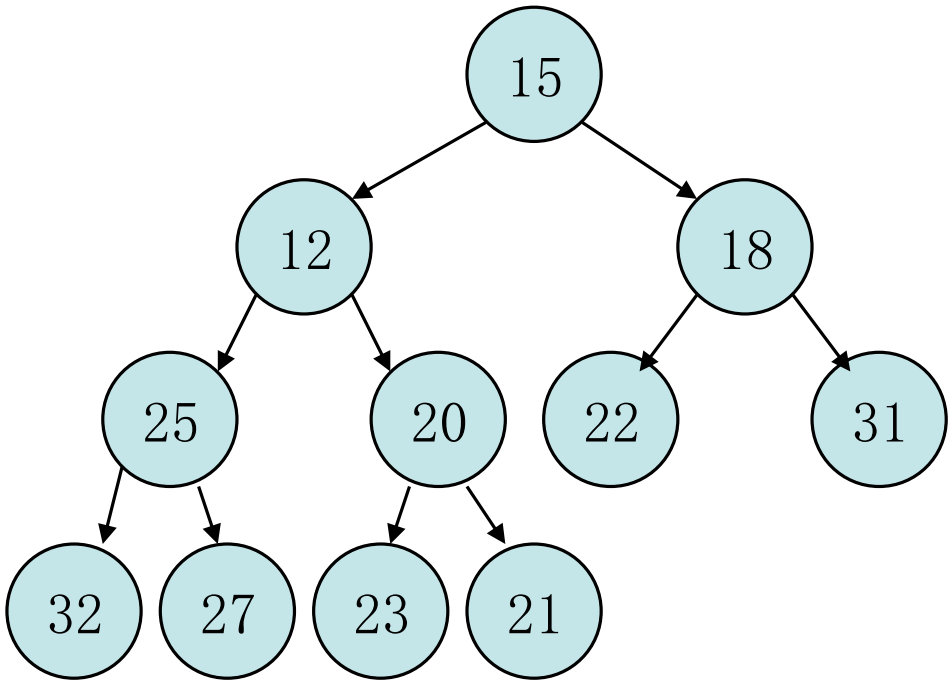
上滤



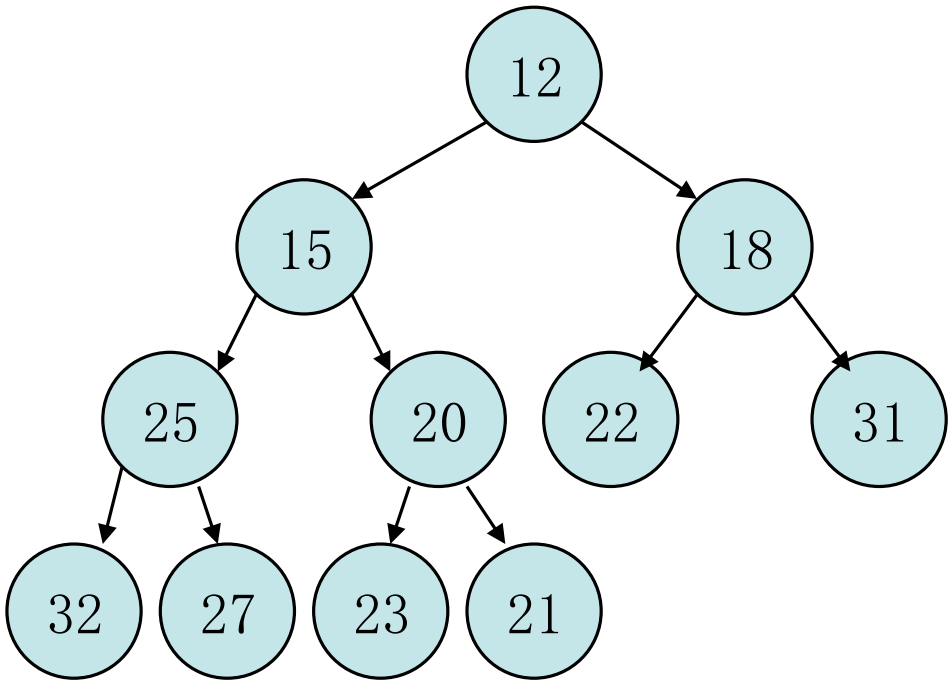
上滤



上滤



上滤

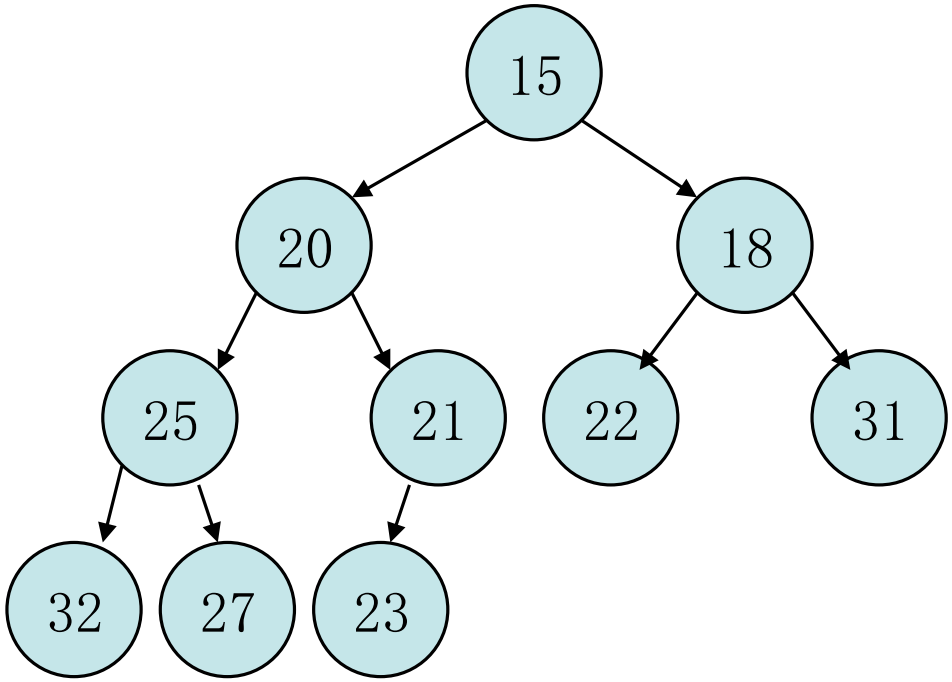


删除最小元

