

Algorithm - P53

059 - Selection

建堆复杂度

首先可以简单地估算 BUILD-MAX-HEAP 的复杂度

($O(n \lg n)$) 注意到其中每一次 MAX-HEAPFY 的时间复杂度为 $O(c \lg n)$

而 BUILD-MAX-HEAP 中有 $O(n)$ 次 MAX-HEAPFY 的调用

于是 BUILD-MAX-HEAP 的时间复杂度为 $O(n \lg n)$

($O(n \lg n)$) 注意这个上界不是渐近紧确的

($O(n \lg n)$) 具体分析

注意到 MAX-HEAPFY 调用的复杂度与调用结点的高度相关

而由于堆是近乎完全的二叉树

堆中大部分的结点高度都较小

则对于有 n 个结点的堆，其高度为 $\lceil \lg n \rceil$

而堆中高度为 h 的结点最多有 $\lceil n/2^{h+1} \rceil$ 个

另外对高度为 h 的结点，调用 MAX-HEAPFY 的复杂度为 $O(h)$

于是 BUILD-MAX-HEAP 的总时间复杂度

$$T(n) = \sum_{h=0}^{\lceil \lg n \rceil} \lceil n/2^{h+1} \rceil O(h)$$

$$= \sum_{h=0}^{\lceil \lg n \rceil} O(n/2^h) \cdot O(h)$$

$$= n \cdot O\left(\sum_{h=0}^{\lceil \lg n \rceil} h/2^h\right) \text{ 又 } \sum_{h=0}^{\lceil \lg n \rceil} h/2^h < \sum_{h=0}^{\infty} h/2^h$$

$$= n \cdot O\left(\sum_{h=0}^{\infty} h/2^h\right) \text{ 又 } \sum_{h=0}^{\infty} h/2^h = \frac{1}{2} \cdot \sum_{h=1}^{\infty} h/2^{h-1} = \frac{1/2}{(1-1/2)^2} = 2$$

$$= O(n)$$

从上面的推导可知，对于给定 n 个元素的无序数组，可以在线性时间内构造一个最大堆

堆排序 对于包含 $n = A.length$ 个元素的数组 $A[1..n]$ ，通过 BUILD-MAX-HEAP 构造为最大堆

则利用 HEAPSORT 将 $A[1..n]$ 由小到大排序

HEAPSORT(A):

BUILD-MAX-HEAP(A) 构造最大堆

for $i := A.length$ down to 2 :

swap $A[1]$ with $A[i]$

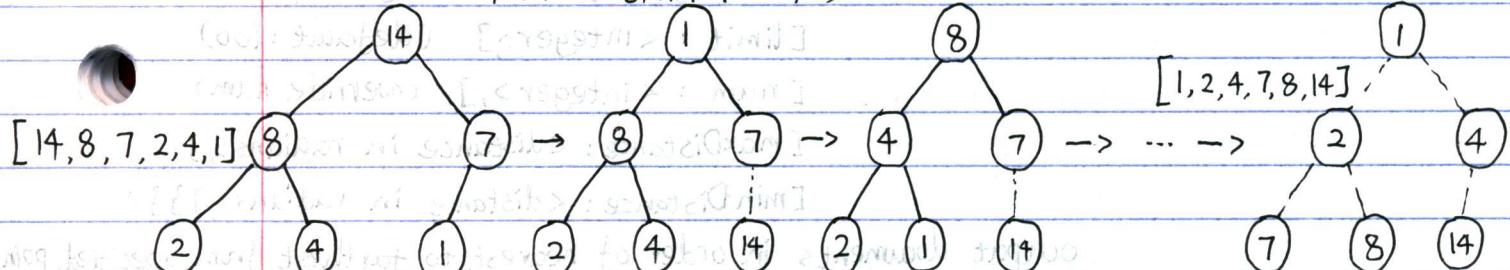
交换 $A[1]$ 与当前堆的最后一个元素

$A.heap-size := A.heap-size - 1$

从堆中删除最后一个元素

MAX-HEAPFY(A, 1)

