中国科学技术大学
University of Science and Technology of China

算法基础
第三讲：基于比较的排序

主　讲:　顾 乃 杰　教授
单　位:　计算机科学技术学院
学　期:　2015-2016 (秋)

創寰宇學府 育天下英才
嚴濟慈 一九八八年五月 題

排序基本概念

- *排序算法的稳定性* 判断标准:

  不管输入数据是如何分布，对任意关键字相同的数据对象，在排序过程中是否能保持相对次序不变。如 2, 2*, 1，排序后若为1, 2*, 2 则该排序方法是不稳定的。

- *内排序与外排序* 区分标准:

  排序过程是否全部在内存中进行

- *排序的时间开销*

  通常用算法执行中的数据比较次数和数据移动次数来衡量。

排序基本概念（续）

- 排序的方法有很多，但简单地判断那一种算法最好，以便能够普遍选用则是困难的。
- 评价排序算法好坏的标准主要有两条：算法执行所需要的时间和所需要的附加空间。另外，算法本身的复杂程度也是需要考虑的一个因素。
- 排序算法所需要的附加空间一般都不大，矛盾并不突出。而排序是一种经常执行的一种运算，往往属于系统的核心部分，因此，排序的时间开销是算法好坏的最重要的标志。

5. 简单排序和 Shell 排序

- 简单排序包括直接插入排序、简单选择排序和冒泡排序等排序算法，他们的最坏情况时间复杂度均是 $O(n^2)$，所需附加空间均是 $\Theta(1)$。
- 直接插入排序和冒泡排序是稳定的排序算法，而简单选择排序是不稳定的。
- Shell 排序利用直接插入排序做为其子过程，Shell 排序也是不稳定的。

5.1 简单选择排序

- 选择排序(Selection Sort)的基本思想是对待排序的记录序列进行 n-1 遍的处理，第 *i* 遍处理是将 $a_i, \ldots, a_n$中最小者与 a_i 交换位置。这样，经过 *i* 遍处理之后，$a_1, a_2, \ldots, a_i$ 有序，前 *i* 个记录的位置已经是正确的了。
- 第 *i* 趟排序: 当前有序区和无序区分别为$a_1, \ldots, a_{i-1}$ 和 $a_i, \ldots, a_n$ $(1 \le i \le n-1)$。该趟排序从当前无序区中选出关键字最小的记录 a_k，将它与无序区的第1个记录 a_i交换，使 $a_1, \ldots, a_i$ 和 $a_{i+1}, \ldots, a_n$ 分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区。

简单选择排序算法描述

简单选择排序的具体算法如下：

```
Selection-sort(A)
1. for i←1 to n-1                //做第i趟排序(1≤i≤n-1)//
2.     do  k←i;
3.     for j←i+1 to n            //在当前无序区A[i..n]中选key最小的记录A[k] //
4.         do if (A[j]<A[k])
5.               then k←j;  //k为目前找到的最小关键字所在位置//
6.     if (k≠i)                                  //交换A[i]和A[k] //
7.         then A[i]↔A[k];
```

简单选择排序算法分析

- 关键字比较次数：

  无论文件初始状态如何，在第 *i* 趟排序中选出最小关键字的记录，需做 *n-i* 次比较，因此，总的比较次数为：$n(n-1)/2= O(n^2)$。

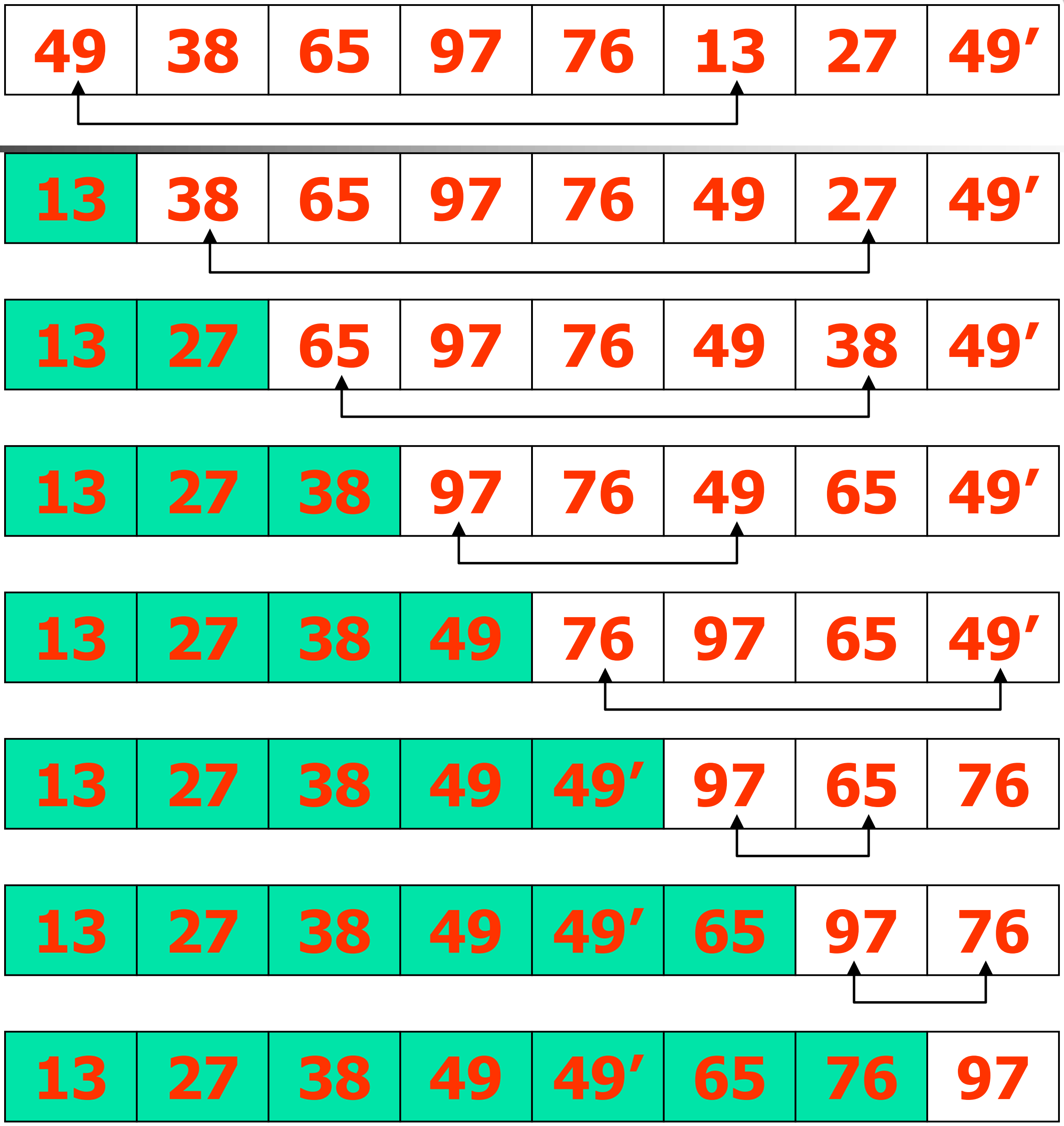
- 记录的移动次数：

  当初始文件为正序时，移动次数为0。文件初态为反序时，每趟排序均要执行交换操作，总的移动次数取最大值$3(n-1)$。简单选择排序的平均时间复杂度为 $O(n^2)$。

- 附加空间：直接选择排序是一个就地排序。
- 稳定性分析：

  直接选择排序是不稳定的。

简单选择排序示例



49	38	65	97	76	13	27	49′
13	38	65	97	76	49	27	49′
13	27	65	97	76	49	38	49′
13	27	38	97	76	49	65	49′
13	27	38	49	76	97	65	49′
13	27	38	49	49′	97	65	76
13	27	38	49	49′	65	97	76
13	27	38	49	49′	65	76	97

5.2 冒泡排序

冒泡排序算法思想:

- 设待排序的记录数组为A[1..n]，初始排序范围从A[1]到A[n]
- 在第i遍排序时，排序范围为A[i]到A[n]，在当前的排序范围之内，自右至左对相邻的两个结点依次进行比较，让值较大的结点往下沉(右移)，让值较小的结点往上冒(左移)。每趟起泡都能保证值最小的结点上移至最左边，即A[i]的位置，下一遍的排序范围为从下一结点A[i+1]到A[n]。
- 在整个排序过程中，最多执行(n-1)遍。但执行的遍数可能少于(n-1)，这是因为在执行某一遍的各次比较没有出现结点交换时，就不用进行下一遍的比较。

冒泡排序算法

```
BUBBLE-SORT( A )
1.  for $i \leftarrow 1$ to $n$-1
2.      do noswap＝TRUE;
3.      for $j \leftarrow n$-1  downto  $i$
4.          do if (A[ $j$+1 ]<A[ $j$ ])
5.              then A[ $j$ ]$\leftrightarrow$A[ $j$+1 ];
6.                  noswap＝FALSE;
7.      if (noswap) break;
```

冒泡排序算法分析

关键字的比较次数和对象移动次数：

- 在最好情况下，初始状态是递增有序的，一趟扫描就可完成排序，关键字的比较次数为 n-1，没有记录移动。
- 若初始状态是反序的，则需要进行 n-1趟扫描，每趟扫描要进行 n- i 次关键字的比较，且每次需要移动记录三次，因此，最大比较次数和移动次数分别为：

$$\text{比较次数的最大值} = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

$$\text{移动次数的最大值} = \sum_{i=1}^{n-1}3(n-i) = 3n(n-1)/2 = O(n^2)$$

- 冒泡排序方法是稳定的。

冒泡排序示例

i	(0)	(1)	(2)	(3)	(4)	(5)
	21	25	49	25*	16	08
1	08	21	25	49	25*	16
2	08	16	21	25	49	25*
3	08	16	21	25	25*	49
4	08	16	21	25	25*	49

5.3 Shell 排序

1959年由D.L. Shell提出，又称缩小增量排序(Diminishing-increment sort)

在插入排序中，只比较相邻的结点，一次比较最多把结点移动一个位置。如果对位置间隔较大距离的结点进行比较，使得结点在比较以后能够一次跨过较大的距离，这样就可以提高排序的速度。

Shell 排序算法思想

- **希尔排序基本思想**

先取一个小于n的整数 d_1 作为第一个增量，把文件的全部记录分成d_1个组。所有距离为 d_1 的倍数的记录放在同一个组中。 先在各组内进行直接插入排序；然后，取第二个增量 $d_2 < d_1$重复上述的分组和排序，直至所取的增量 $d_t = 1$ $(d_t < d_{t-1} < \cdots < d_2 < d_1)$，即所有记录放在同一组中进行直接插入排序为止。该方法实质上是一种分组插入方法。

Shell 算法描述

```
Shell-Pass( A， d)                              //希尔排序中的一趟排序，d为当前增量//
1.  for i←d+1 to n                           //将A[d+1．．n]分别插入各组当前的有序区//
2.      do if (A[i]<A[i-d])
3.          then  A[0]←A[i];  j←i-d;                 //A[0]只是暂存单元，不是哨兵//
4.                while (j>0 && A[0]<A[j].key)           //查找R[i]的插入位置//
5.                    do A[j+d]←A[j];                           //后移记录//
6.                       j←j-d;                                //查找前一记录//
7.                A[j+d]←A[0];                          //插入A[i]到正确的位置上//

ShellSort(A, D)
1.  i←1;
2.  while( i ≤ Length[D])
3.      do  increment←D[i]; i←i+1;
4.          Shell-Pass(A， increment)；   //一趟增量为increment的Shell插入排序//
```

Shell 算法描述 （续）

```
Shell-Pass( A， d)                         //希尔排序中的一趟排序，d为当前增量//
1.  for i←d+1 to n                  //将A[d+1．．n]分别插入各组当前的有序区//
2.     do if (A[i]<A[i-d])
3.          then  A[0]←A[i]; j←i-d;               //A[0]只是暂存单元，不是哨兵//
4.                 while (j>0 && A[0]<A[j].key)          //查找R[i]的插入位置//
5.                      do A[j+d]←A[j];                          //后移记录//
6.                          j←j-d;                           //查找前一记录//
7.                  A[j+d]←A[0];                     //插入A[i]到正确的位置上//
```

```
Shell-Sort(A)
1.   increment←m；  //增量初值，不妨设 m>0  //
2.   while(increment>1)
3.      do increment← increment/3+1；  //求下一增量 //
4.          Shell-Pass(A，increment)；  //一趟增量为 increment的Shell插入排序 //
```

希尔排序示例

i	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	增量d_i
	21	25	49	25*	16	08	27	04	55	48	
1	21	-	-	25*	-	-	27	-	-	48	3
		25	-	-	16	-	-	04			
			49	-	-	08	-	-	55		
	21	04	08	25*	16	49	27	25	55	48	
2	21	-	08	-	16	-	27	-	55		2
		04	-	25*	-	49	-	25	-	48	
	08	04	16	25*	21	25	27	48	55	49	
3	08	04	16	25*	21	25	27	48	55	49	1
	04	08	16	21	25*	25	27	48	49	55	

Shell 排序算法分析

- Shell排序的运行时间依赖于增量序列，增量序列应满足：

  ① 最后一个增量必须为1；

  ② 应该尽量避免序列中的值互为倍数。

- Shell排序的时间性能优于直接插入排序

  ① 当文件基本有序时直接插入排序所需比较和移动次数均较少。

  ② 当 n值较小时，直接插入排序的最好和最坏时间复杂度差别不大。

  ③ 希尔排序在开始时增量较大，分组较多，每组记录数少，各组内直接插入排序较快；随着增量 d_i 逐渐缩小，分组数减少，各组的记录数逐渐增多，但由于已经按d_{i-1}作为增量排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快。因此，希尔排序的实际效率较直接插入排序有较大改进。

Shell 排序算法分析

- 对希尔排序的复杂度的分析很困难，在特定情况下可以准确地估算关键字的比较和对象移动次数，但是考虑到与增量之间的依赖关系，并要给出完整的数学分析，目前还做不到。
- Knuth的统计结论是，平均比较次数和对象平均移动次数在 $n^{1.25}$ 与 $1.6n^{1.25}$之间。
- 目前，关于希尔排序上下界的很多问题仍然没有得到圆满的解决，尽管很多人尝试去做。希尔排序易于实现，并且无论是对于接近有序的文件还是完全无序的文件，它都优于其它算法，而且希尔排序对空间要求低。

增量序列与运行时间的分析

- Stasevich,1965;Pratt,1971：增量序列为 $2^n - 1$ (即1, 3, 7, 15, 31…) 时，希尔排序的时间复杂度为 $\Theta(N^{3/2})$。
- Pratt，1971：增量序列为 $2^i 3^j$ (即1，2，3，4，6，9，8，12…)时，希尔排序的时间复杂度为 $O(N(\log(N))^2)$，由于增量太多(增量序列太长), 在实际中并不具有竞争力。
- Sedgewick,1982：增量序列为 $4^{j+1} + 3*2^j + 1$(即1，8,23,77…)时，希尔排序的时间复杂度为 $O(N^{4/3})$。
- Sedgewick,1985；Selmer,1987：存在长为 O(log(N))的增量序列，使得希尔排序的时间复杂度为 $O(N^{1+(1/k)})$ 。

增量序列与运行时间的分析

- Poonen：某一常数 c>0，在最坏情况下，M 趟排序一个长为n的文件，希尔排序的比较次数为 $\Omega(n^{1+c/m})$, $m=M^{1/2}$。
- Plaxton 和 Suel 给出了同样结果的证明，如果取 $M=\Omega(\log n)$ 可得Sedgewick 的方法对于较短的增量序列是最佳的。
- Cypher：具有递减增量序列的希尔排序需要的比较交换次数至少为 $\Omega(N(\log N)^2/\log\log N)$ 。Cypher的结果同增量序列的长度无关，但是只适用于单调增量序列。

Shell排序的平均时间复杂度

- Tao Jiang，Ming li及 Paul vitany 在1999年给出了希尔排序在平均复杂度下的一个下界：对于任意的增量序列，*p* 趟希尔排序的平均比较次数为$\Omega(p\, n^{1+(1/p)})$。
- S.Janson,E.Knuth : 如果 $h=\Theta(n^{7/15})$, $g=\Theta(n^{1/5})$，g，h互质，则 (h，g，1) 希尔排序的时间复杂度 $O(n^{23/15})$。
- **课外补充学习：**有关Shell排序的最新研究成果?