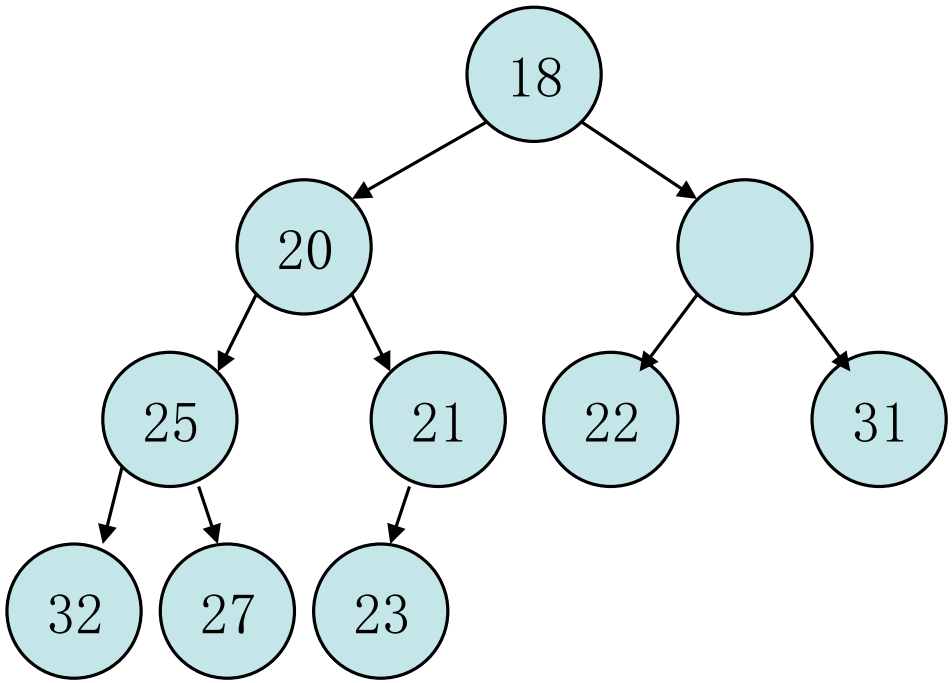
- 记住最小值永远在根节点上
- 但是删除根会生成两个子树：另一个节点必须要移动
- 哪一个？两个孩子中较小的一个
- 重复操作直到到达底部。
- 如果需要填充删除后出现的空穴，用底部最右边的节点。这样可以使树保持完整。