在 $A[1]$ 上调用 MAX-HEAPFY



Algorithm - P54

堆排序

循环不变量：在 $i := A.length \text{ downto } 2$ 的每次循环迭代开始前

子数组 $A[1..i]$ 是包含数组 $A[1..n]$ 中前 i 个最小元素的最大堆

而子数组 $A[i+1..n]$ 是包含数组 $A[1..n]$ 中 $n-i$ 个最大元素的已排序数组

初始化：在第一次迭代开始前，有 $i = n$

则在执行 BUILD-MAX-HEAP(A)之后

子数组 $A[1..i]$ 即数组 $A[1..n]$ ，包含 $A[1..n]$ 的所有元素

于是子数组 $A[1..i]$ 是包含 $A[1..n]$ 前 i 个最小元素的最大堆平凡地为真

而子数组 $A[i+1..n]$ 为空数组

于是子数组 $A[i+1..n]$ 是包含 $A[1..n]$ 中 $n-i$ 个最大元素的已排序数组平凡地为真

保持：假设在第 i 次迭代前，循环不变量为真，则考虑第 $i+1$ 次迭代后，令迭代计数器为 $i+1$

由于子数组 $A[1..i]$ 是包含 $A[1..n]$ 前 i 个最小元素的最大堆

则此时数组 $A[1..n]$ 的第 $i+1$ 大的元素位于 $A[i]$

执行 $\text{swap } A[i] \text{ with } A[i+1]$ 之后

数组 $A[1..n]$ 的第 $i+1$ 小的元素位于 $A[i]$

而此时 $A[1..i-1]$ 包含 $A[1..n]$ 的 $i-1$ 个最小元素

由于子数组 $A[i+1..n]$ 是包含 $A[1..n]$ 中 $n-i$ 个最大元素的已排序数组

又此时 $A[i+1..n]$ 是第 $i+1$ 小的元素，即第 $n-i+1$ 个最大元素

于是子数组 $A[i+1..n]$ 是包含 $A[1..n]$ 中 $n-i+1$ 个最大元素的已排序数组

注意到由于 $A[2..i-1]$ 没有变化，则在子数组中

结点 $A[2], A[3], \dots, A[i-1]$ 都是最大堆的根结点

则对 $A[i]$ 调用 MAX-HEAPIFY(A, i) 之后

结点 $A[i]$ 也成为最大堆的根结点

于是子数组 $A[1..i-1]$ 是包含 $A[1..n]$ $i-1$ 个最小元素的最大堆

于是 $k+1$ 次迭代之前，迭代计数器为 $i-1$

此时子数组 $A[1..i-1]$ 是包含 $A[1..n]$ 中 $i-1$ 个最小元素的最大堆

子数组 $A[i..n]$ 是包含 $A[1..n]$ 中 $n-i+1$ 个最大元素的已排序数组

终止：循环终止时，迭代计数器为 1，则根据循环不变量

子数组 $A[1..1]$ 只包含 $A[1..n]$ 中的最小元素

子数组 $A[2..n]$ 为 $A[1..n]$ 其余元素的已排序数组

于是可以合并 $A[1..1]$ 和 $A[2..n]$

此时 $A[1..n]$ 为一个已排序数组

由此可知 HEAPSORT 算法的正确性

时间复杂度分析

空间复杂度分析

Algorithm - P55

堆排序

对于 n 个不同的元素，HEAPSORT 的最好情形时间复杂度是 $\Omega(n \lg n)$ 的

assume that the heap is full binary tree with $n = 2^k - 1$

there are 2^{k-1} leaves and $2^{k-1} - 1$ inner nodes

假设堆是满二叉树且有 $n = 2^k - 1$ 个结点，则有 2^{k-1} 个叶结点与 $2^{k-1} - 1$ 个内点
look at sorting the first 2^{k-1} elements of the heap

考虑对于前 2^{k-1} 个元素的排序

consider arrangement in the heap

and color the leaves to be red and inner nodes to be blue

考虑这些元素在堆中的分布，将叶结点标记为红色，内点标记为蓝色

colored nodes are a subtree of the heap (otherwise contradiction)

着色的结点为堆的一个子树（否则会产生矛盾）

since there are 2^{k-1} colored nodes, at most 2^{k-2} red, at least 2^{k-2} blue