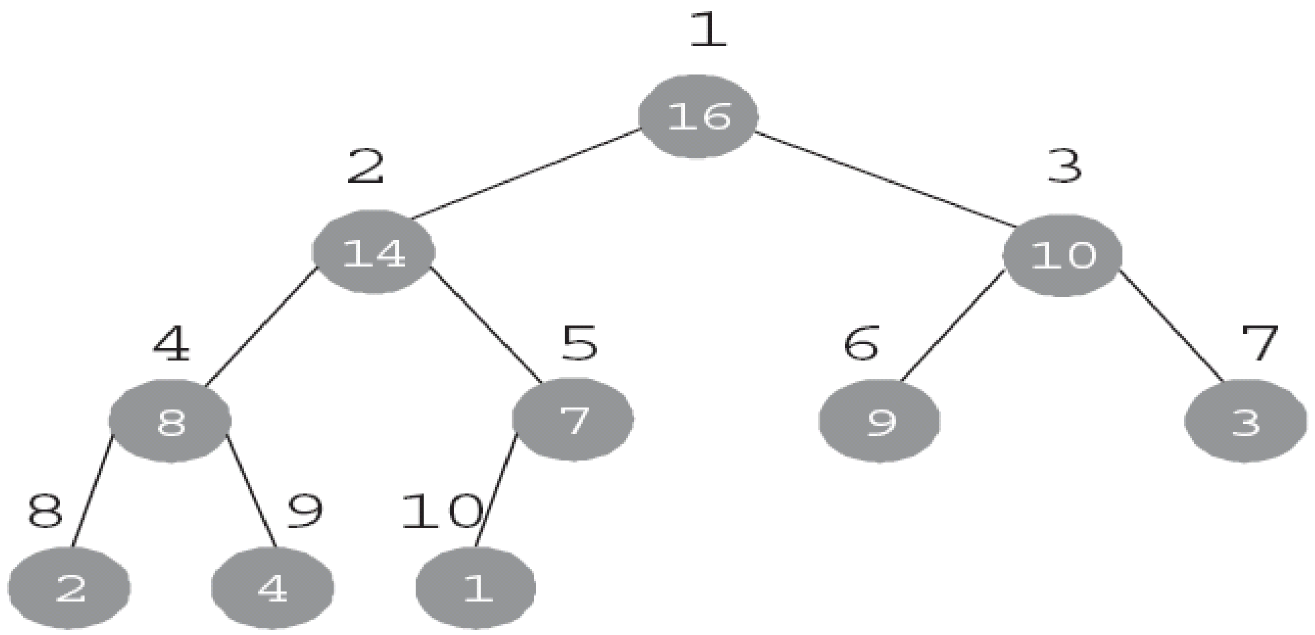
  （从图书馆、网络等多种途径进行调研）

6. HEAPSORT

- A sorting algorithm which combines the better attributes of merge sort and insertion sort;
- The worst case running time is $O(n \cdot \log n)$;
- It sorts in place and is not Stable;
- It introduces a new data structure--heap(堆)

6.1 Heaps

- **Heap**： a data structure which is an array object that can be viewed as a complete binary tree （完全二叉树）



堆的表示和存贮

- An array *A* that represents a heap is an object with two attributes:
- *Length* [*A*] -- the number of elements in the array *A*
- *Heap-size* [*A*] -- the number of elements in the heap stored within array *A*
- *Heap-size* [*A*] $\leq$ *Length* [*A*]

大根堆、小根堆

- There are two kinds of binary heaps:

  Max-heap,   Min-heap

- Max-heap-- for every node i other than the root,

  $$A\,[\mathrm{PARENT}(i\,)] \geq A[i\,]\;;$$

- Min-heap-- for every node i other than the root,

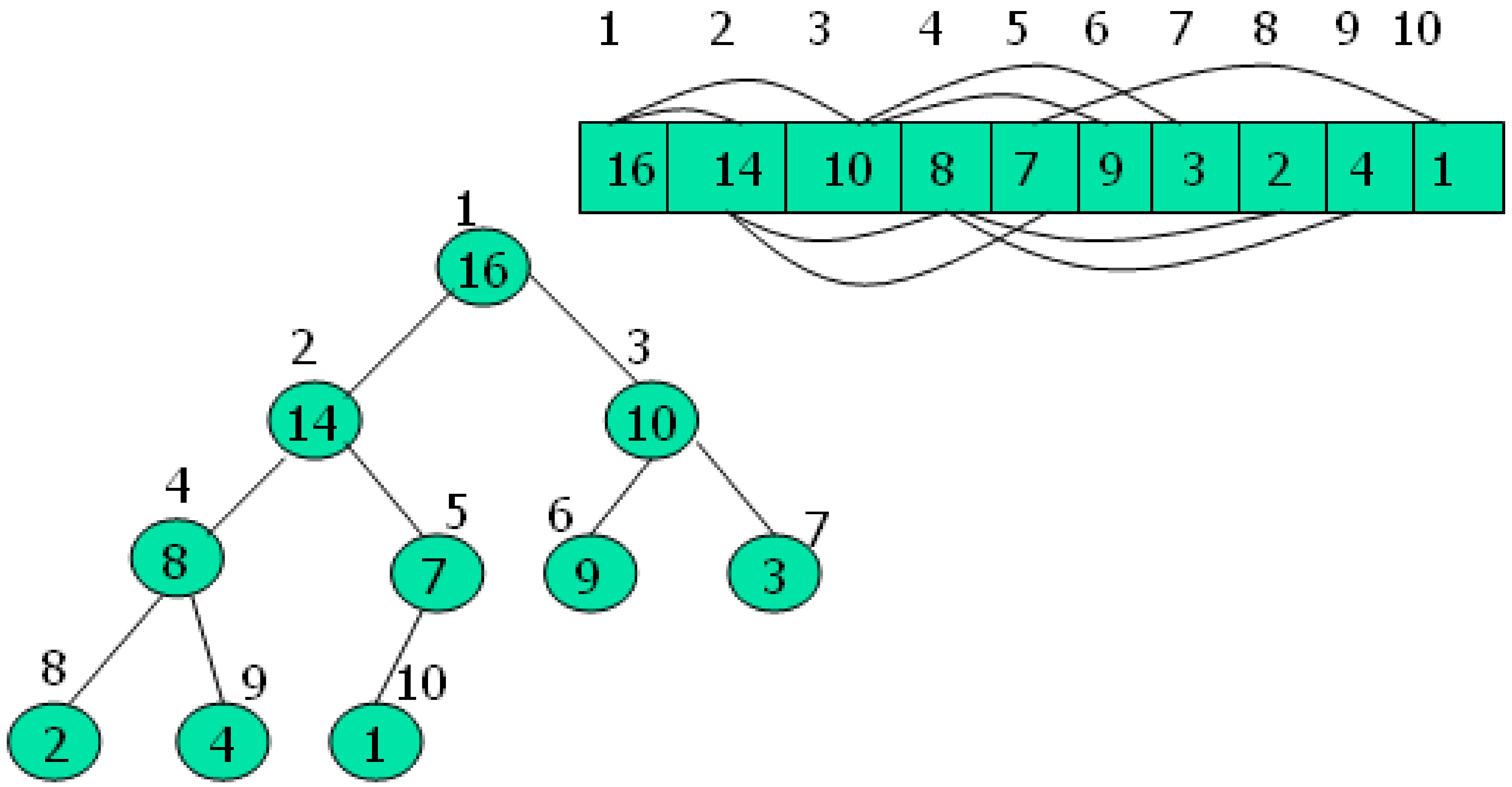
  $$A\,[\mathrm{PARENT}(i\,)] \leq A[i\,]\;;$$

- The root of the tree is $A[1]$

堆的性质

- Given the index *i* of a node, the indices of its parent PARENT(*i*), left child LEFT(*i*), and right child RIGHT(*i*) can be computed simply:
  - PARENT(*i*) **return** $\lfloor i/2 \rfloor$
  - LEFT(*i*) **return** $2i$
  - RIGHT(*i*) **return** $2i + 1$

Example of max-heap



Height of the Heap

- ***Height*** of a node in a heap—

  The number of edges on the longest simple path from the node to a leaf.

- The height of the heap -- is the height of its root.
- The basic operations on heaps take $O(\log n)$ time

Homework 6.1

- **Page 74: 6.1-3, 6.1-6;**

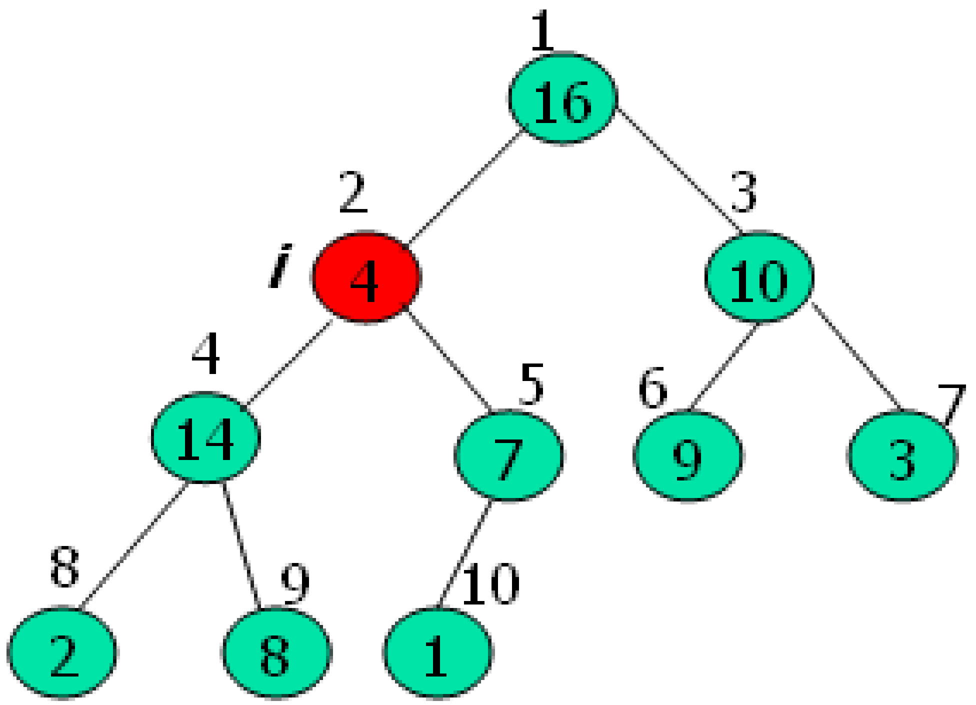
6.2 维护堆

- Let the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but *A*[*i*] may be smaller than its children.
- To maintaining the max-heap property, we using MAX-HEAPIFY procedure, which runs in *O*(log *n*) time.
- When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps.

调整为堆

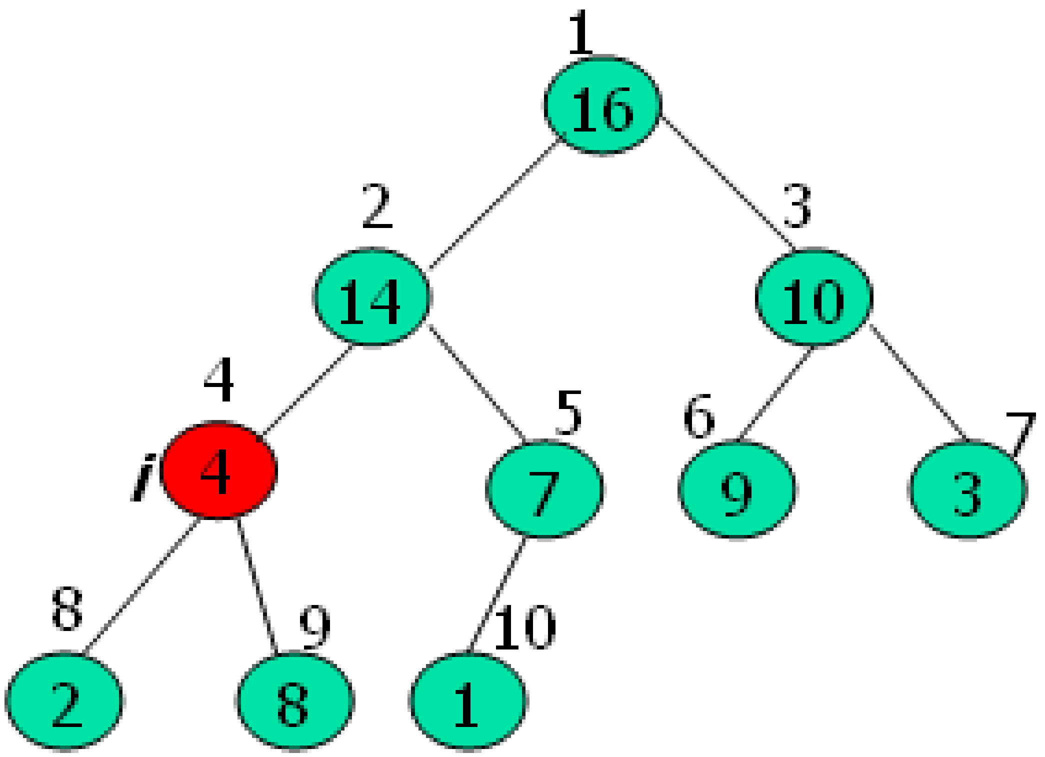
```
MAX-HEAPIFY($A$, $i$)
 1  $l \leftarrow$ LEFT($i$)
 2  $r \leftarrow$ RIGHT($i$)
 3  if $l \le heap\text{-}size[A]$ and $A[l] > A[i]$
 4      then $Largest \leftarrow l$
 5      else $Largest \leftarrow i$
 6  if $r \le heap\text{-}size[A]$ and $A[r] > A[Largest]$
 7      then $Largest \leftarrow r$
 8  if $Largest \neq i$
 9      then exchange $A[i] \leftrightarrow A[Largest]$
10          MAX-HEAPIFY($A$, $Largest$)
```