下滤

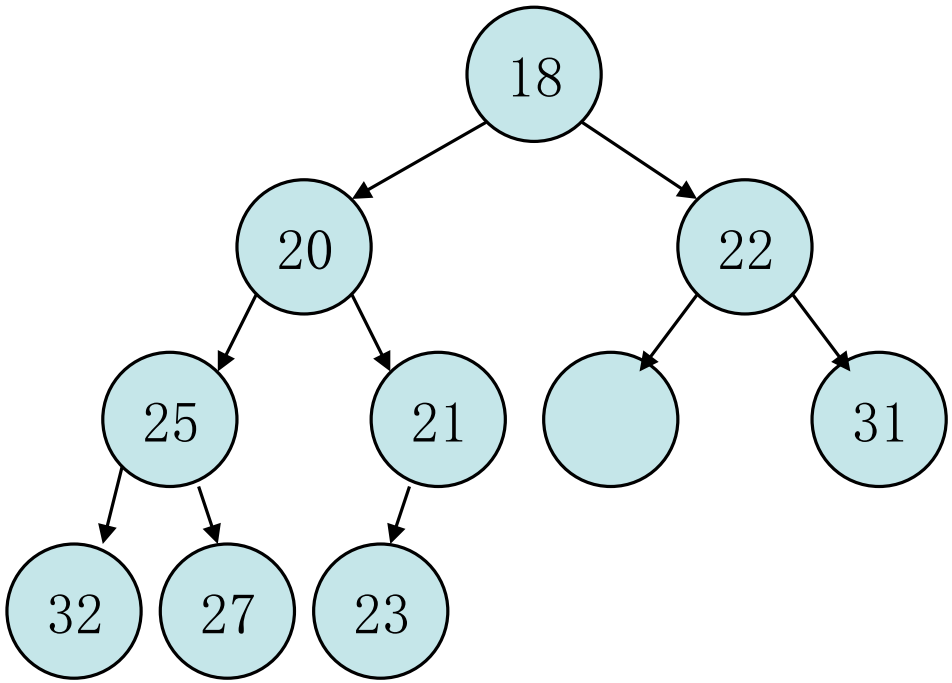
在堆中删除15



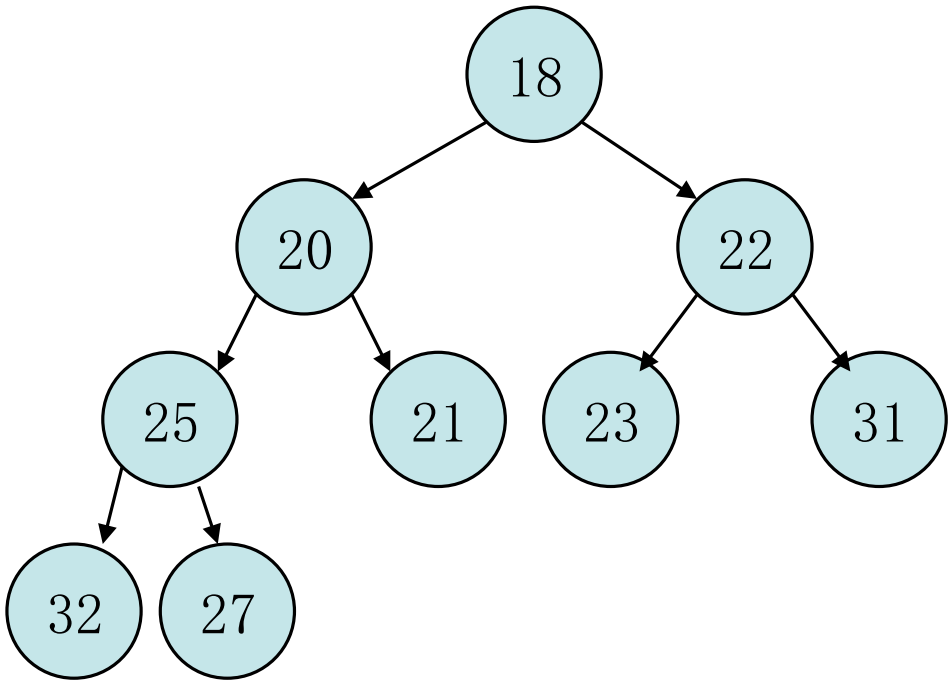
下滤



下滤



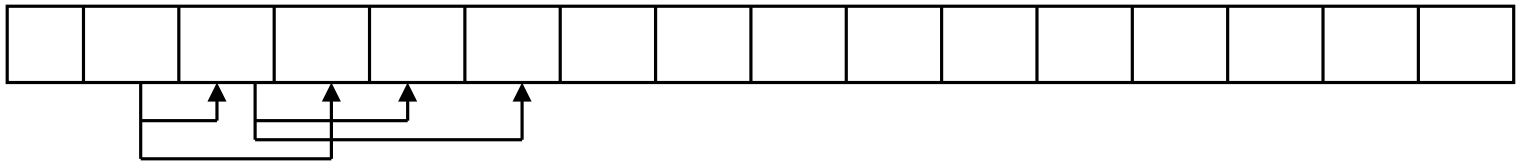
下滤



复杂度

- 最坏情况：
  - 插入，$O(\log\ n)$
  - 删除最小元，$O(\log\ n)$
- 平均情况
  - 插入，$O(1)$，实际上是 2.607 次比较，1.607 次移动