有 2^{k-1} 个着色的点，于是至多有 2^{k-2} 红点，至少 $2^{k-2} - 1$ 个蓝点

red nodes can jump directly to the root,

红点可以直接跳到根结点，

blue nodes need to travel up before get removed

蓝点在移除前需要在堆中移动

minimal case of number of swaps to move blue nodes to root

考虑将蓝点移动到根结点所需的最少交换次数

there are $2^{k-2} - 1$ blue nodes

and arranged in binary tree

if there are d such blue nodes,

there would be $i = \lg d$ levels, each containing 2^i nodes with length i

如果存在 d 个这样的蓝点，则有 $i = \lg d$ 层，

每一层包含 2^i 个路径长度为 i 的点，

于是交换的次数为

$$\sum_{i=0}^{\lg d} i 2^i = 2 + (\lg d - 2) \cdot 2^{\lg d}$$

$$O(\lg d \cdot 2^{\lg d}) = O(d \lg d)$$

lazy (but cute) trick: = 简单的技巧或巧妙的策略

由于已找到一个排序堆中一半元素的紧约束

a tight bound on sorting half of the heap

于是有递归式 $T(n) = T(n/2) + \Omega(n \lg n)$

则根据主定理，可知 $T(n) = \Omega(n \lg n)$

Algorithm - P56

2.9 - TBM.23

优先队列 (priority queue)，用于维护由一组元素构成的集合 S 的数据结构

关键字 (key)：每个元素关联的一个值

最大优先队列 (max-priority queue) 支持的方法有

$\text{INSERT}(S, x)$ ：将元素 x 插入集合 S ，等价于 $S := S \cup \{x\}$

$\text{MAXIMUM}(S)$ ：返回集合 S 中键值最大的元素

$\text{EXTRACT-MAX}(S)$ ：移除并返回集合 S 中键值最大的元素

$\text{INCREASE-KEY}(S, x, k)$ ：将集合 S 中元素 x 的键值提高到 k

假设参数 k 不小于元素 x 的键值

相应地有最小优先队列 (min-priority queue)：

$\text{INSERT}(S, x)$ ：将元素 x 插入集合 S ，等价于 $S := S \cup \{x\}$

$\text{MINIMUM}(S)$ ：返回集合 S 中键值最小的元素

$\text{EXTRACT-MIN}(S)$ ：移除并返回集合 S 中键值最小的元素

$\text{DECREASE-KEY}(S, x, k)$ ：将集合 S 中元素 x 的键值降低到 k

假设参数 k 不大于元素 x 的键值

最大优先队列可用于共享计算机系统的作业调度 (scheduling job on shared computer)

最小优先队列可用于基于事件驱动的模拟器 (event-driven simulator)

优先队列可以用堆 (heap) 实现，应用中的对象 对应于一个给定的优先队列元素

句柄 (handle)：指针 / 整型数，准确含义依赖于具体的应用程序

用堆实现优先队列时，需要在堆的每个元素中存储对应对象的句柄

相对地在应用程序对象中存储堆中对应元素的句柄

通常是存储堆的数组的数组下标

注意在堆的操作过程中，元素会改变在数组中的位置，于是下标会改变

具体实现中，重新确定堆元素位置时，更新相应的应用程序对象的句柄

注意对应用程序对象的访问细节强烈依赖于应用程序实现方式

句柄应当被正确地维护

MAXIMUM

对于以最大堆 A 实现的最大优先队列

$\text{HEAP-MAXIMUM}(A)$

return $A[1]$

由于 A 是维护好的最大堆，于是第一个元素 (根结点) 即最大元素

可以在 $O(1)$ 的时间内实现 MAXIMUM 操作

Algorithm - P57

strong

EP19 - 从堆中取出最大值

EXTRACT-MAX，对于以最大堆A实现的最大优先队列

HEAP-EXTRACT-MAX(A) :

if $A.\text{heap-size} < 1$: [当最大堆为空时，抛出错误]
error "heap underflow"

max := $A[1]$

$A[1] := A[A.\text{heap-size}]$] 从最大堆A中移除最大元素

$A.\text{heap-size} := A.\text{heap-size} - 1$] 并进行堆维护

MAX-HEAPIFY(A, 1)

return max

由于其中只有 MAX-HEAPIFY(A, 1) 的复杂度是 $O(\lg n)$ ，其他语句均为 $O(1)$

所以 HEAP-EXTRACT-MAX 在 $O(\lg n)$ 的时间复杂度内实现 EXTRACT-MAX

对于以最大堆A实现的最大优先队列，

假定 ~~提高~~ 一个元素 $A[i]$ ，使得 $A[i] > A[\text{PARENT}(i)]$ ，而其他元素
则进行最大堆A的维护，即 ~~堆的性质~~ 均符合最大堆性质

MAX-PASSBALL(A, i) :

while $i > 1$ and $A[i] > A[\text{PARENT}(i)]$] 与 $A[i]$ 的深度相关
swap $A[i]$ with $A[\text{PARENT}(i)]$] 时间复杂度为 $O(\lg n)$

