真·超浓缩DSA整合攻略（期中篇）

Time complexity

`Comparison-based Sorting Algorithm`

Algorithm	worst-case	average-case	best-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	linear function of n
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort (general)	$\Theta(n \cdot logn)$	$\Theta(n \cdot logn)$	/
Merge	/	$\Theta(n)$	/
[Heap Sort (general)](	$O(n \cdot logn)$/ $\Theta(n \cdot logn)$	/	/
buildMaxHeap	/	bottum-up:$O(n)$ one-by-one: $O(nlogn)$	/
MaxHeapify	/	$O(logn)$	/
Priority Queue	insert $O(logn)$	remove $O(logn)$	maximum $\Theta(1)$
Quick Sort	$\Theta(n^2)$	Balanced/expected Case $O(nlogn)$	$\Theta(n \cdot logn)$

> **Notice that the time complexity (in general) of insertion sort is in $O(n^2)$ but not in $\Theta(n^2)$**

`Linear Sorting Algorithm`

Algorithm	worst-case	average-case
Counting Sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix Sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$

Stability and in-place

Algorithm	stable	in-place
Insertion Sort	$\odot$	$\odot$
Merge Sort	$\odot$	$\otimes$

Heap Sort|$\otimes$|$\odot$
Quick Sort|mostly unstable|$\odot$
Counting Sort|$\odot$|$\otimes$
Radix Sort|$\odot$ |$\otimes$
Bucket Sort|$\odot$ depend on the underlying sort|$\otimes$

Some Definitions

Bounds

Θ asymptotic tight bound('$=$')
O asymptotic upper bound('$\leq$')
Ω asymptotic lower bound('$\geq$')
o asymptotic upper bound not tight('$<$')
ω asymptotic lower bound not tight('$>$')

> **Asymptotic tight bound: Θ**

for $f, g : \mathbb{N} \to \mathbb{R}^+ : f(n) \in \Theta(g(n))$ if
$\exists c, c' > 0 \in \mathbb{R} \exists n_0 > 0 \in \mathbb{N} \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow c \cdot g(n) \leq f(n) \leq c' \cdot g(n)$

$f(n)$ is eventually '**as large as** $g(n)$' up to constants

We write $f(n) \in \Theta(g(n))$ or also $f(n) = \Theta(g(n))$

Notice that ($\Theta(g(n))$ is a set of functions)

> **Asymptotic upper bound: O**

for $f, g : \mathbb{N} \to \mathbb{R}^+ : f(n) \in O(g(n))$ if
$\exists c > 0 \in \mathbb{R} \exists n_0 > 0 \in \mathbb{N} \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)$

$f(n)$ is eventually '**smaller-equal** $g(n)$' up to constants

> **Properties**

- $f(n) \in \Theta(g(n))$ if
  $f(n) \in O(g(n)) \, and \, g(n) \in O(f(n))$

This is the same as:
$a = b$ if $a \leq b$ and $b \leq a$

- Big-O properties:

`large degrees beat lower degrees`
$n^a \in O(n^b)$ for all $0 < a \leq b$

e.g. $n^2 \in O(n^3)$

`polynomials beat log`
$log_a(n) \in O(n^b)$ for all $a, b > 0$

e.g. $logn \in O(\sqrt{n})$

`exponentials beat polynomials`
$n^a \in O(b^n)$ for all $a > 0$ and $b > 1$

e.g. $n^5 \in O(2^n)$

`logs euqally large`
$log_a n \in O(log_b n)$ for all $a, b > 0$

e.g. $log_2 n \in O(log_3 n)$

This is because

$a^{log_a b \cdot log_b n} = (a^{log_a b})^{log_b n} = b^{log_b n} = n$

hence we have: $log_a b \cdot log_b n = log_a n$

Tree

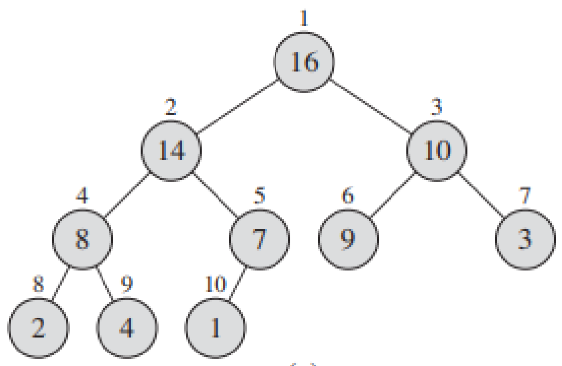
- Set of nodes with a **parent-child** relation
- A unique distinguished node **root**
- every non-root node has a **unique** ancestor/ predecessor / parent
- a node may have successors / descendants / children
- a node without successors is a **leaf** or **external** (node without children)
- a node with successors is **internal** (node with children)
- the depth of a node is the length of the path to the root (**depth: 该node到root的距离**)
- the height of a node is the maximal length of a path to a leaf (height: **该node到最远leaf的距离**)
- **the height of a tree = the height of the root = maximal depth**
- a **level** or **layer** of a tree consists of all **nodes** of the **same depth**
- number of levels = height of tree + 1

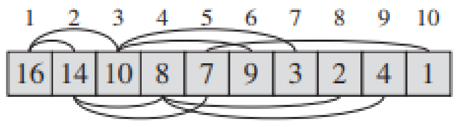
> **binary tree**

- every node has 0, 1 or 2 (ordered) successors
- binary tree is complete if all levels are completely filled
- binary tree is almost or nearly complete if all levels are completely filled except possibly the lowest one which is filled left-to-right

complete $\Rightarrow$ almost complete $\Rightarrow$ normal binary

`an almost complete binary tree corresponds naturally to an array`


(a)


(b)

- let the height of an almost complete binary tree is h
  `if the lowest level contains one element`
  number of element $n = 1 + 2 + 3 + ... + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$
  `if the lowest level is full`
  number of element $n = 1 + 2 + 3 + ... + 2^h = 2^{h+1} - 1$

so $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$

so $h \leq logn \leq h + 1$

so $h = \lfloor logn \rfloor$

> **parent-children relation in the array**

- i an index in the array

`已知child的index，返回parent的index`
$Algorithm\ parent(i)$
$return\ \lfloor i/2 \rfloor$

`已知parent的index， 返回左边child的index`
$Algorithm\ left(i)$
$return\ 2i$

`已知parent的index， 返回右边child的index`
$Algorithm\ left(i)$
$return\ 2i + 1$

> **Decision Tree**

Heap

- A heap is an almost complete binary tree
- all levels from as full as possible
- if we walk downwards then keys decrease
-

> max-heap

- an almost complete binary tree
- every node is labeled with a key/ label from a totally ordered set
- on every path from the root to a leaf the labels / keys are non-increasing (根最大，越往下越小)

$H[parent(i)] \geq H[i]$

- 如果一个heap所有的node有相同的值，这也是一个max-heap

Pseudocode

Insertion Sort

```
//A: Input Array
//n: Array Size

Alogorithm insertionSort(A,n):
  for j:= 2 to n do
    key := A[j]
    i := j - 1
    while i > 0 and A[i] > key do
      A[i+1] := A[i]
      i := i - 1
    A[i+1] := key
```