  - 删除最小元，$O(\log\ n)$
- 最大值堆可以类似定义

数组实现

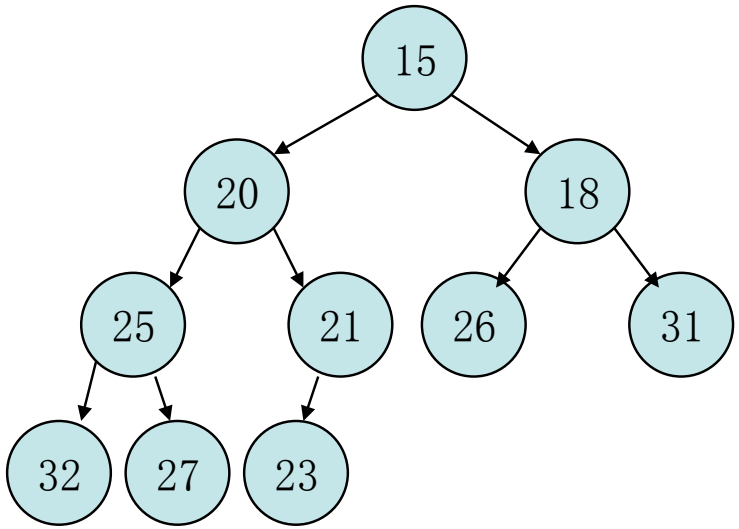


父到子的链接已经默认设定好了，由数组中的索引决定， 左孩子2i，右孩子2i+1， 父在i/2，以此排列下去

```
typedef struct hEAP
{
    int capacity;
    int size;
    int *data;
} HEAP;
```