$i := \text{PARENT}(i)$

某次迭代：数组 $A[1..A.\text{heap-size}]$

循环不变量：在 while 循环开始前，或者 ~~已满足最大堆性质~~

或者 $A[i] > A[\text{PARENT}(i)]$ 是唯一一对不满足最大堆性质的元素

初始化 ~~满足最大堆性质~~：由于对于满足最大堆性质的数组 $A[1..A.\text{heap-size}]$ 元素

或者 提高 $A[i]$ ， $i \in [1..A.\text{heap-size}]$ 的值，或者在 $A[A.\text{heap-size}+1]$ 插入新

则此时 $A[i] > A[\text{PARENT}(i)]$ 是唯一一对不满足最大堆性质的元素

维持：假设在 while 循环的某次迭代开始前，仅有 $A[i] > A[\text{PARENT}(i)]$ 不满足最大堆性质

在该次迭代中，交换 $A[i]$ 与 $A[\text{PARENT}(i)]$ 的值，~~满足最大堆性质~~

由于 $A[\text{PARENT}(i)]$ 大于 $A[i]$ 原本的值

则交换后以 $A[i]$ 为根的子树满足最大堆性质

使 $i := \text{PARENT}(i)$ 后， $A[i]$ 的 ~~深度~~ 减少 1

终止：while 循环的终止条件有两种

或者 $i = 1$ ，即 $A[i]$ 为数组A的根，于是A满足最大堆性质

或者 $A[i] \leq A[\text{PARENT}(i)]$ ，即以 $A[\text{PARENT}(i)]$ 为根的子树满足最大堆性质

于是此时数组A满足最大堆性质

Algorithm - P58

MAX-PASSBALL

MAX-HEAP-INCREASE-KEY

注意到在 MAX-PASSBALL 中，交换元素通常需要 3 次赋值

可以参考 INSERTION-SORT 中内循环部分，非递归版本

使得在 while 循环的每一次迭代中仅使用一次赋值

MAX-PASSBALL(A, i)

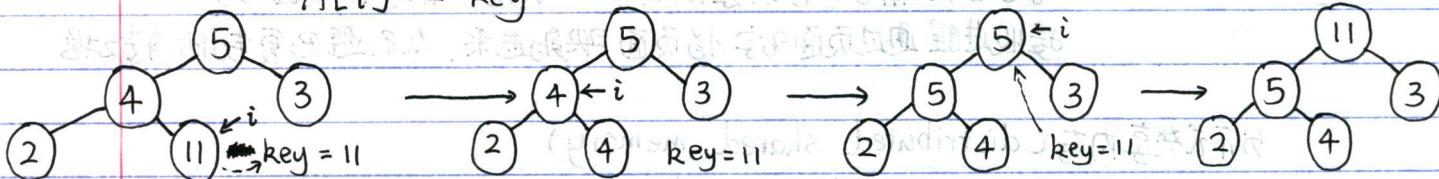
let key := A[i]

while i > 1 and key > A[PARENT(i)] 与 A[i] 的深度相关

 A[i] := A[PARENT(i)] 时间复杂度为 $O(\lg n)$

 i := PARENT(i)

 A[i] := key



INCREASE-KEY

对于以最大堆 A 实现的最大优先队列

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

 if key < A[i] 由于堆中所有结点的键值都大于等于其父结点的键值

 error "new key is smaller than current key"

 A[i] := key

 MAX-PASSBALL(A, i)

由于其中只有 MAX-PASSBALL(A, i) 的复杂度为 $O(\lg n)$ ，其他语句均为 $\Theta(1)$

所以 HEAP-INCREASE-KEY 在 $O(\lg n)$ 的时间复杂度内实现 INCREASE-KEY

INSERT

对于以最大堆 A 实现的最大优先队列

 INSERT 操作的输入为插入最大堆 A 的新元素的键值

 MAX-HEAP-INSERT(A, key)

 A.heap-size := A.heap-size + 1

 A[A.heap-size] := -∞

 HEAP-INCREASE-KEY(A, A.heap-size, key)

注意到先对 A[A.heap-size] 赋值为 $-∞$ 保证了插入操作的正确性

以保证之后的 HEAP-INCREASE-KEY 操作不会出错

由于其中只有 HEAP-INCREASE-KEY 的复杂度为 $O(\lg n)$ ，其他语句均为 $\Theta(1)$

所以 MAX-HEAP-INSERT 在 $O(\lg n)$ 的时间复杂度内实现 INSERT

Algorithm - P59

STUDENT

INSERT

可以不通过 HEAP-INCREASE-KEY 而直接使用 MAX-PASSBALL 实现

MAX-HEAP-INSERT(A, key)