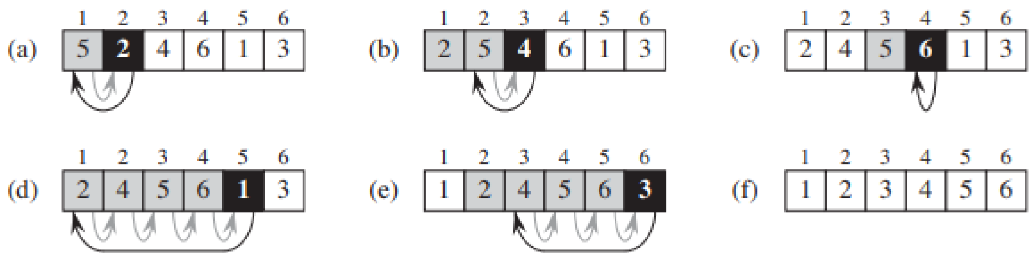


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Merge Sort

```
//A: Array
//p: The start index of the array
//r: The end index of the array

Algorithm mergeSort(A, p, r):
  if p < r then
    q := floor((p + r)/2) // take the floor
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    Merge(A, p, q, r)
```

```
MERGE(A,p,q,r)
  n1 = q - p + 1 //lhs array size
  n2 = r - q //rhs array size
  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays //array size is 1 place bigger