举例

	15	20	18	25	21	26	31	32	27	23					



Insert代码

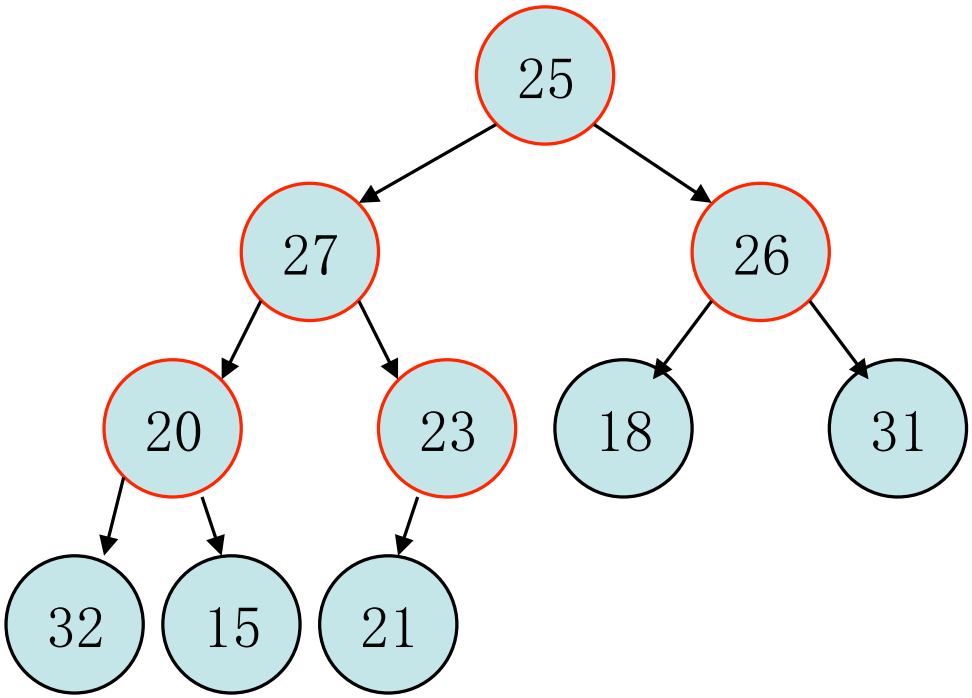
```
void insert (HEAP *h, int x)
{
    int i;

    if (h->size == h->capacity)     /* heap full */
        return;

    /* last element: h->elements[h->size - 1]
       parent of element i: i/2 */
    for (i = ++h->size; i > 1
         && h->elements[i/2] > x; i /= 2)
        h->elements[i] = h->elements[i/2];

    h->elements[i] = x;
}
```

建堆

- 最大元素在哪里？在底部，也许所有叶子都是最大元素。
- 如何构建给定n个元素的堆？
- 每次插入一个元素，或
- 从元素位置任意设置的完整二叉树开始
- 然后调整内部节点的位置以满足堆的特性
- 遵循自下而上的方式直到根节点被调整

Build a Heap

	25	27	26	20	23	18	31	32	15	21



Build a Heap

	25	27	26	20	23	18	31	32	15	21

Build a Heap

	25	27	26	20	23	18	31	32	15	21

	25	27	26	20	21	18	31	32	15	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21
	25	27	26	20	21	18	31	32	15	23
	25	27	26	15	21	18	31	32	20	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21
	25	27	26	20	21	18	31	32	15	23
	25	27	26	15	21	18	31	32	20	23
	25	27	18	15	21	26	31	32	20	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21
	25	27	26	20	21	18	31	32	15	23
	25	27	26	15	21	18	31	32	20	23
	25	27	18	15	21	26	31	32	20	23
	25	15	18	20	21	26	31	32	27	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21
	25	27	26	20	21	18	31	32	15	23
	25	27	26	15	21	18	31	32	20	23
	25	27	18	15	21	26	31	32	20	23
	25	15	18	20	21	26	31	32	27	23
	15	20	18	25	21	26	31	32	27	23