A.heap-size := A.heap-size + 1

A[A.heap-size] := key

MAX-PASSBALL(A, A.heap-size)

是依旧可以在 $O(\lg n)$ 的时间复杂度内实现 INSERT 操作

UPDATE-KEY

对于以最大堆 A 实现的最大优先队列

考虑 将元素 A[i] 的关键字更新为新值 key

而相比于 INCREASE-KEY 只支持增大 A[i] 的关键字

UPDATE-KEY 也支持减少 A[i] 关键字的情况

HEAP-UPDATE-KEY(A, i, key)

if key > A[i]

A[i] := key

MAX-PASSBALL(A, i)

if key < A[i]

A[i] := key

MAX-HEAPFY(A, i)

由于其中只有 MAX-PASSBALL 和 MAX-HEAPFY 的时间复杂度为 $O(\lg n)$,

且其中 最多执行一个, 而其他操作均为 $O(1)$

于是 HEAP-UPDATE-KEY 在时间复杂度 $O(\lg n)$ 实现 UPDATE-KEY

HEAP-DELETE

对于以最大堆 A 实现的最大优先队列

考虑从堆 A 中删除结点 A[i] 的操作, 并且返回删除结点的关键字

HEAP-DELETE(A, i) • ret := A[i]

if i == A.heap-size

] 如果 A[i] 是最后一个元素

A.heap-size := A.heap-size - 1

] 则 不必执行 MAX-HEAPFY

else

A[i] := A[A.heap-size]

类似于 HEAP-EXTRACT-MAX

A.heap-size := A.heap-size - 1

中的操作

MAX-HEAPFY(A, i)

时间复杂度为 $O(\lg n)$

return ret

于是在时间复杂度 $O(\lg n)$ 实现 HEAP-DELETE

Algorithm - P60

left-heap



插入方法建堆

对于大小为 $n = A.length$ 的数组 $A[1..n]$ 转换为最大堆

除了通过调用 MAX-HEAPIFY

也可以通过反复调用 MAX-HEAP-INSERT 实现向堆中插入元素

BUILD-MAX-HEAP'(A)

$A.heap-size := 1$
for $i := 2$ to $A.length$

MAX-HEAP-INSERT(A, A[i])

也可以通过直接调用 MAX-PASSBALL 实现

BUILD-MAX-HEAP'(A)

for $i := 2$ to $A.length$

MAX-PASSBALL(A, i)

$A.heap-size := A.length$

由于在两种实现中

只有 MAX-HEAP-INSERT 和 MAX-PASSBALL 的时间复杂度为 $O(c \lg n)$

于是 BUILD-MAX-HEAP 在时间复杂度 $O(c n \lg n)$ 内实现建堆

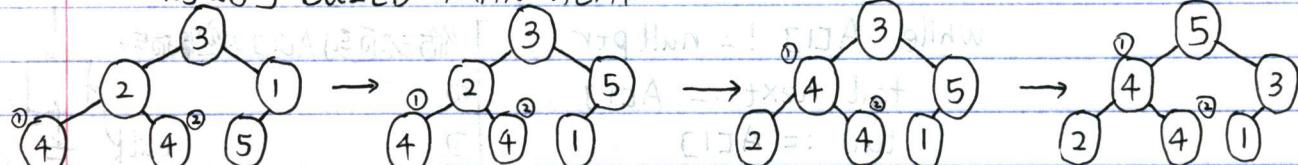
特别地有，当输入为已排序的升序数组，则每个插入元素都会移动到根结点

于是 BUILD-MAX-HEAP' 的时间复杂度为 $O(c n \lg n)$

BUILD-MAX-HEAP/BUILD-MAX-HEAP' 的堆排序不是稳定的 (stable)