Example: MAX-HEAPIFY



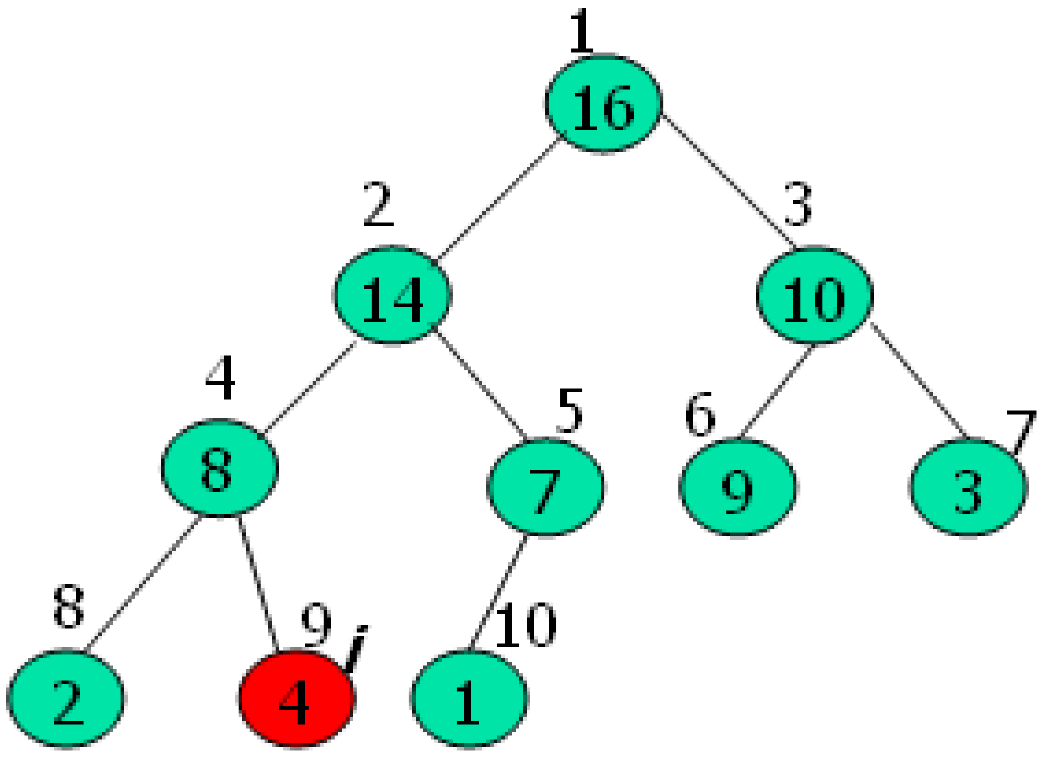
Step 1

Example: MAX-HEAPIFY



Step 2

Example: MAX-HEAPIFY



Step 3

分析 MAX-HEAPIFY 的运行时间

- MAX-HEAPIFY 的运行时间可以用下面的递归递归方程描述：

$$T(n) \le T(2n/3) + \Theta(1)$$

$$T(n) = O(\log n)$$

- It takes Θ(1) time to fix up the relationships among the elements $A[i]$, $A[\mathrm{LEFT}(i)]$ and $A[\mathrm{RIGHT}(i)]$;
- The Child's sub-trees each have size at most $2n/3$ — the worst case occurs when the last row of the tree is exactly half full.
- The time to run M**AX**-H**EAPIFY** on a sub-tree rooted at one of the Child of node *i* is no larger than $T(2n/3)$.

Homework 6.2

- **Page 76**: 6.2-1, 6.2-2， 6.2-4;

6.3 建堆： Building a heap

- BUILD-MAX-HEAP 可将任意的一个数组$A[1 \cdot\cdot n]$ 调整为大根堆, 其中 $n = Length[A]$
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) \cdot\cdot \quad \cdot\cdot n]$ are all leaves of the tree.
- $Length[A]/2$ is the last node that is not leaf, the BUILD-MAX-HEAP procedure goes through the remaining internal nodes of the tree and runs MAX-HEAPIFY on each one.

建堆算法

```
BUILD-MAX-HEAP(A)
  1  Heap-size[A] ← Length[A]
  2  for i ← ⌊Length[A]/2⌋ downto 1
  3        do MAX-HEAPIFY(A, i)
```

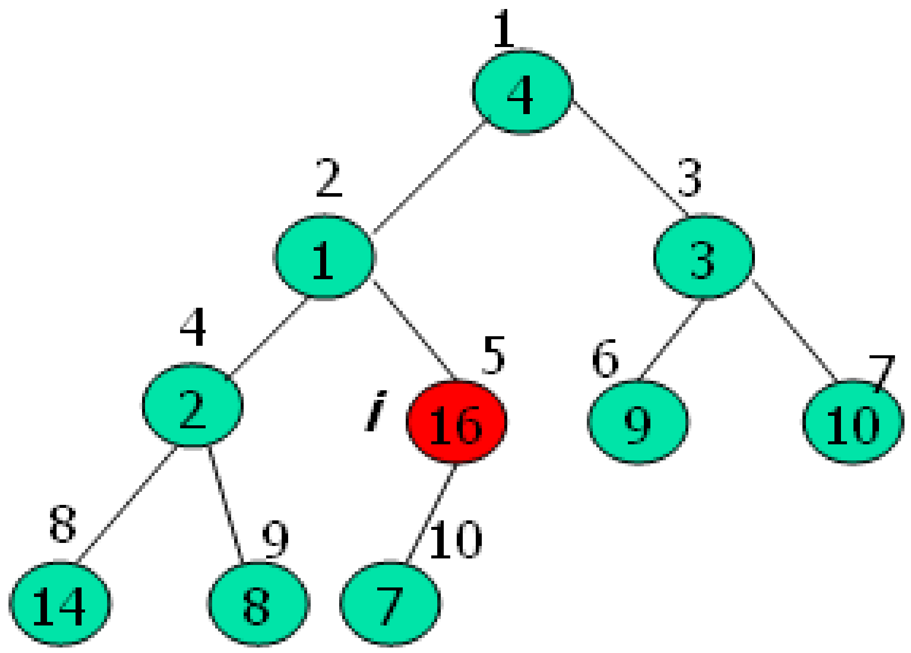
- A simple upper bound on the running time of BUILD-MAX-HEAP is as follows:
  - Each call to MAX-HEAPIFY costs $O(\log n)$ time
  - There are $O(n)$ such calls ;
  - The running time is $O(n \log n)$;
  - This upper bound, though correct, is not asymptotically tight !!!