than n1 and n2
  for i = 1 to n1
    L[i] = A[p + i - 1] //copy lhs into L
  for j = 1 to n2
    R[j] = A[q + j] //copy rhs into R
  L[n1 + 1] = infinity //last element be infinity
  R[n2 + 1] = infinity
  i = 1 //reset i and j
  j = 1
  for k = p to r // actual merge starts from here
    if L[i]<= R[j]
      A[k] = L[i]
      i = i + 1
    else A[k] = R[j]
      j = j + 1
```

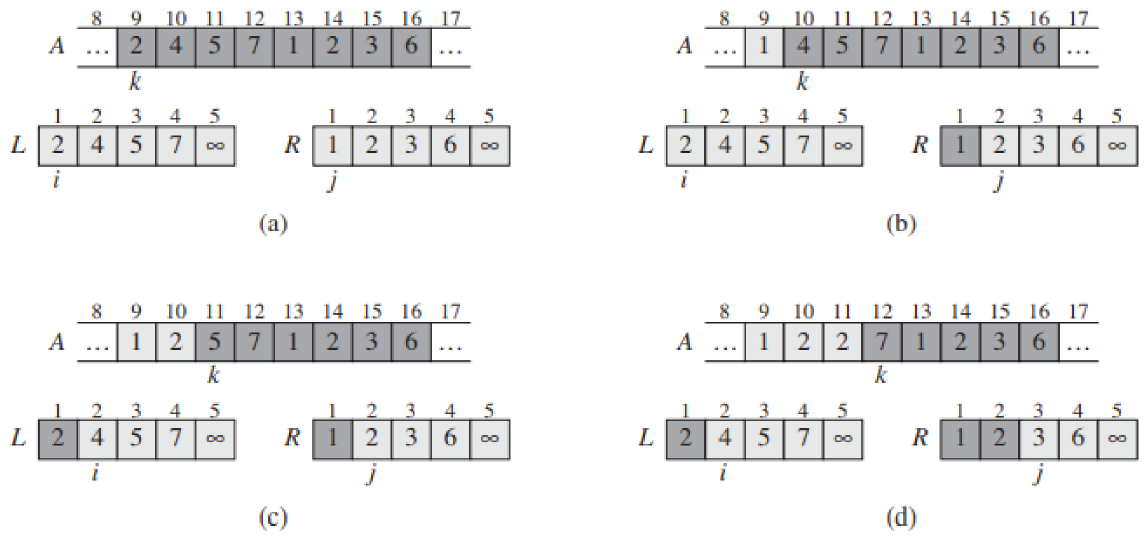
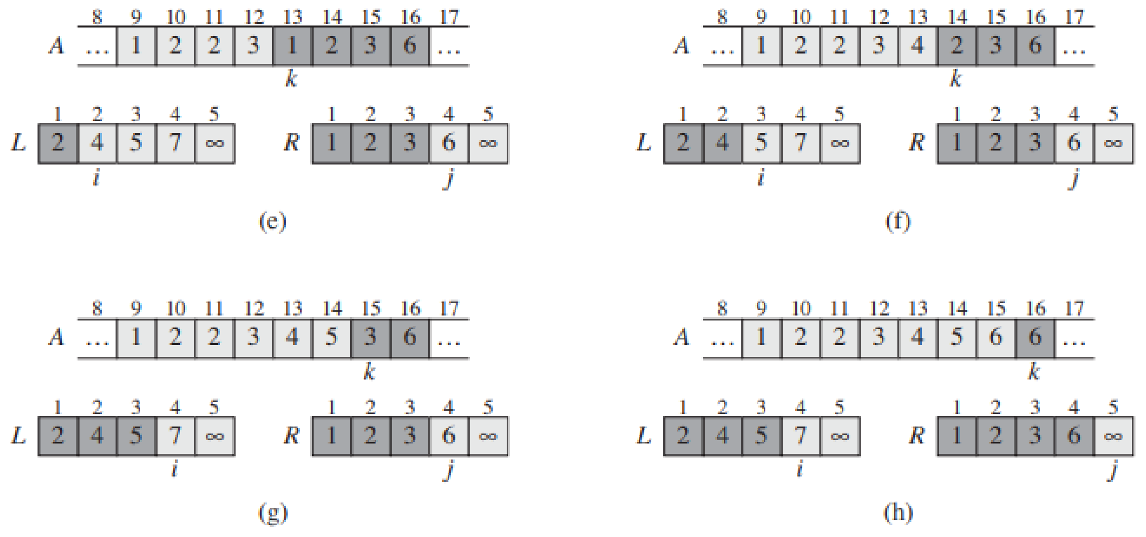