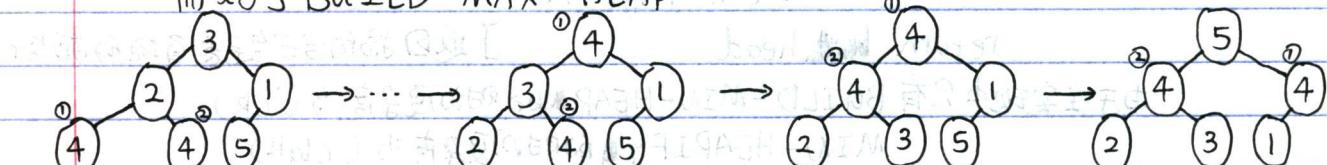
如对于输入数组 $[3, 2, 1, 4^0, 4^0, 5]$

则对于 BUILD-MAX-HEAP



于是从数组上看为 $[5, 4^0, 3, 2, 4^0, 1]$

而对于 BUILD-MAX-HEAP'



于是从数组上看为 $[5, 4^0, 4^0, 2, 3, 1]$

则调用 HEAP-SORT 分别生成排序 $[1, 2, 3, 4^0, 4^0, 5]$

以及 $[1, 2, 3, 4^0, 4^0, 5]$

Algorithm - P61

有序链表合并

对于 k 个有序链表，可使用最小堆实现合并为一个有序链表

假设 k 个有序链表总共包含 n 个元素

首先将 k 个有序链表构建为一个包含 k 个元素的最小堆 A

其中每一项为指向一个有序链表第一项的指针

则返回指定项的关键字的函数为 $\text{KEY}(A, i)$

$\text{KEY}(A, i)$

if $A[i] == \text{nullptr}$

return $-\infty$

else

return $A[i].value$

KEY 的时间复杂度为 $O(1)$

于是每一项的关键字为链表首项保存的值，如果链表为空则返回 $-\infty$

并且所有堆操作中的比较均使用这个 key 值

则实现的合并算法为

对于包含 k 个有序链表的数组 $A[1..k]$ ，其中每一项为指向一个有序链表首项的指针

$\text{MERGE-LINKLIST}(A)$

$\text{BUILD-MIN-HEAP}(A)$] 将数组 A 转换为最小堆

$A.\text{heap-size} := A.length$

$\text{head} := A[1]$

$\text{tail} := A[1]$

$A[1] := A[1].next$

$\text{MIN-HEAPIFY}(A, 1)$

while $A[1] \neq \text{nullptr}$

$\text{tail}.next := A[1]$

$\text{tail} := A[1]$

$A[1] := A[1].next$

$\text{MIN-HEAPIFY}(A, 1)$

return head

初始化返回的链表

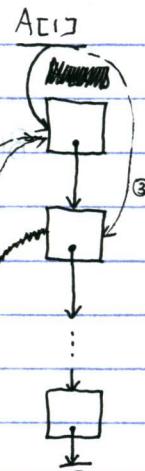
head to tail

return linklist

$\text{head} \rightarrow \boxed{\quad}$

\downarrow

$\text{tail} \leftarrow \boxed{\quad}$



由于在实现中只有 BUILD-MIN-HEAP 的时间复杂度为 $O(k)$

MIN-HEAPIFY 的时间复杂度为 $O(\lg k)$

而且 while 循环需要 $n-1$ 次迭代，于是共有 n 次 MIN-HEAPIFY

则 MERGE-LINKLIST 在时间复杂度 $O(n \lg k)$ 内实现有序链表合并

其中 k 为有序链表个数， n 为有序链表中的总结点数

Algorithm - P62

HEAPIFY - 完全d叉堆的构建

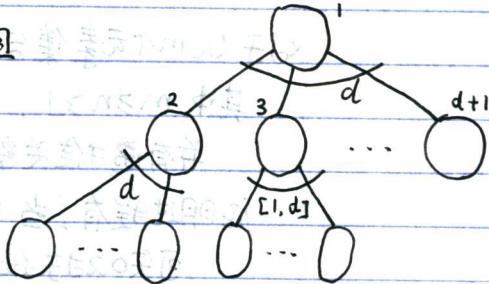
d叉堆

与二叉堆类似，指一个可以看作近似完全d叉树的数组

d叉树中的每个结点，对应数组中的一个元素

除最底层外，d叉树是完全填满的，

且每一层均为从左向右填充



与二叉堆类似，首先需要实现对给定的结点下标，计算父结点和第k个子结点。

PARENT(d, i)

return $\lfloor \frac{i-1}{d} \rfloor + 1$

CHILD(d, i, k)

return $d * \lfloor \frac{i-1}{d} \rfloor + k + 1$

于是有，d叉堆的内点的最大编号为 $\lfloor \frac{n-1}{d} \rfloor + 1$

叶结点的编号为 $\lfloor \frac{n-1}{d} \rfloor + 2, \dots, n$

d叉堆的高度是 $\Theta(d \log n)$ 的，即 $\Theta(\lg^n / \lg d)$ 的

与二叉堆类似，实现d叉堆的堆维护，其中 $d > 2$ 且 d叉堆为最大堆

MAX-HEAPIFY-LOOP-D(A, d, i)