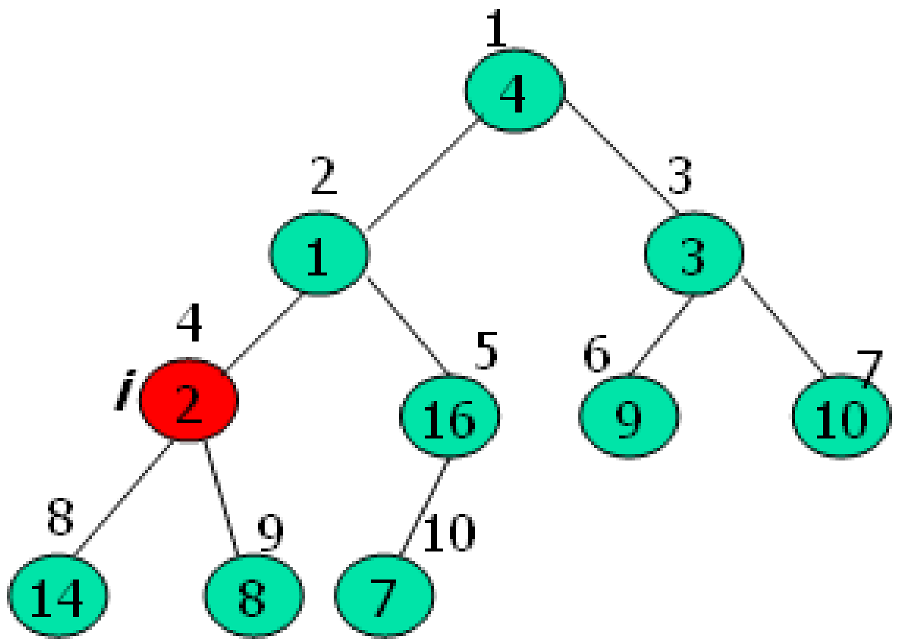
School of CST

UNIVERSITY OF SCIENCE & TECHNOLOGY OF CHINA

Example: 建堆

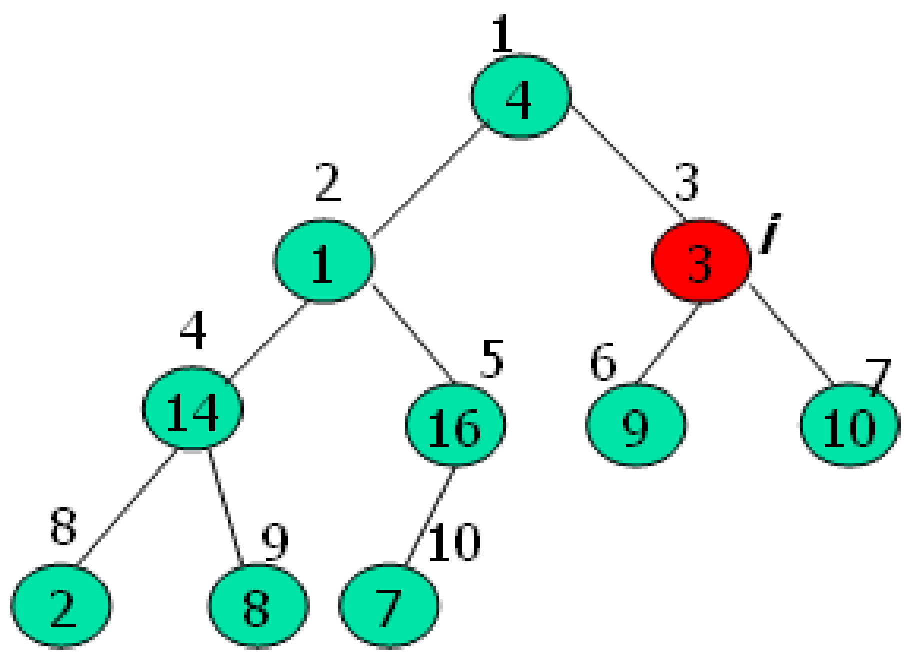
A	4	1	3	2	16	9	10	14	8	7

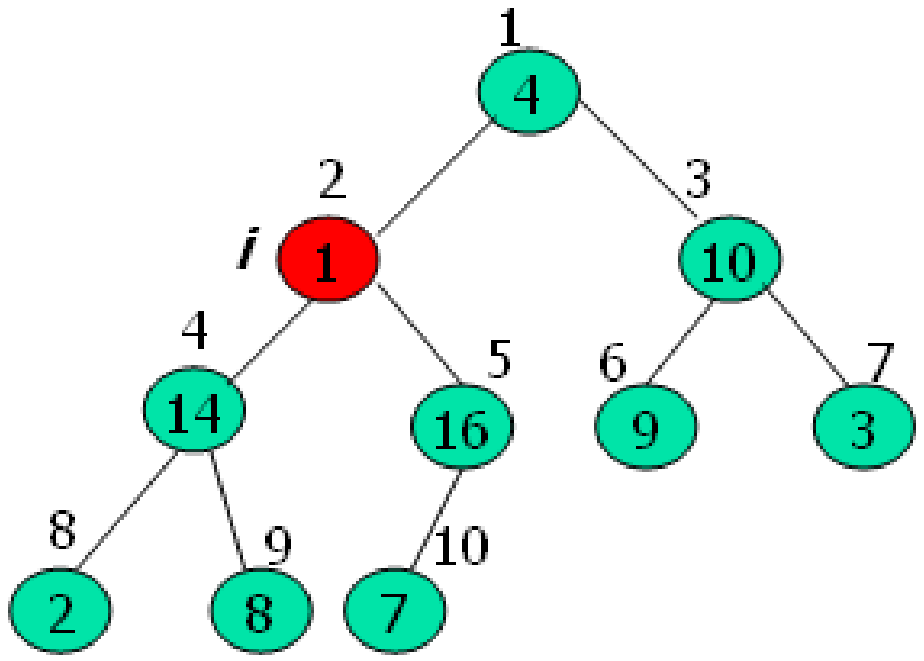




Example: 建堆

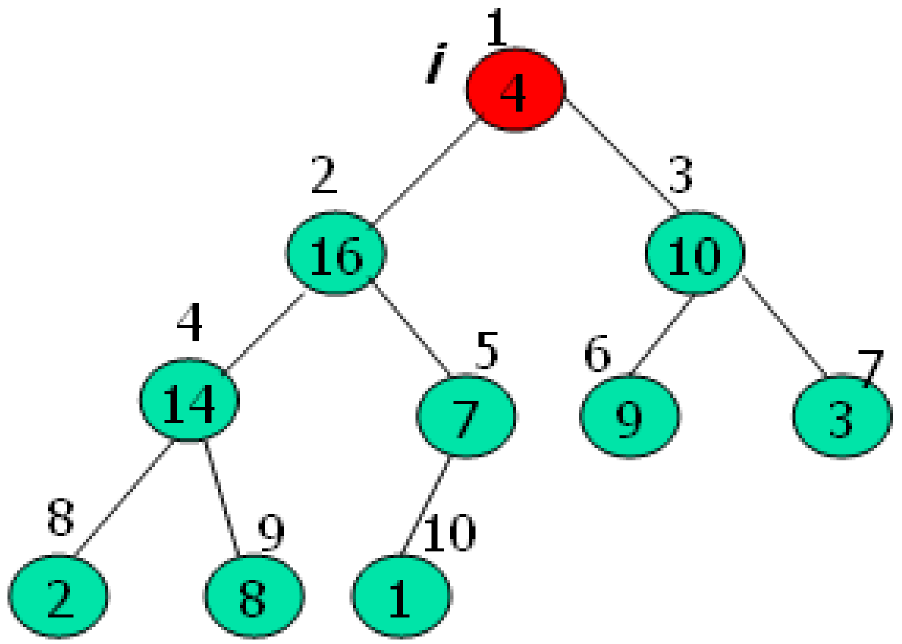
A	4	1	3	14	16	9	10	2	8	7

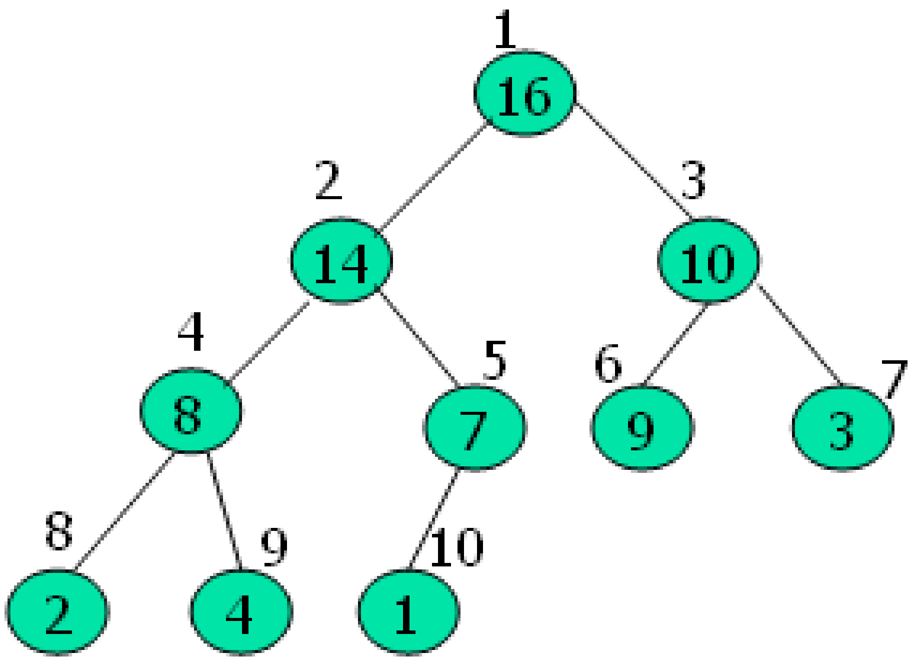




Example: 建堆

A	4	16	10	14	7	9	3	2	8	1





建堆算法运行时间分析

- The time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree;
- The heights of most nodes are small;

An n-element heap has height $\lfloor \log n \rfloor$;

- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

建堆算法运行时间分析（续）

- The total cost of BUILD-MAX-HEAP is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

So BUILD-MAX-HEAP procedure runs in linear time !

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

Homework 6.3

- Page 78: 6.3-1, 6.3-3;

6.4 The heapsort algorithm

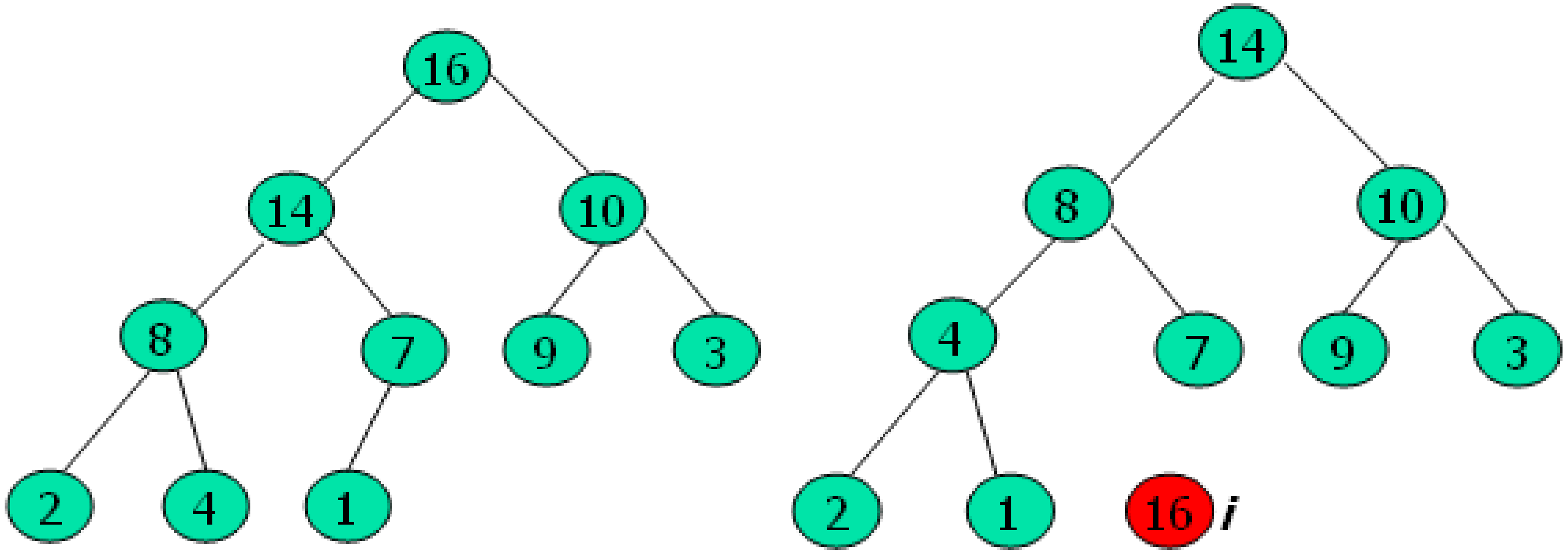
- The HEAPSORT procedure sorts an array $A[1 \cdot\cdot n]$ in place, where $n = Length[A]$. It runs in $O(n \log n)$ time.

- **HEAPSORT**(A)

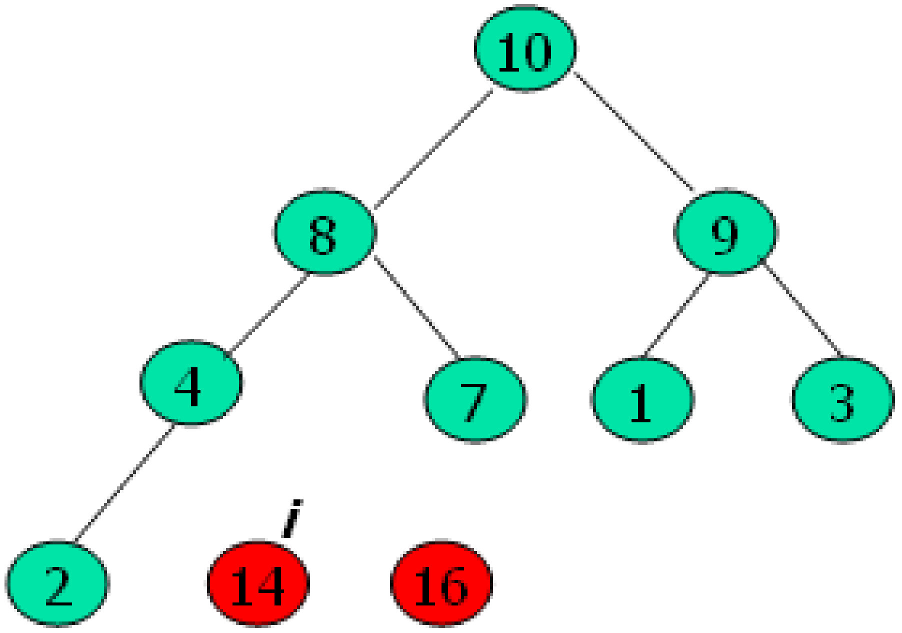
```
1  BUILD-MAX-HEAP($A$)
2  for $i \leftarrow Length[A]$ downto 2
3       do exchange $A[1] \leftrightarrow A[i]$
4            $Heap\text{-}size[A] \leftarrow Heap\text{-}size[A] - 1$
5            MAX-HEAPIFY($A$, 1)
```

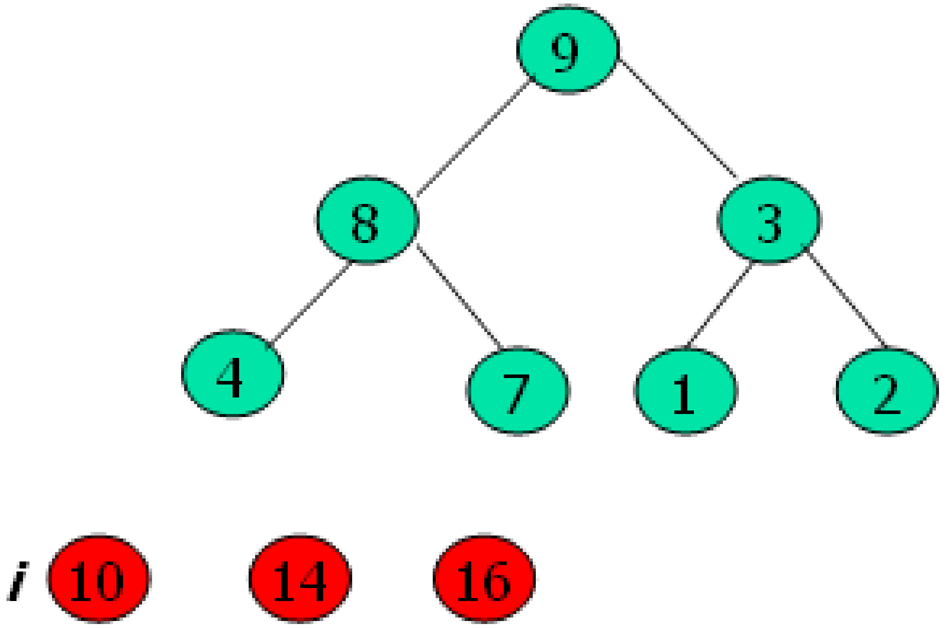
Example of heapsort

- The operation of heapsort after the max-heap is initially built:

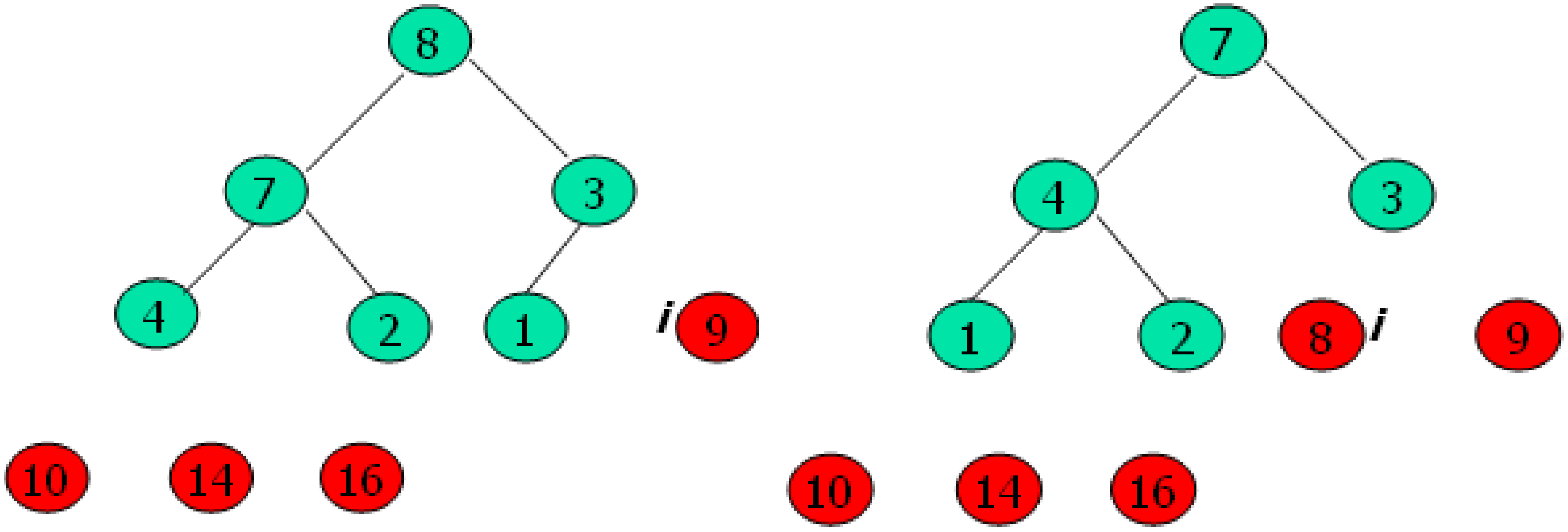


Example of heapsort (续)