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.



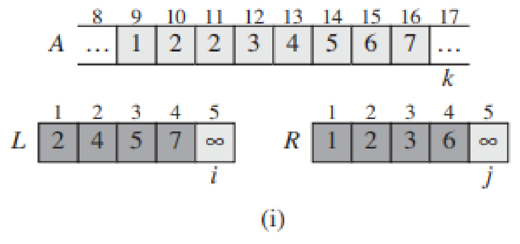


Figure 2.3, continued **(i)** The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Heap Sort

`MaxHeapify`

```
//Maintaining the heap property
//We reconstruct the max-heap property using a down-heap bubble

Algorithm MaxHeapify(A,i):
  l := left(i) //2i return index instead of value
  r := right(i) //2i+1
  if l <= A.heap_size and A[l] > A[i] then //l <= A.heap_size is to make sure l
is still an index of A
    largest := l
  else
    largest := i

//Comparison part
  if r <= A.heap_size and A[r] > A[largest] then
    largest := r

//swap part
  if largest != i then
    swap(A[i], A[largest])
    MaxHeapify(A, largest)
```

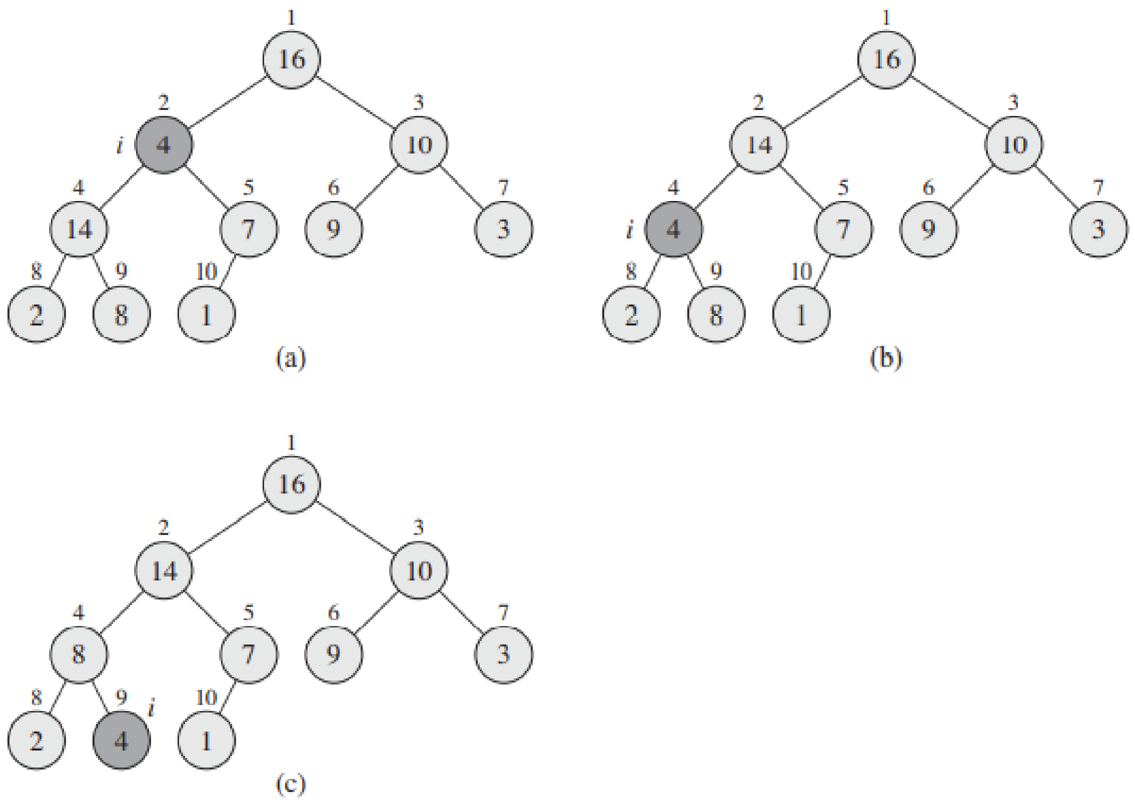


Figure 6.2 The action of Max-Heapify$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call Max-Heapify$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call Max-Heapify$(A, 9)$ yields no further change to the data structure.

`buildMaxHeap`

```
Algorithm buildMaxHeap(H):
  H.heap_size := H.length
  for i = floor(H.length/2 ) downto 1 do //index great than H.leangth/2 are leaves(trivial max heap)
    MaxHeapify(H,i)
```