while $i \leq \lfloor \frac{A.heap-size-1}{d} \rfloor + 1$

 largest := i

 for $k := \text{CHILD}(i, 1)$ to $\text{CHILD}(i, d)$

 if $k \leq A.heap-size$ and $A[k] > A[\text{largest}]$:

 largest := k

 if $\text{largest} \neq i$:

 swap $A[i]$ with $A[\text{largest}]$

$i := \text{largest}$

 else:

 break

由于 for 循环的迭代次数为 $\Theta(d)$ ，while 循环的迭代次数为 $\Theta(\lg^n / \lg d)$

于是 MAX-HEAPIFY-LOOP-D 在时间复杂度 $\Theta(d \lg^n / \lg d)$ 内实现 MAX-HEAPIFY

MAX-PASSBALL(A, d, i)

key := $A[i]$

while $i > 1$ and $\text{key} > A[\text{PARENT}(d, i)]$

$A[i] := A[\text{PARENT}(d, i)]$

$i := \text{PARENT}(d, i)$

$A[i] := \text{key}$

于是时间复杂度为 $O(d \lg^n / \lg d)$

Algorithm - P63

d叉堆

对于以d叉堆实现的最大堆，其中 $d > 2$

已在时间复杂度 $\Theta(d \lg n / \lg d)$ 内实现 d叉堆的堆维护 MAX-HEAPIFY

已在时间复杂度 $O(d \lg n / \lg d)$ 内实现 MAX-PASSBALL

于是 d叉堆也可以以相同的形式实现二叉堆优先队列的操作

在时间复杂度 $O(d \lg n / \lg d)$ 内实现 INSERT

在时间复杂度 $\Theta(1)$ 内实现 MAXIMUM

在时间复杂度 $\Theta(d \lg n / \lg d)$ 内实现 EXTRACT-MAX

在时间复杂度 $O(d \lg n / \lg d)$ 内实现 UPDATE-KEY

在时间复杂度 $O(d \lg n / \lg d)$ 内实现 HEAP-DELETE

2	3	8	∞
4	9	12	∞
5	14	∞	∞
16	∞	∞	∞

Young氏矩阵 (Young tableau)，对于 $m \times n$ 矩阵

其中每一行都是从左到右排序，每一列都是从上到下排序

即对于矩阵中的元素 $A[i, j]$ ，都有 $A[i, j] \leq A[i+1, j]$ 且 $A[i, j] \leq A[i, j+1]$

矩阵中存在值为 ∞ 的数据项，用于表示不存在的元素

特别地当类似于最小堆时取 $+\infty$ ，而当类似于最大堆时取 $-\infty$

于是 Young 氏矩阵可以用于存储 $r \leq mn$ 个数

对于 Young 氏矩阵，可知 $A[1, 1]$ 是矩阵中的最小元素

而 $A[m, n]$ 是矩阵的最大元素

于是当 $A[1, 1] = \infty$ 时，则其中任意元素 $A[i, j] \geq A[1, 1] = \infty$

于是当 $A[1, 1] = -\infty$ 时，则其中任意元素 $A[i, j] \leq A[1, 1] = -\infty$

当 $A[m, n] < \infty$ 时，则其中任意元素 $A[i, j] \leq A[m, n] < \infty$

于是 $A[1, 1] < \infty$ ，即 Young 氏矩阵为满

与堆维护 MAX-HEAPIFY 类似地实现向右下角的 SINK

MIN-SINK(Y, i, j)

~~for i := 1 to m do for j := 1 to n do~~

~~while i < m & j < n do~~

~~largest := (i, j)~~

~~if i < m and Y[largest] > Y[i+1, j] then largest := (i+1, j)~~

~~if j < n and Y[largest] > Y[i, j+1] then largest := (i, j+1)~~

~~if largest != (i, j) then swap Y[i, j] with Y[largest]~~

循环迭代次数

是 $O(mn)$

迭代中的所有语句

是 $\Theta(1)$

于是在时间复杂度

$O(mn)$

实现 SINK

~~(i, j) := largest~~

~~break~~

~~swap Y[i, j] with Y[largest]~~

Algorithm - P64

9. Task

Young矩阵 与堆的 MAX-PASS BALL 类似地实现向左上角的 BUBBLE

MIN-BUBBLE(Y, i, j)，其中 Y 为 $m \times n$ 的 Young 矩阵

$m := Y.rows, n := Y.columns$

while ($i, j > 0$ and $Y[i, 1] < Y[i, 2]$) do

~~smallest~~ := (i, j)