Percolate Down代码

```
void percolate_down (HEAP *h, int i)
{
    int j, tmp, k;

    for (j = i, tmp = h->elements[j]; j * 2 < h->size; j = k)
    {
        /* find the smaller child */
        k = h->elements[2*j] < h->elements[2*j + 1] ? 2*j : 2*j + 1;
        /* if the root is bigger, move child one layer up */
        if (tmp > h->elements[k])
            h->elements[j] = h->elements[k];
        else
            break;
    }
    h->elements[j] = tmp;
}
```

建堆

- N: 堆中的键的个数，用数组实现
- 复杂度: O(N)，N次插入，记住插入的平均复杂度是O(1)
- 最坏情况: 每次插入需要 # 次移动 = 被插入节点的高度，所有 # 次移动总和为 O(N)

```
for (i = h->size; i > 0; i--)
{
    percolate_down (i);
}
```

建堆的复杂度

Theorem

构建堆的复杂度是 O(n)

证明想法：2次比较内部节点，1次找到一个更小的孩子，1与那个孩子比较。下滤操作的移动总数=节点的高度。然后我们需要知道堆中所有内部节点的高度之和。这个值在高度为h-1和完全二叉树和高度为h的完全二叉树的节点的个数之间。

复杂度

$$S = \sum_{i=0}^{h-1} 2^i * (h - i) = h + 2(h-1) + ... + 2^{h-1}$$

$$2S = 2h + 2 * 2(h-1) + ... + 2 * 2^{h-1}$$

-S+2S, we have

$$S = -h + 1 + 2 + ... + 2^h = (2^{h+1} - 1) - h = n - h$$

堆排序

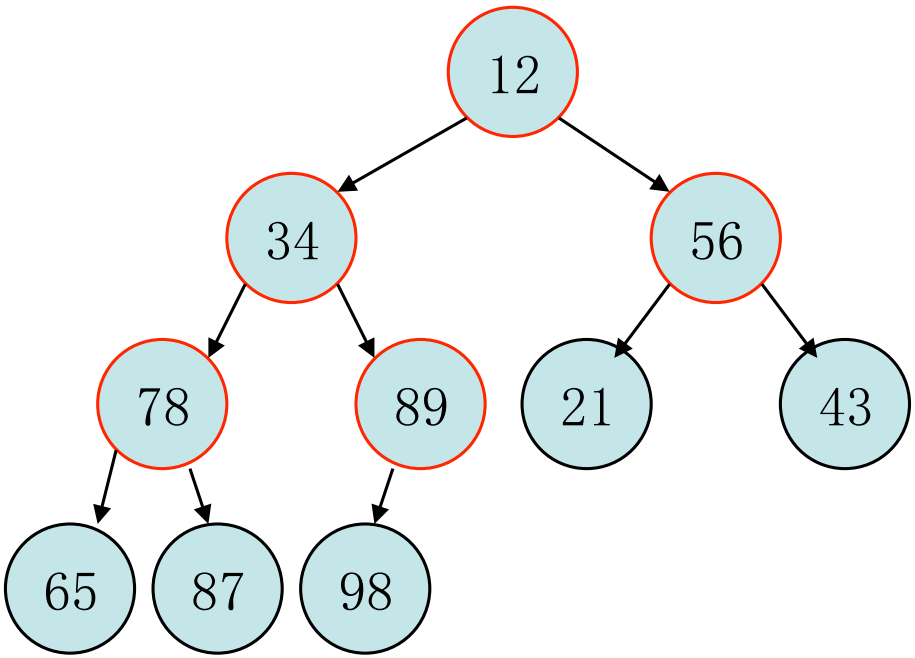
- 总是删除/返回堆的根节点，并将结果复制到一个新数组中
- 数组通过删除最小元方法排序好
- 浪费空间，需要额外数组
- 因为删除最小元方法使堆变小并重新排序。所以我们得到了一个递减的顺序
- Max heap（最大堆） + deleteMax（删除最大元）
- 复杂度 = 构建堆 + 删除最大元n次，O(n log n)