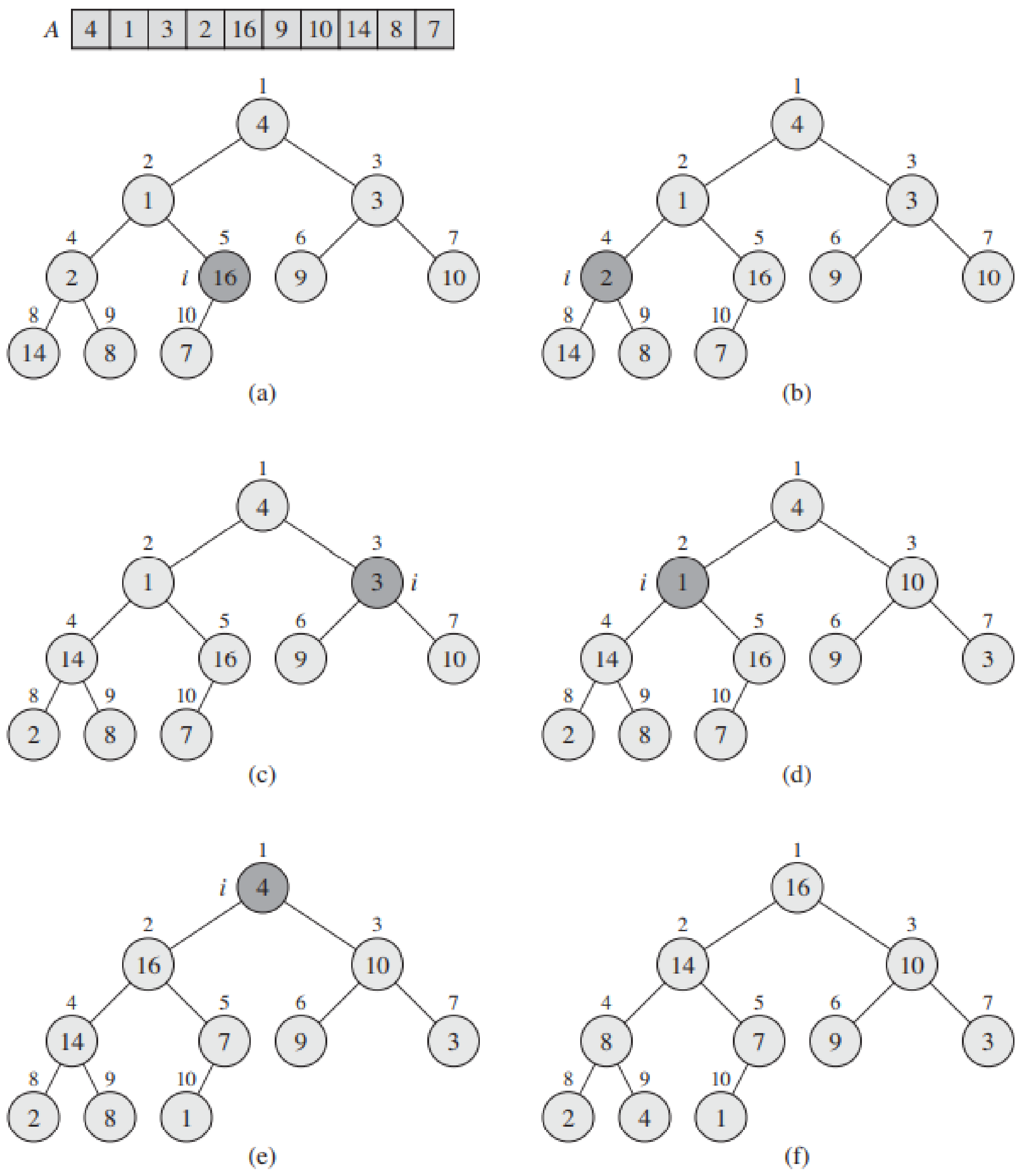


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

`heapsort`

```
Algorithm heapsort(H):
  buildMaxHeap(H)
  for i = H.lenth downto 2 do
    swap H[1] and H[i]
    H.heap_size := H.heap_size - 1
    MaxHeapify(H,1)
```