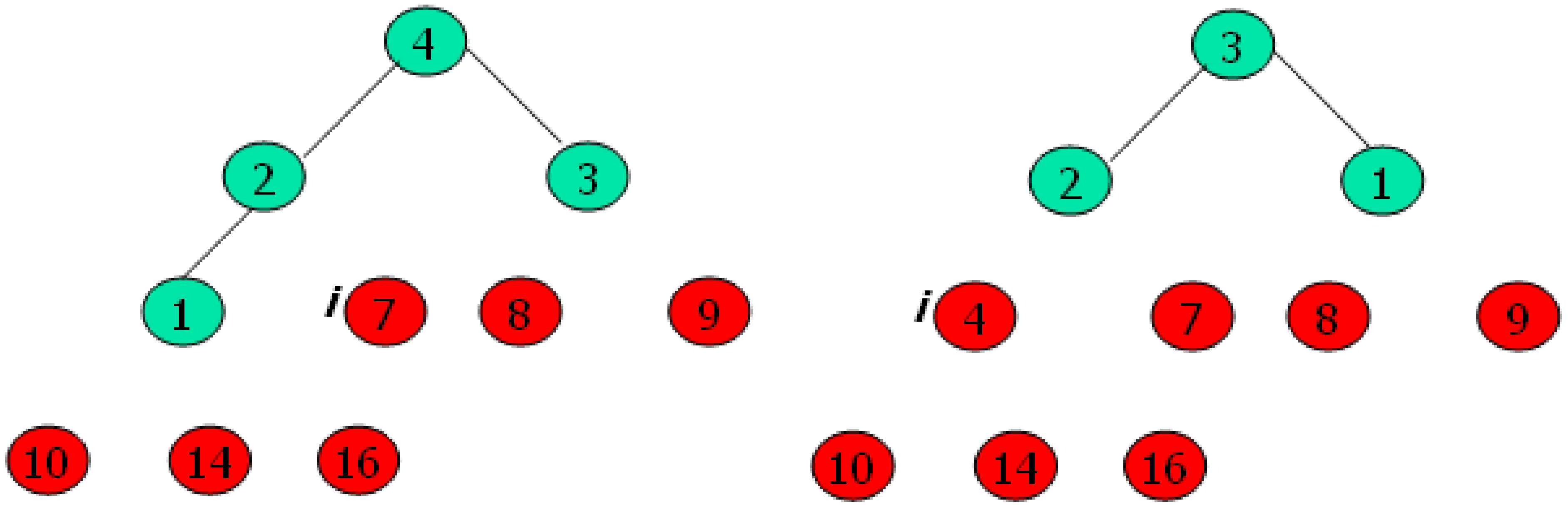




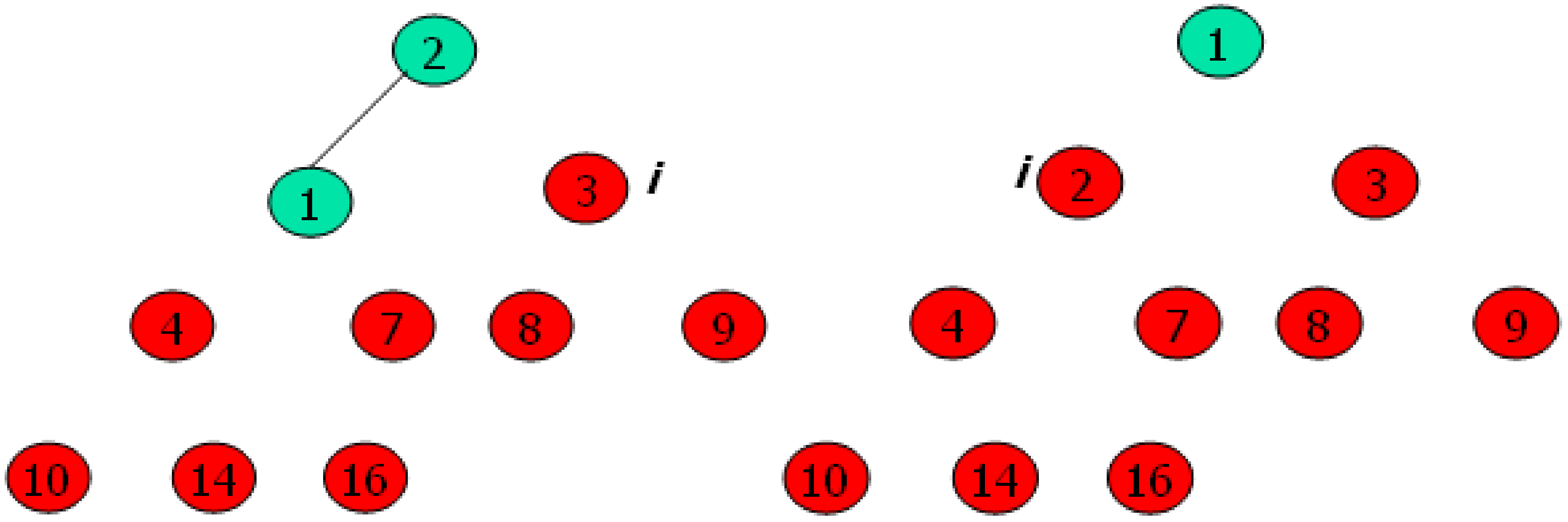
Example of heapsort (续)



Example of heapsort (续)



Example of heapsort (续)



A

1	2	3	4	7	8	9	10	14	16

The Running Time of Heapsort

- The call to BUILD-MAX-HEAP takes time $O(n)$
- Each of the n - 1 calls to MAX-HEAPIFY takes time $O(\log n)$
- The HEAPSORT procedure takes time $O(n \cdot \log n)$

Homework 6.4

- Page 80: 6.4-3， 6.4-4

6.5 Priority queues

- A ***priority queue*** is a data structure for maintaining a set *S* of elements, each with an associated value called a ***key***;
- It is one of the most popular applications of a heap ;
- There are two kinds of priority queues:

  max-priority queues, min-priority queues;

优先队列的基本操作

- A ***max-priority queue*** supports the following operations:
- INSERT(*S*, *x*) : inserts the element *x* into the set *S*.
- MAXIMUM(*S*) : returns the element of *S* with the largest key.
- EXTRACT-MAX(*S*): removes and returns the element of *S* with the largest key.
- INCREASE-KEY(*S*, *x*, *k*) : increases the value of element *x*'s key to the new value *k*, which is assumed to be at least as large as *x*'s current key value.

优先队列的基本操作（续）

```
HEAP-MAXIMUM($A$)
  1  return  $A[1]$
```

Its running time is $\Theta(1)$

```
HEAP-EXTRACT-MAX($A$)
  1  if $heap\text{-}size[A] < 1$
  2      then error "heap underflow"
  3  $max \leftarrow A[1]$
  4  $A[1] \leftarrow A[heap\text{-}size[A]]$
  5  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
  6  MAX-HEAPIFY($A$, 1)
  7  return $max$
```

Its running time is $O(\log n)$

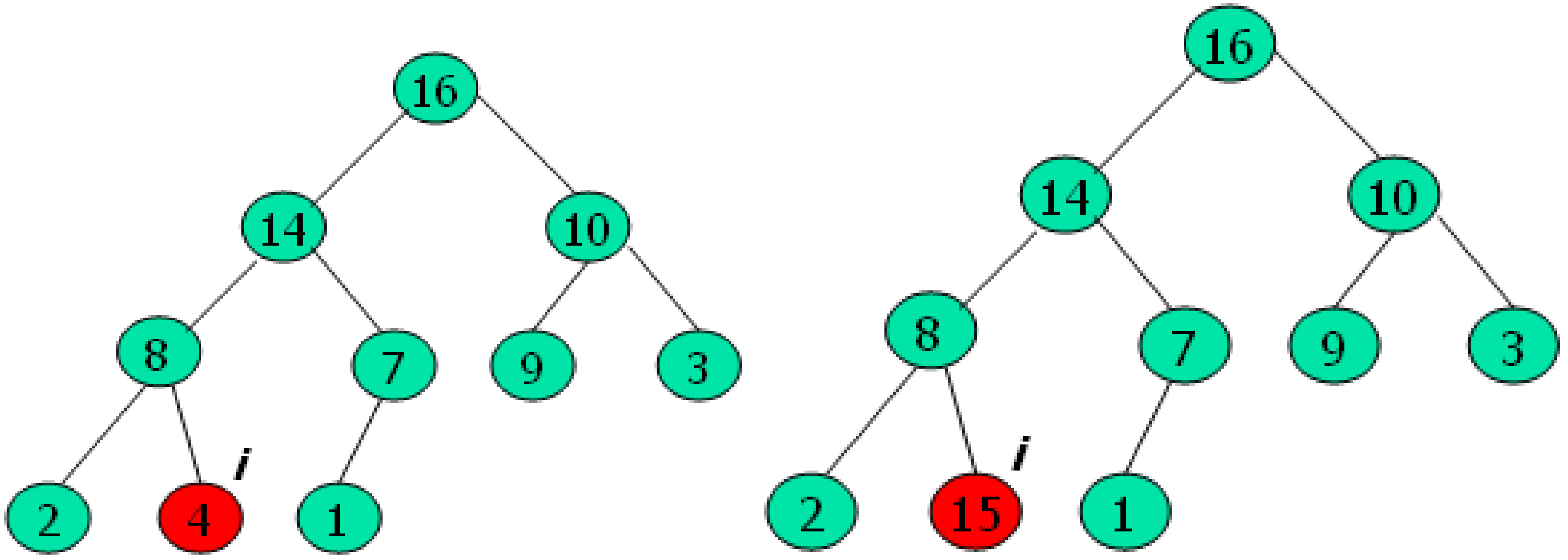
优先队列的基本操作（续）

```
HEAP-INCREASE-KEY($A$, $i$, $key$)
1  if $key < A[i]$
2        then error "new key is smaller than current key"
3  $A[i] \leftarrow key$
4  while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5        do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6              $i \leftarrow \text{PARENT}(i)$
```

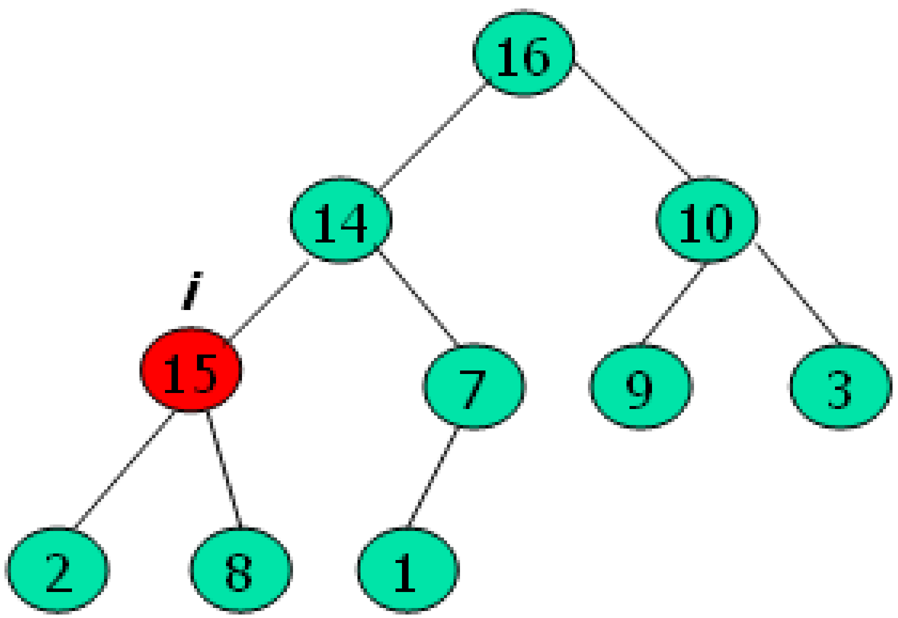
Its running time is $O(\log n)$

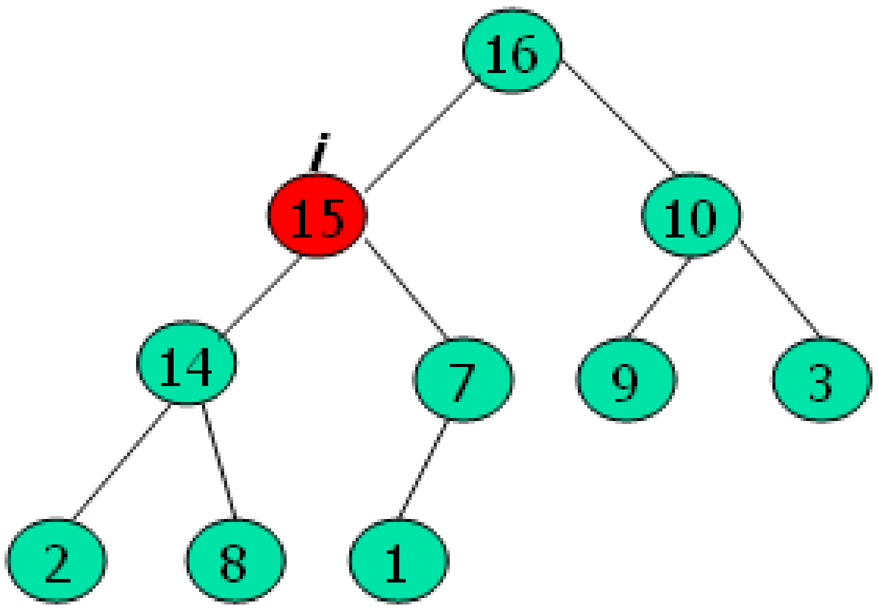
Example：HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(*A*，*i*, 15) operation



Example： HEAP-INCREASE-KEY





优先队列的基本操作（续）

```
MAX-HEAP-INSERT($A$, $key$)
    1  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
    2  $A[heap\text{-}size[A]] \leftarrow -\infty$
    3  HEAP-INCREASE-KEY($A$, $heap\text{-}size[A]$, $key$)
```

- The running time of MAX-HEAP-INSERT on an *n*-element heap is $O(\log n)$
- A heap can support any priority-queue operation on a set of size *n* in $O(\log n)$ time

Homework 6.5

- Page 82: 6.5-3, 6.5-7;

7. Quicksort

- Quicksort is a in place sorting algorithm, its worst-case running time is $\Theta(n^2)$;
- The average case running time is $\Theta(n \log n)$, and the constant factors hidden in the $\Theta(n \log n)$ notation are quite small ;
- It sorts in place.
- Quicksort is based on the divide-and-conquer paradigm.

7.1 Description of quicksort

- The three-step divide-and-conquer process for sorting a typical subarray $A[p \cdot\cdot r]$ is as follows:
  - **Divide:** Partition the array $A[p \cdot\cdot r]$ into two subarrays $A[p \cdot\cdot q - 1]$ and $A[q + 1 \cdot\cdot r]$ such that each element of $A[p \cdot\cdot q - 1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q + 1 \cdot\cdot r]$；
  - **Conquer:** Sort the two subarrays $A[p \cdot\cdot q -1]$ and $A[q +1 \cdot\cdot r]$ by recursive calls to quicksort；
  - **Combine:** no work is needed to combine them and the entire array $A[p \cdot\cdot r]$ is now sorted。

Quicksort 伪代码

```
QUICKSORT($A$, $p$, $r$)
 1  if $p < r$
 2        then  $q \leftarrow$ PARTITION($A$, $p$, $r$)
 3              QUICKSORT($A$, $p$, $q - 1$)
 4              QUICKSORT($A$, $q + 1$, $r$)
```

- The initial call is QUICKSORT(A, 1, *Length*[A])
- Where the PARTITION procedure rearranges the subarray $A[p \cdot\cdot r]$ in place;

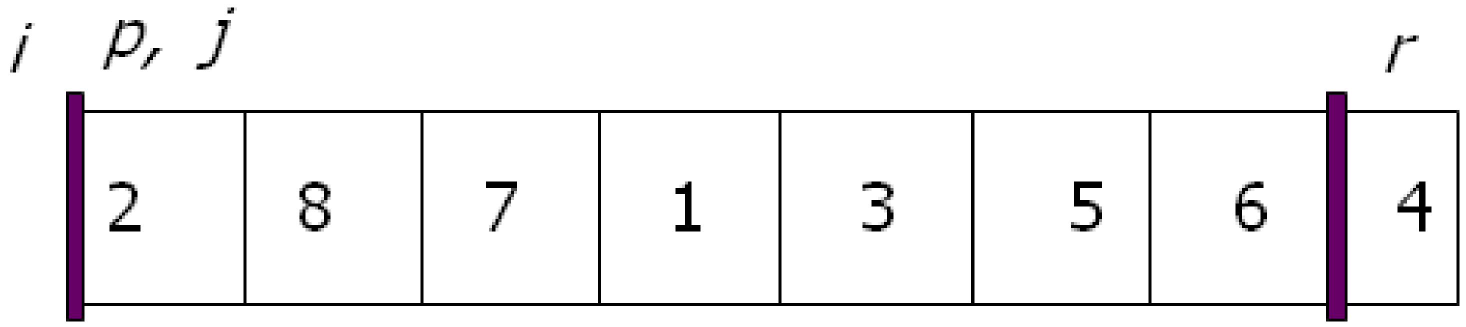
PARTITION 伪代码 (1)