堆排序

- 给定数组：12，34，56，78，89，21，43，65，87，98
- 首先构建一个最大堆
- 然后反复使用删除最大元方法，把被删除的元素到堆的最后一个单元格

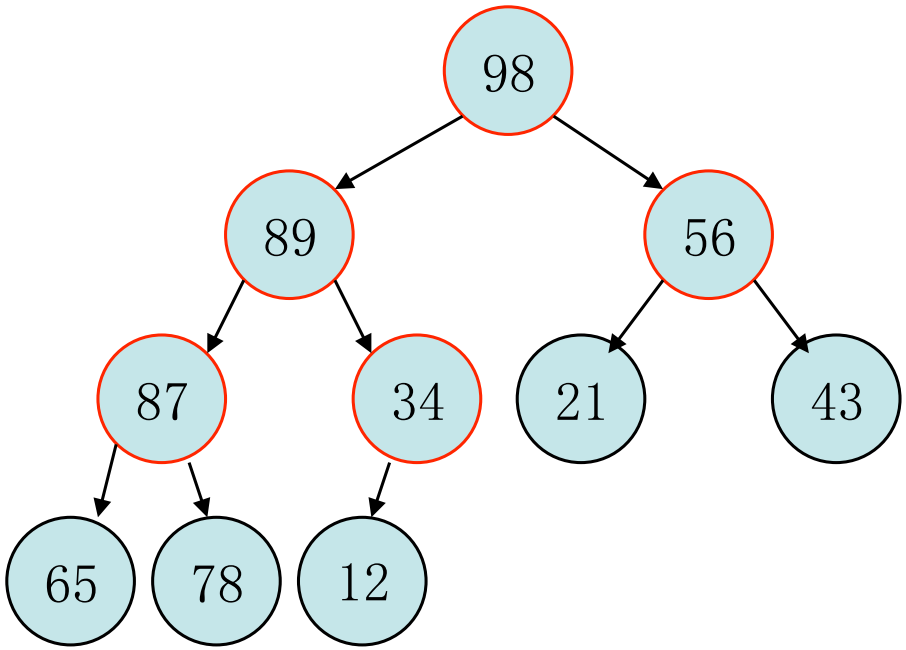
Original Array

	12	34	56	78	89	21	43	65	87	98



Max Heap

	98	89	56	87	34	21	43	65	78	12



	89	87	56	78	34	21	43	65	12	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12

Heap Sort

	98	89	56	87	34	21	43	65	78	12

	89	87	56	78	34	21	43	65	12	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98
	21	12	34	43	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98
	21	12	34	43	56	65	78	87	89	98
	12	21	34	43	56	65	78	87	89	98

小结

- 二叉堆：优先队列，非线性结构
  - 结构：完全二叉树，可以用数组实现
  - 堆序：任意子树，根结点为最小（大）值
- 快速操作：insert O(1)，deleteMin(Max)，O(logn)
- 建堆：对所有非树叶节点依次进行下滤，O(n)
- 堆排序：先构建最大堆，再进行多次deleteMax，替代最后一个树叶

实验7

- 1, 使用分离链路法处理冲突
  - Hash表的大小为2^k-1，初始可以为15
  - 表中的装填因子达到3/4时，增加表的大小至$2^{k+1}-1$，完成再哈希
  - 实现插入，删除和查找操作
  - 设计两个针对可变长度字符串的hash函数，并设计数据评价其性能
- 2, 书上5-7，多项式乘法的改进