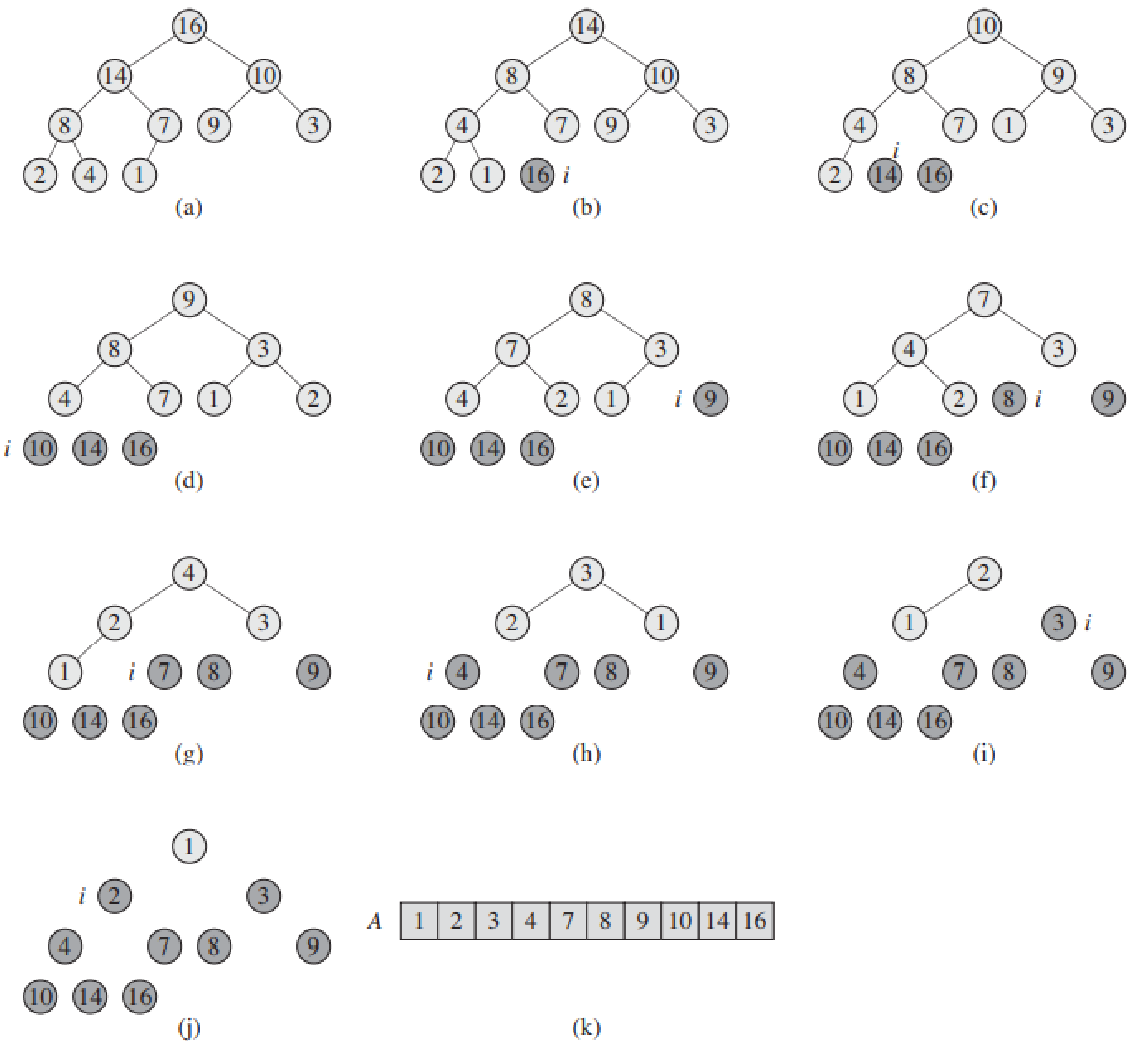


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Priority Queue

`maximum`

```
//H is a max-heap; return the max key
//does not change the heap, both H.size and H.heapsize do not change
//Theta(1)

Algorithm heapMaximum(H):
  return H[1]
```

`remove`

```
//H -> max-heap
//remove and return maximum;error omitted
//H.size does not change; H.heapsize decreases
//O(logn)

Algorithm heapExtractMax(H):
  max := H[1]
  H[1] := H[H.heap_size] //swap the first and the last
  H.heap_size := H.heap_size - 1
  maxHeapify(H,1)   //O(logn)
  return max
```

`insert`

```
//H.size stays the same; H.heapsize increases
//O(logn)

Algorithm heapInsert(H,k):
  H.heap_size := H.heap_size + 1
  H[H.heap_size] := -infinity
  HeapIncreaseKey(H,H.heap_size,k)

Algorithm heapIncreaseKey(H,i,k)
  if k < H[i] then
    return error
  H[i] := k
  while i > 1 and H[parent(i)] < H[i] do
    swap(H[parent(i)]), H[i])
    i := parent(i) // floor(i/2)
```