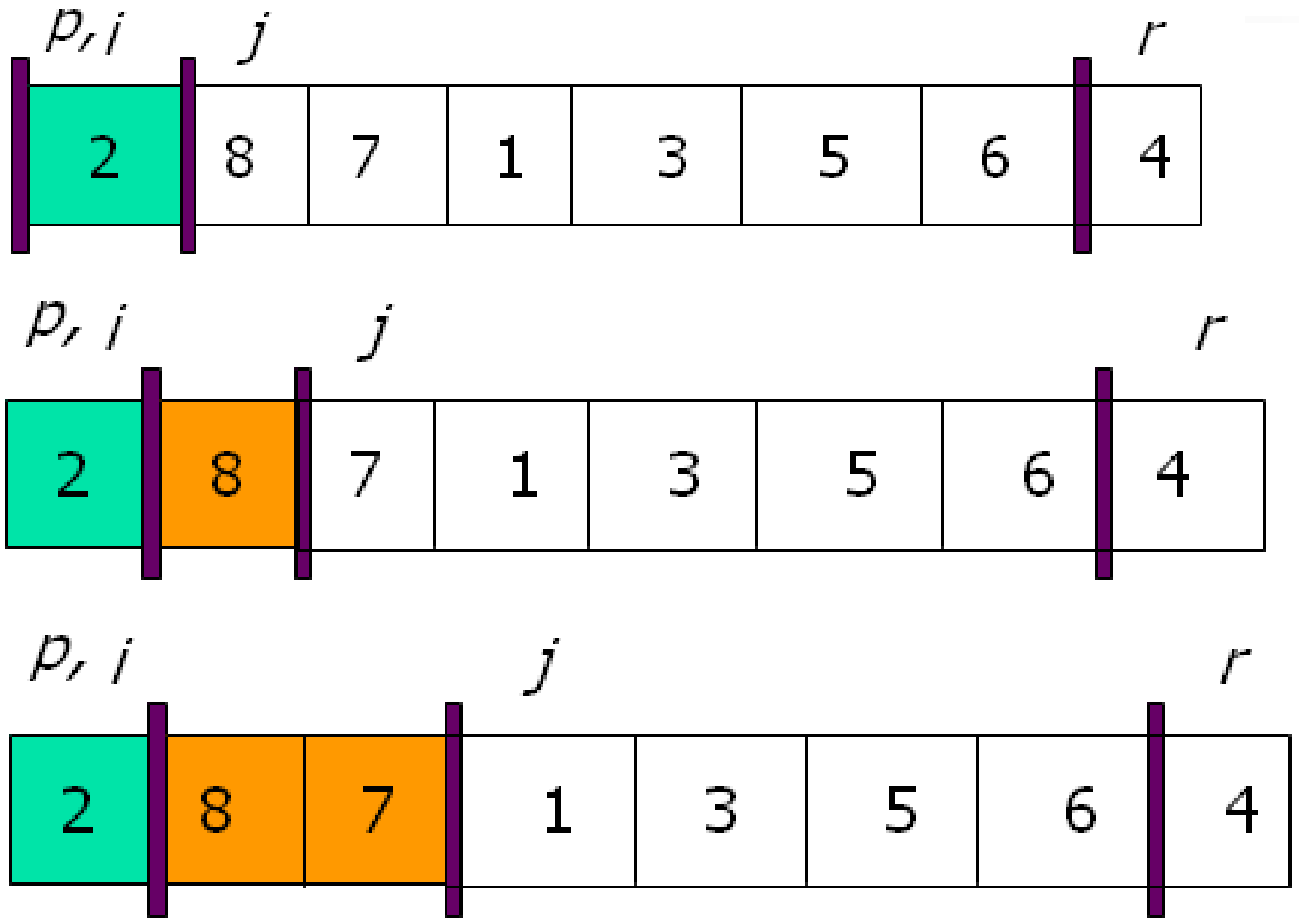
```
PARTITION($A$, $p$, $r$)
  1  $x \leftarrow A[r]$
  2  $i \leftarrow p - 1$
  3  for $j \leftarrow p$ to $r - 1$
  4        do if $A[j] \leq x$
  5              then $i \leftarrow i + 1$
  6                   exchange $A[i] \leftrightarrow A[j]$
  7  exchange $A[i + 1] \leftrightarrow A[r]$
  8  return $i + 1$
```

The running time of PARTITION on the subarray $A[p \cdot\cdot r]$ is $\Theta(n)$, where $n = r - p + 1$.

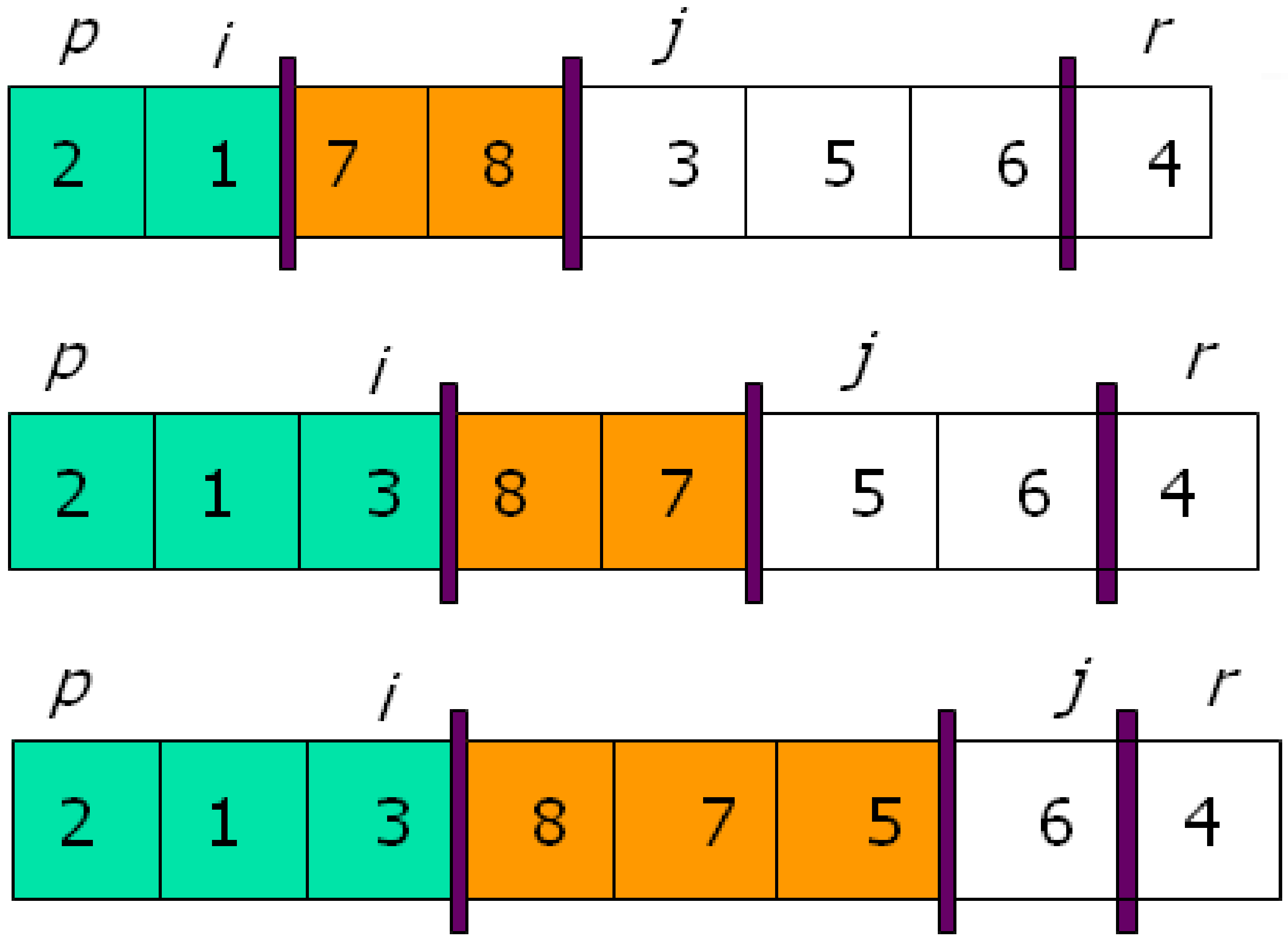
Example: PARTITION

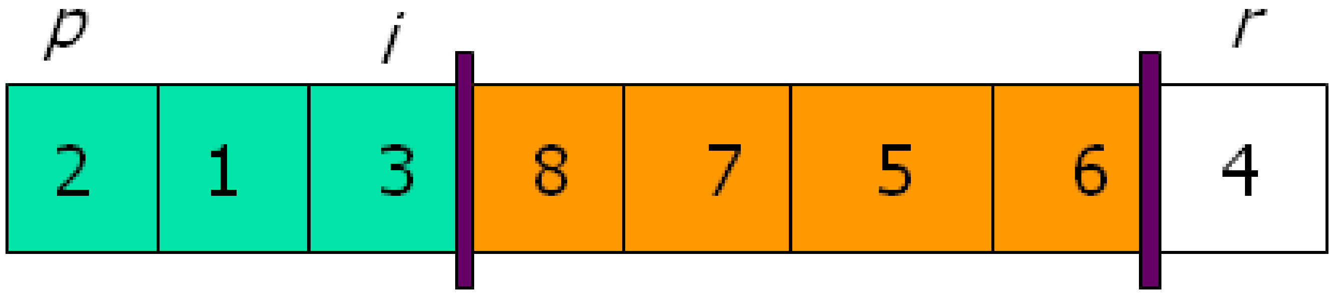
The operation of PARTITION on an 8-element array is as follows:

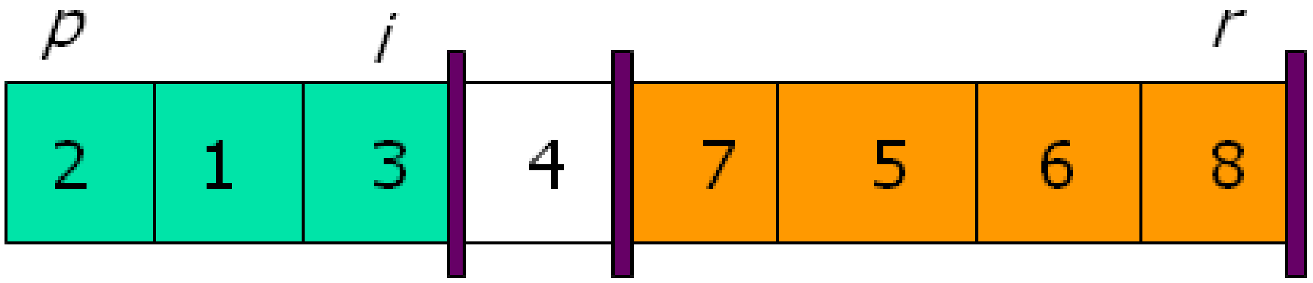




p, i
j
r
2 8 7 1 3 5 6 4
p, i
j
r
2 8 7 1 3 5 6 4
p, i
j
r
2 8 7 1 3 5 6 4


p
i
j
r
2
1
7
8
3
5
6
4
p
i
j
r
2
1
3
8
7
5
6
4
p
i
j
r
2
1
3
8
7
5
6
4


p
i
r
2
1
3
8
7
5
6
4


p
i
r
2
1
3
4
7
5
6
8

另一种 PARTITION 伪代码

```
PARTITION($A$, $p$, $r$)
1.  $i \leftarrow p$;  $j \leftarrow r$;  $temp \leftarrow A[i]$;
 while ( $i \neq j$ )
       do while (($A[j] \geq temp$) && ( $i<j$ ))
              do  $j \leftarrow j-1$;
           if ( $i<j$ )  then   $A[i] \leftarrow A[j]$; $i \leftarrow i+1$;
           while (($A[i] \leq temp$) && ( $i<j$ ))
              do  $i \leftarrow i+1$;
           if ( $i<j$ ) then  $A[j] \leftarrow A[i]$; $j \leftarrow j-1$;
  $A[i] \leftarrow temp$;
  return $i$;
```

以第一个元素作为划分元素

[快速排序算法]

```
template <class T>
void QuickSoft(T a[ ], int p, int r)
{  if(p<r){
     int q=Partition(a, p, r)
     QuickSort(a, p, q-1); //对左半
     QuickSoft(a, q+1, r); //对右半段
```

[复杂性分析]

$$T_{max}(n)=\begin{cases} O(1) & n<=1 \\ T(n-1)+O(n) & n>1 \end{cases}$$

$$T_{min}(n)=\begin{cases} O(1) & n<=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

得: $T_{min}(n)=O(n\log n)$

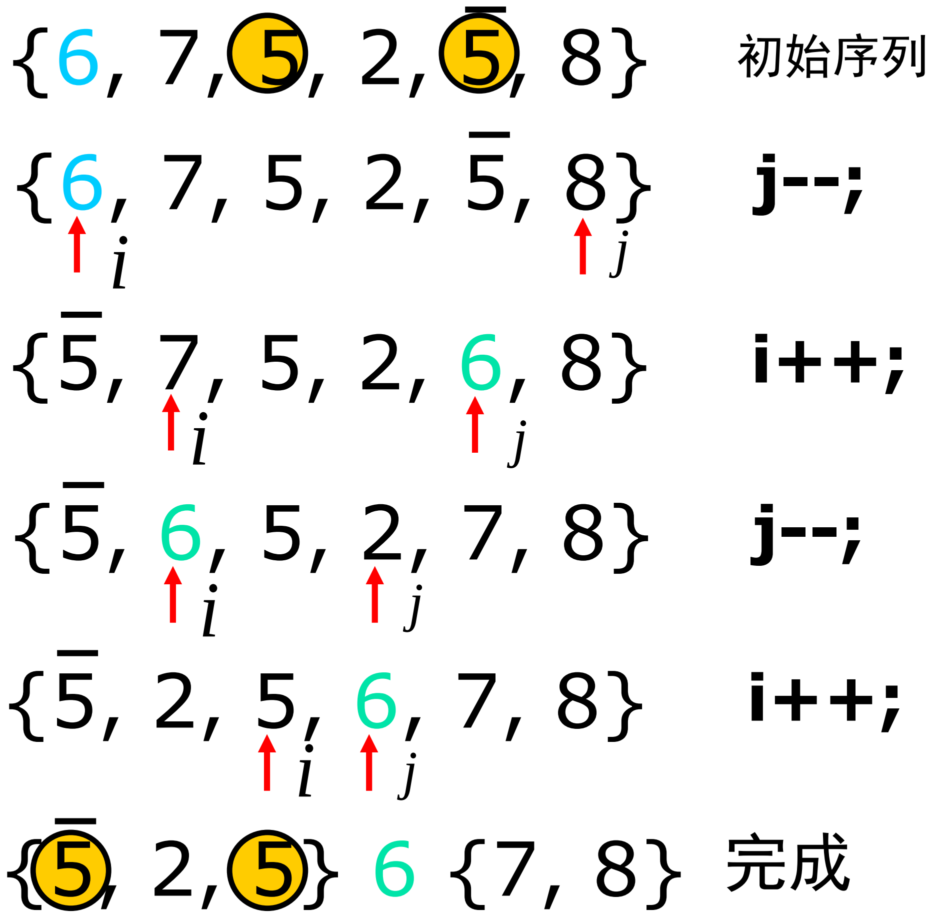
```
template<class T>
int Partion(T a[ ],int p, int r )
{  int i=p;  j=r+1;
     t  x=a[p]；  //取支点
     //将≥x的元素交换到左边
  //将≤x的元素交换到右边
     while (true) {
          while(a[++i] < x)；
          while(a[--j] > x)；
          if (i>=j ) break;
          swap(a[i],a[j]); }
     a[p] = a[j]；
     a[ j] = x;        } 设置支点
     return j  }
```