 if $i > 1$ and $Y[smallest] < Y[i-1, j]$ then

~~smallest~~ := $(i-1, j)$

 if $j > 1$ and $Y[smallest] < Y[i, j-1]$ then

~~smallest~~ := $(i, j-1)$

 if $smallest \neq (i, j)$ then

 swap $Y[i, j]$ with $Y[smallest]$

$(i, j) := smallest$

 else break

break

与 MIN-SINK 类似，while 循环的迭代次数是 $O(m+n)$ ，且

且循环迭代中的语句时间复杂度均为 $\Theta(1)$ 的

即在时间复杂度 $O(m+n)$ 实现了 MIN-SINK

与优先队列类似地实现 MINIMUM 方法

对于已知且合法的 Young 矩阵 Y ， $m \times n$ 矩阵

YOUNG-MINIMUM(Y)
return $Y[1, 1]$

由于对于 $1 \leq i \leq m, 1 \leq j \leq n$ ， $A[i, j] \leq A[i+1, j] \wedge A[i, j] \leq A[i, j+1]$

于是 $Y[1, 1]$ 即是最小元素，则在 $\Theta(1)$ 时间复杂度内实现 MINIMUM

与优先队列的 EXTRACT-MAX 类似地实现 EXTRACT-MIN

YOUNG-EXTRACT-MIN(Y):

min := $Y[1, 1]$

$Y[1, 1] := Y[m, n], Y[m, n] := \infty$

MIN-SINK($Y, 1, 1$)

return min

由于其中只有 MIN-SINK($Y, 1, 1$) 的时间复杂度为 $O(m+n)$ ，其他语句均为 $\Theta(1)$

所以 YOUNG-EXTRACT-MIN 在 $O(m+n)$ 时间复杂度内实现 EXTRACT-MIN

Algorithm - P65

Young代矩阵

YOUNG-UPDATE-KEY(Y, i, j, key)

Young代矩阵与优先队列的UPDATE-KEY方法类似地实现，对于 $m \times n$ 的Young代矩阵

YOUNG-UPDATE-KEY(Y, i, j, key)

如果 $Y[i, j] < key$:

$Y[i, j] := key$

 MIN-SINK(Y, i, j)

如果 $Y[i, j] > key$:

$Y[i, j] := key$

 MIN-BUBBLE(Y, i, j)

由于其中只有 MIN-SINK 和 MIN-BUBBLE 的时间复杂度为 $O(m+n)$

且至多只有其中之一被调用一次

于是 YOUNG-UPDATE-KEY 在 $O(m+n)$ 的时间复杂度内实现了 UPDATE-KEY

对于 $m \times n$ 的 Young 代矩阵实现插入一个新元素

YOUNG-INSERT(Y, key)

$m := Y.rows, n := Y.columns$

if $Y[m, n] < \infty$

 error "matrix full"

else

$Y[m, n] := key$

 MIN-BUBBLE(Y, m, n)

由于其中只有 MIN-BUBBLE 的时间复杂度为 $O(m+n)$ 且仅调用一次，其他语句均为 $\Theta(1)$ 的

于是 YOUNG-INSERT 在 $O(m+n)$ 的时间复杂度内实现插入新元素

对于 $m \times n$ 的 Young 代矩阵实现查找一个给定元素是否存在

YOUNG-SEARCH(Y, key)

$m := Y.rows, n := Y.columns, i := 1, j := n + 1$

while TRUE:

 if $Y[i, j] = key$

 return (i, j)

 if $Y[i, j] < key$ and $i < m$

$i := i + 1$

 else if $Y[i, j] > key$ and $i < j$

$j := j - 1$

 return NULL

search (11)

2	3	8	10	17
4	9	12	13	∞
5	(11)	14	∞	∞
15	16	∞	∞	∞

循环中所有语句均为 $\Theta(1)$ ，且其中清

且可以看作从右上角 $[i, n]$

向左下角 $[m, 1]$ 移动

于是最多进行 $n+m-2$ 次迭代

则 YOUNG-SEARCH(Y, key) 在 $O(m+n)$

的时间复杂度内实现搜索元素