没有检查i是否越界递归调用的过程中，x为整个数组最大值，当前递归用的是p-r的元素，整个数组最大元素为n (n>r)，那么这会让i加到n

快速排序

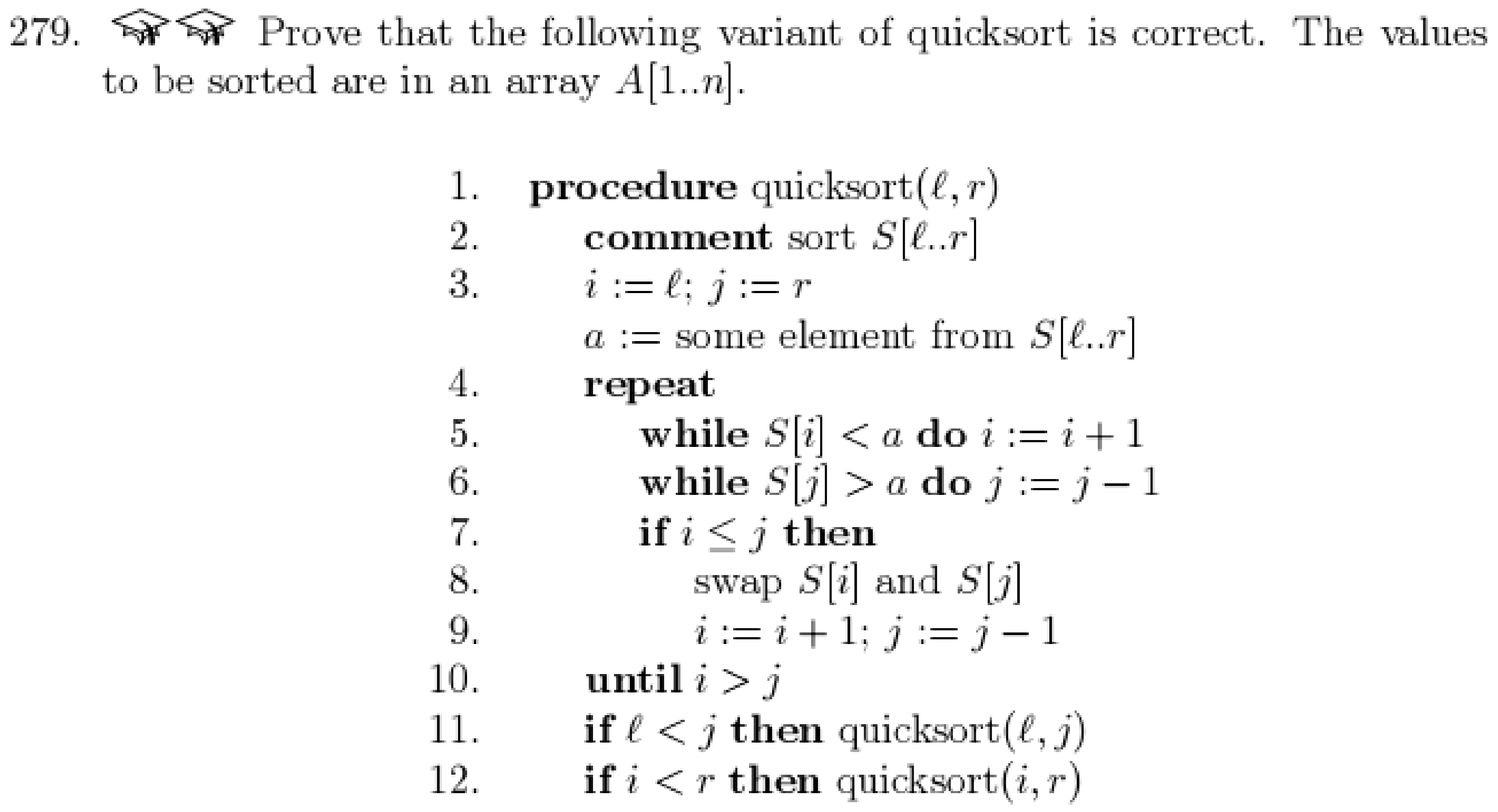
```
private static int partition (int p, int r)
  {
     int i = p,
        j = r + 1;
     Comparable x = a[p];
     // 将>= x的元素交换到左边区域
     // 将<= x的元素交换到右边区域
     while (true) {
       while (a[++i].compareTo(x) < 0);
       while (a[--j].compareTo(x) > 0);
       if (i >= j) break;
       MyMath.swap(a, i, j);
       }
     a[p] = a[j];
     a[j] = x;
     return j;
  }
```

如果 **x = a[p]**是最大值，结果如何？


{6, 7, 5, 2, $\bar{5}$, 8} 初始序列
{6, 7, 5, 2, $\bar{5}$, 8} j--;
i
j
{$\bar{5}$, 7, 5, 2, 6, 8} i++;
i
j
{$\bar{5}$, 6, 5, 2, 7, 8} j--;
i
j
{$\bar{5}$, 2, 5, 6, 7, 8} i++;
i
j
{$\bar{5}$, 2, 5} 6 {7, 8} 完成

快速排序具有**不稳定性**。

另一种 PARTITION 伪代码



279. Prove that the following variant of quicksort is correct. The values to be sorted are in an array $A[1..n]$.

```
procedure quicksort($\ell, r$)
    comment sort $S[\ell..r]$
    $i := \ell$; $j := r$
    $a :=$ some element from $S[\ell..r]$
    repeat
        while $S[i] < a$ do $i := i + 1$
        while $S[j] > a$ do $j := j - 1$
        if $i \leq j$ then
            swap $S[i]$ and $S[j]$
            $i := i + 1$; $j := j - 1$
    until $i > j$
    if $\ell < j$ then quicksort($\ell, j$)
    if $i < r$ then quicksort($i, r$)
```

Homework 7.1

- Page 87: 7.1-2, 7.1-3;

此处开始看
ipad笔记

7.2 Quicksort 算法性能分析

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning ;
- . If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort ;
- If the partitioning is unbalanced, it can run asymptotically as slowly as insertion sort.

Quicksort 的最坏情况:

- Worst-case partitioning :
  - The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements ;
  - The recurrence for the running time of this case is:

$$T(n)=T(n-1)+T(0)+\Theta(n)$$
$$=T(n-1)+\Theta(n)$$

$$\Rightarrow \quad T(n)=\Theta(n^2)$$

  - 由此可知， Quicksort在最坏情况的运行时间为： $\boldsymbol{\Omega(n^2)}$

Quicksort 最坏情况时间:

- Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n , We have the recurrence：

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + C_1 n$$

- We guess that $T(n) \le Cn^2$ for some constant C.
- Substituting this guess into above recurrence, we obtain:

$$T(n) \le \max_{0 \le q \le n-1} (Cq^2 + C(n-1-q)^2) + C_1 n$$

$$= C \cdot \max_{0 \le q \le n-1} (q^2 + (n-1-q)^2) + C_1 n$$

Quicksort 最坏情况时间(续)

- 由于$(q^2+(n-q)^2)$是q的二次函数，求导可得，在区间$[1..n]$范围内，该函数只可能在$q=1$, $q=n$, $q=n/4$等三个点处取极值，由此可知：

$$\max_{0\le q\le n-1}(q^2+(n-1-q)^2)\le n^2$$

- 所以有：

$$T(n)\le C(n-1)^2+C_1n=C\cdot n^2-2Cn+C_1n+C$$

- 这样，当取 $C>C_1$ 时，$T(n)\le Cn^2$ 对所有$n\ge1$成立。
- 因此，$T(n) = O(n^2)$。
- 由于QUICKSORT在最坏情况时的运行时间至少为$\Omega(n^2)$，综上所述，可知QUICKSORT在最坏情况时的运行时间为$\Theta(n^2)$。

Quicksort最好情况时间:

- Best-case partitioning
  - In the most even possible split, PARTITION produces two subproblems, one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$
  - In this case, The recurrence for the running time is

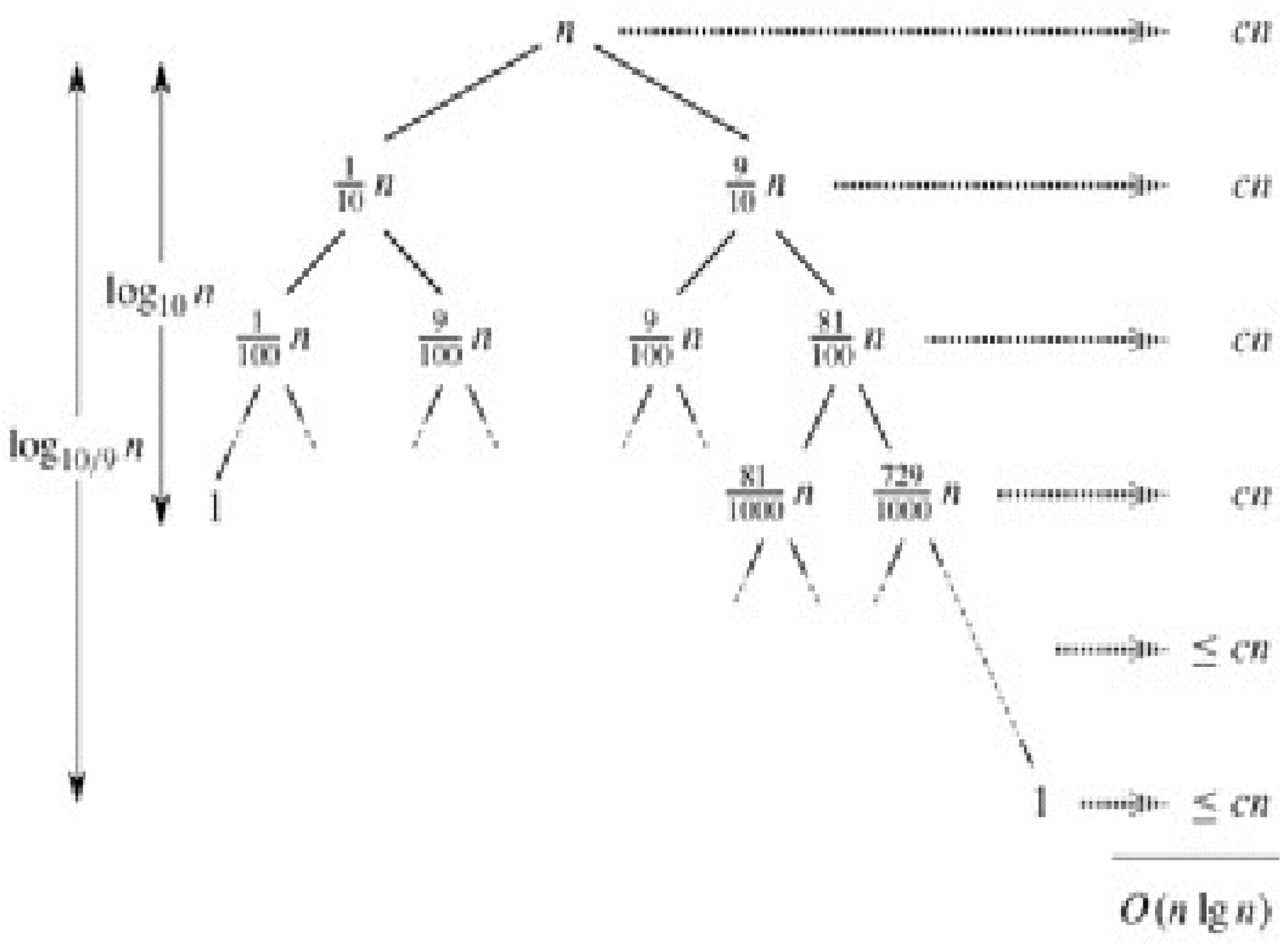
$$T(n) \le 2T(n/2) + \Theta(n)$$

$$T(n) = O(n \lg n).$$

  - The equal balancing of the two sides of the partition at every level of the recursion produces an asymptotically faster algorithm.
  - The average-case running time of quicksort is much closer to the best case than to the worst case ;

Example of Balanced Patition

- Suppose that the partitioning algorithm always produces a 9-to-1 proportional split ;
- The recursion terminates at depth $\log_{10/9} n = \Theta(\log n)$ and the cost at each level is $O(n)$, so the total cost of quicksort is $O(n \cdot \log n)$
- The recursion terminates at depth $\log_{10/9} n = \Theta(\log n)$ and the cost at each level is $O(n)$, so the total cost of quicksort is $O(n \cdot \log n)$ 。
- The following figure shows the recursion tree for this recurrence



n
cn
$\frac{1}{10}n$
$\frac{9}{10}n$
cn
$\log_{10} n$
$\frac{1}{100}n$
$\frac{9}{100}n$
$\frac{9}{100}n$
$\frac{81}{100}n$
cn
$\log_{10/9} n$
1
$\frac{81}{1000}n$
$\frac{729}{1000}n$
cn
$\leq cn$
1
$\leq cn$
$O(n \lg n)$

Quicksort 平均时间:

- 设 $T(n)$ 为输入规模为n 时 QUICKSORT 算法的平均运行时间， $T_k(n)$为所选划分元序号为 k+1时 QUICKSORT 算法的平均运行时间，则$T(n)$ 满足以下递归方程：

$$T_k(n) = \frac{1}{n}\sum_{k=0}^{n-1}(T(k)+T(n-k-1)+cn)$$

$$T(n) = \sum_{k=0}^{n-1} p(k+1)T_k(n) = \sum_{k=0}^{n-1}\frac{1}{n}\cdot T_k(n)$$

$$= \frac{1}{n}\sum_{k=0}^{n-1}(T(k)+T(n-k-1)+cn)$$

$$T(n)=\frac{1}{n}\Big(\sum_{k=0}^{n-1}(T(k))+\Big(\sum_{k=0}^{n-1}T(n-k-1)\Big)\Big)+cn=\frac{2}{n}\sum_{k=0}^{n-1}T(k)+cn$$

- 解递归方程可得：

$$n\square T(n)=2\sum_{k=0}^{n-1}T(k)+cn^2$$

$$(n-1)\square T(n-1)=2\sum_{k=0}^{n-2}T(k)+c(n-1)^2$$

- 两式相减，可得：

$$nT(n)-(n-1)T(n-1)=2T(n-1)+c(2n-1)$$

$$\frac{T(n)}{n+1}\le\frac{T(n-1)}{n}+\frac{2c}{n}$$

- 令 $G(n)=T(n)/(n+1)$, 可得:

$$G(n) \le G(n-1) + 2c/n = G(n-2) + 2c(\frac{1}{n-1} + \frac{1}{n})$$

$$= G(n-3) + 2c(\frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}) = \cdots\cdots$$

$$= G(n-k) + 2c(\frac{1}{n-k+1} + \cdots + \frac{1}{n-1} + \frac{1}{n})$$

$$= G(1) + 2c\sum_{k=0}^{n-2} \frac{1}{n-k} = 2c\sum_{k=2}^{n} \frac{1}{k} \le 2c \cdot H_n \le 2c\log n$$

(参见 P. 1066　公式 A 10)

- 所以，Quicksort 算法的平均时间复杂度为:

$$T(n) = G(n)(n+1) = \Theta(n\log n)$$

Homework 7.2

- Page 90: 7.2-3， 7.2-4;
- Page 93: 7.4-1 , 7.4-3;

7.3 Randomized Quicksort

- Instead of always using $A[r]$ as the pivot, we will use a randomly chosen element from the subarray $A[p \cdot\cdot r]$;
- In randomized quicksort, using a different randomization technique, called ***random sampling*** ;
- For large enough inputs, the randomized version of quicksort can obtain good average-case performance over all inputs ;

Randomized Quicksort 算法

```
RANDOMIZED-PARTITION ($A$, $p$, $r$)
  1  $i \leftarrow$ RANDOM ($p$, $r$)
  2  $A[r] \leftrightarrow A[i]$
  3  return PARTITION($A$, $p$, $r$)

RANDOMIZED-QUICKSORT ($A$, $p$, $r$)
 1 if $p < r$
 2      then $q \leftarrow$ RANDOMIZED-PARTITION($A$, $p$, $r$)
 3           RANDOMIZED-QUICKSORT($A$, $p$, $q - 1$)
 4           RANDOMIZED-QUICKSORT($A$, $q + 1$, $r$)
```

7.4 Quicksort vs Heapsort:

- 在不同的计算模型下， Quicksort 和 Heapsort 的性能会有所不同。
- 下面我们引用UCSD的 Larry Carter 介绍的一个例子。

CSE 202 - Algorithms

Quicksort vs Heapsort:

the "inside" story

or

A Two-Level Model of Memory

Where are we

- Traditional (RAM-model) analysis: Heapsort is better
  - Heapsort worst-case complexity is $\Theta(n \log n)$
  - Quicksort worst-case complexity is $O(n^2)$.
    - average-case complexity should be ignored.
    - probabilistic analysis of randomized version is $\Theta(n \log n)$
- Yet Quicksort is popular.
- Goal: a better model of computation.
  - It should reflect the real-world costs better.
  - Yet should be simple enough to perform asymptotic analysis.

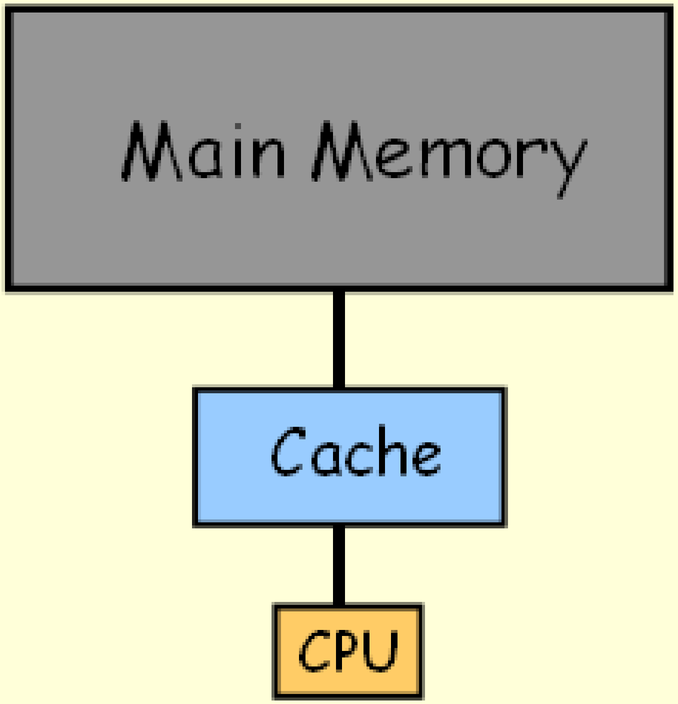
2-level memory hierarchy model (MH_2)

Data moves in "blocks" from Main Memory to cache.

- A block is b contiguous items.
- It takes time b to move a block into cache.
- Cache can hold only b blocks.
  - Least recently used block is evicted.

Individual items are moved from Cache to CPU.

- Takes 1 unit of time.



Note - "b" affects:

1. block size
2. cache capacity (b^2)
3. transfer time

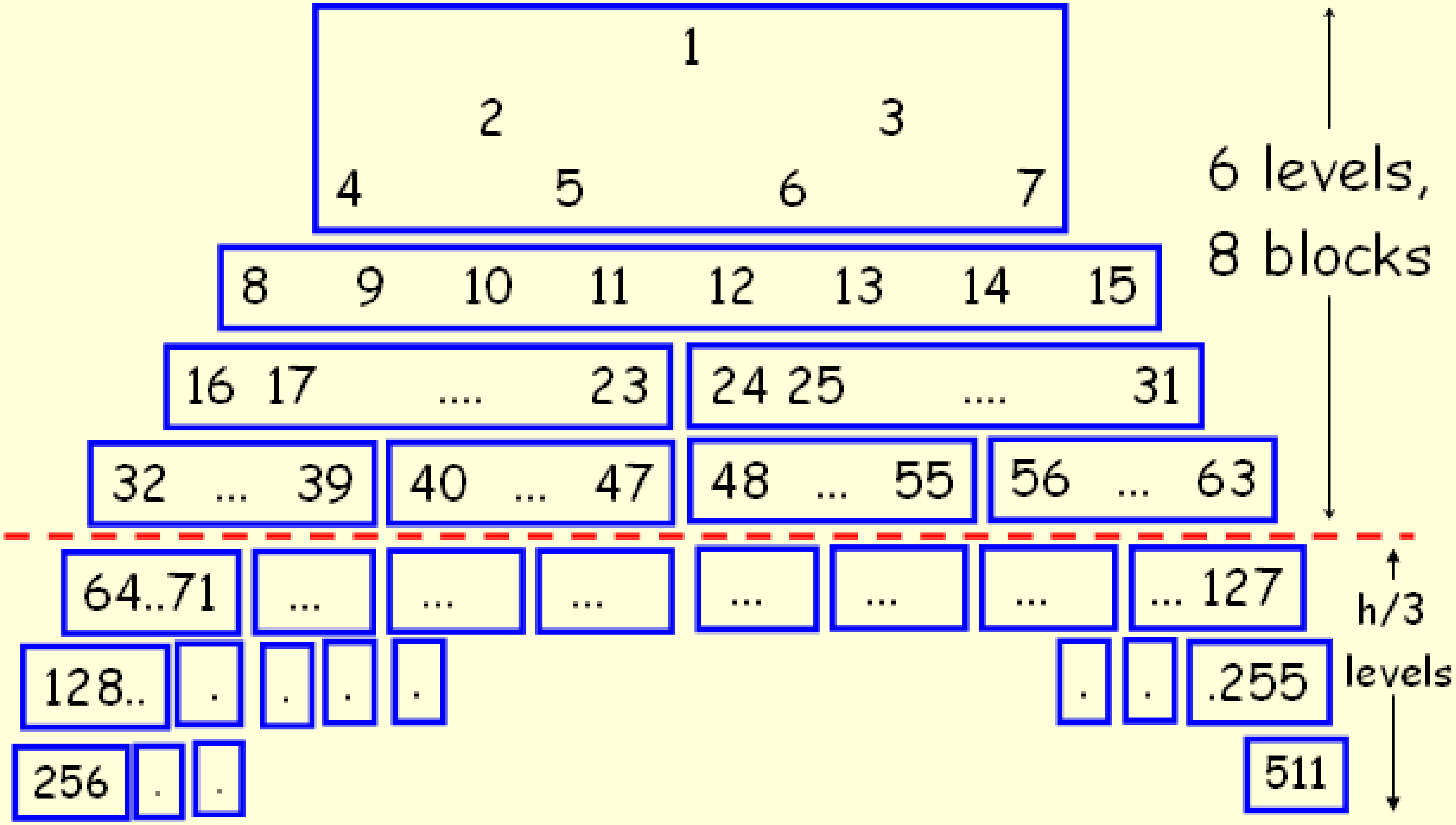
2-level memory hierarchy model (MH_2)

For asymptotic analysis, we want b to grow with n

$b = n^{1/3}$ or $n^{1/4}$ are plausible choices

	block size = b (Bytes)	cache size = b^2 (Bytes)	transfer (cycles)	memory = n (Bytes)
Memory = DRAM Cache = SRAM	2^6 - 2^8	2^{13} - 2^{20}	2^5 - 2^7	2^{26} - 2^{30}
$b = n^{1/4}$	2^7	2^{14}	2^7	2^{28}
Memory = disk Cache = Dram	2^{12} - 2^{13}	2^{26} - 2^{30}	2^{15} - 2^{20}	2^{33} - 2^{38}
$b=n^{1/3}$	2^{13}	2^{26}	2^{13}	2^{39}

Cache lines of heap (b=8, n=511, h=9)



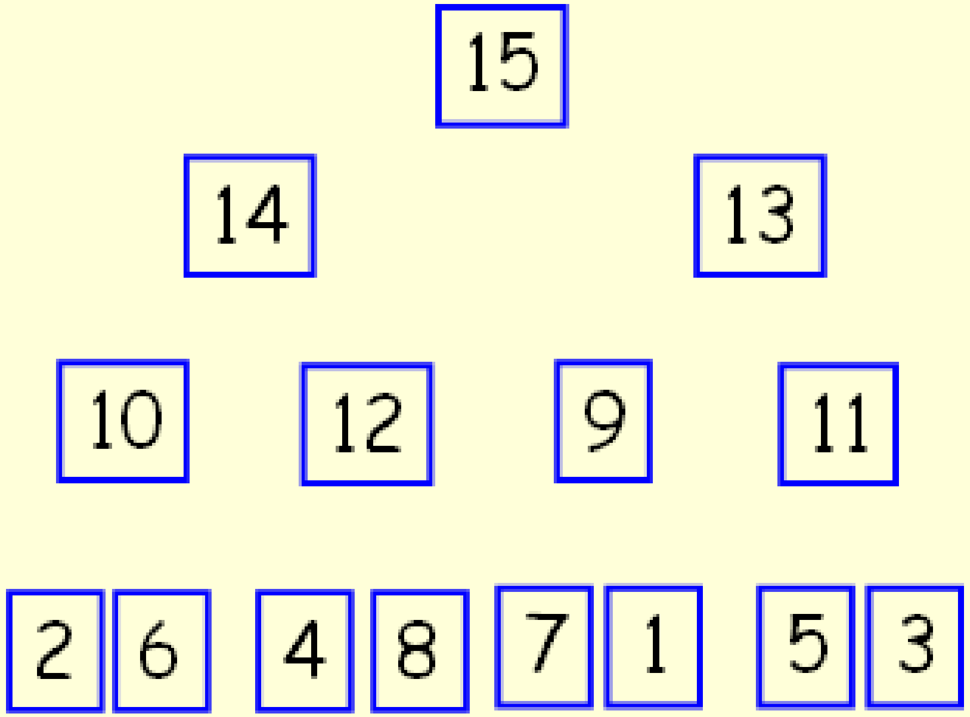
A worst-case Heapsort instance

Each Extract-Max goes all the way to a leaf.

Visits to each node alternate between left and right child.

Actually, for any sequence of paths from root to leaves, one can create example.

Construct starting with 1-node heap



MH_2 analysis of Heapsort

- Assume $b = n^{1/3}$.
  - Similar analysis works for $b = n^a$, $0 < a < \frac{1}{2}$.
- Effect of LRU replacement:
  - First $n^{2/3}$ heap elements will "usually" be in cache.
    - Let $h = \lfloor \log n \rfloor$ be height of the tree.
    - These elements are all in top $\lceil (2/3)h \rceil$ of tree.
  - Remaining elements won't usually be in cache.
    - In worst case example, they will *never* be in cache when you need them.
    - *Intuition:* Earlier blocks of heap are more likely to be references than a later one. When we kick out an early block to bring in a later one, we increase misses later.

MH_2 analysis of Heapsort (worst-case)

- Every access below level $\lceil (2/3)h \rceil$ is a miss.
- Each of the first n/2 Extract-max's "bubbles down" to the leaves.
  - So each has at least (h/3)-1 misses.
  - Each miss takes time b.
- Thus, $T(n) > (n/2)\ ((h/3)-1)\ b$.
  - Recall: $b = n^{1/3}$ and $h = \lfloor \log n \rfloor$.
- Thus, T(n) is $\Theta(n^{4/3} \log n)$.
- And obviously, T(n) is $O(n^{4/3} \log n)$.
  - Each of c n log n accesses takes time at most $b = n^{1/3}$.
    (where c is constant from RAM analysis of Heapsort).

Quicksort MH_2 complexity

- Accesses in Quicksort are sequential
  - Sometimes increasing, sometimes decreasing
- When you bring in a block of b elements, you access every element.
  - Not 100% true, but I'll wave my hands
- We take b time to get block for b accesses
- Thus, time in MH_2 model is same as RAM.
  - $\Theta(n \lg n)$

Bottom Line: MH2 analysis shows Quicksort has lower complexity than Heapsort!