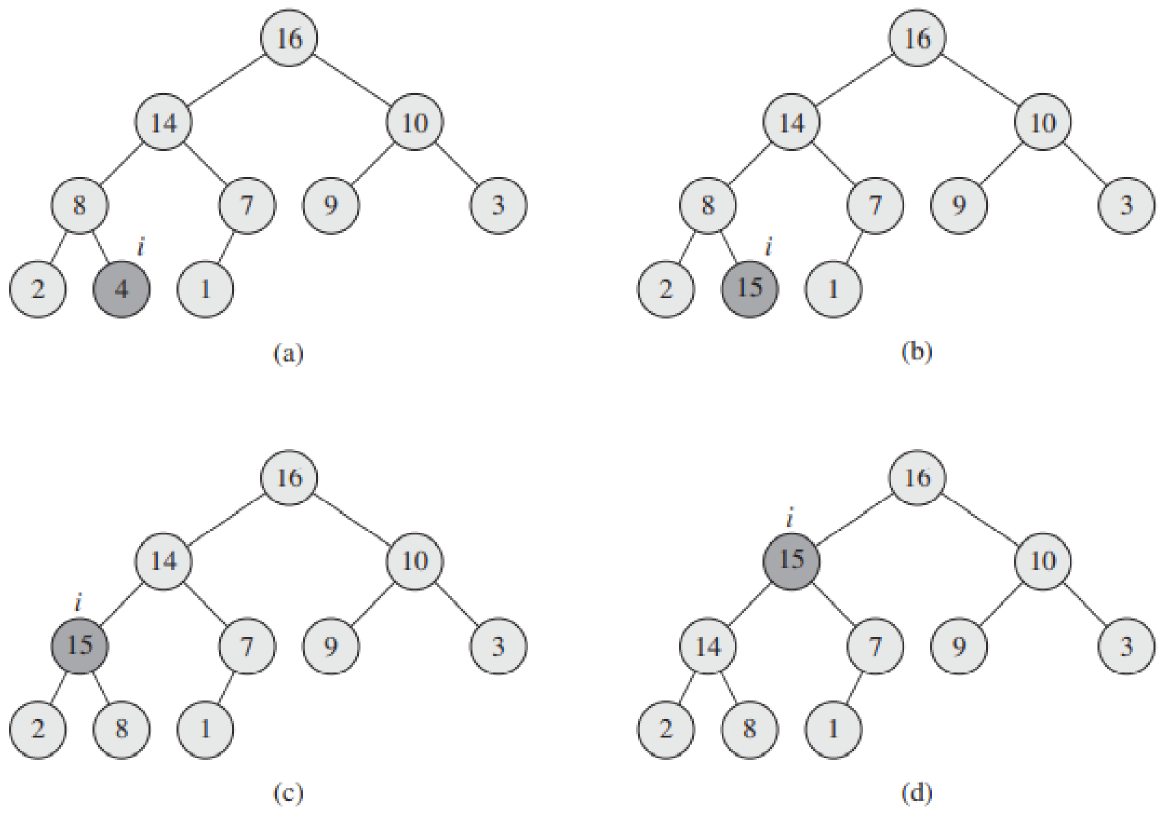


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

Quick Sort

`partition`

```
//A: Array
//p: first index
//r: last index

Algorithm partition(A,p,r)
  x := A[r]
  i := p - 1
  for j = p to r - 1 do
    if A[j] <= x then
      i := i + 1
      exchange A[i] with A[j]
  exchange A[i+1] with A[r]
  return i + 1
```

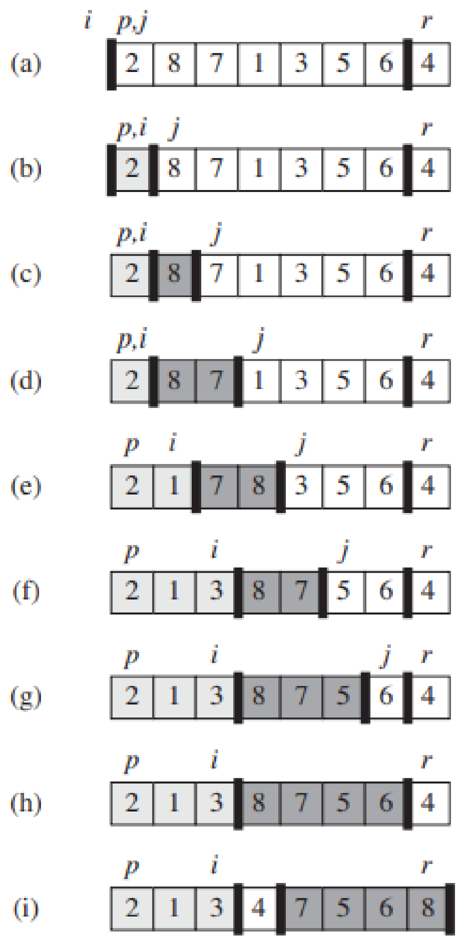


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

`quickSort`

```
//Initial : (A, 1, A.length)

Algorithm quickSort(A,p,r)
  if p < r then
    q := partition(A,p,r) //index of pivot after partition done
    quickSort(A,p,q - 1)
    quickSort(A, q + 1, r)

//recursion stops if p = r
```

Counting Sort

```
//A: Input Array
//B: Output Array
//range from 0 up to k

Algorithm countingSort(A,B,k)
  new array C[0...k] //take space
  //init C
  for i:= 0 to k do //init C
    C[i] := 0

  //count occurence in A
  for j := 1 to A.length do
    C[A[j]] := C[A[j]] + 1

  //count "up to" occurence
  for i := 1 to k do
    C[i] := C[i] + C[i - 1]

  //genarate B
  for j := A.length downto 1 do
    B[C[A[j]]] := A[j]
    C[A[j]] := C[A[j]] - 1
```

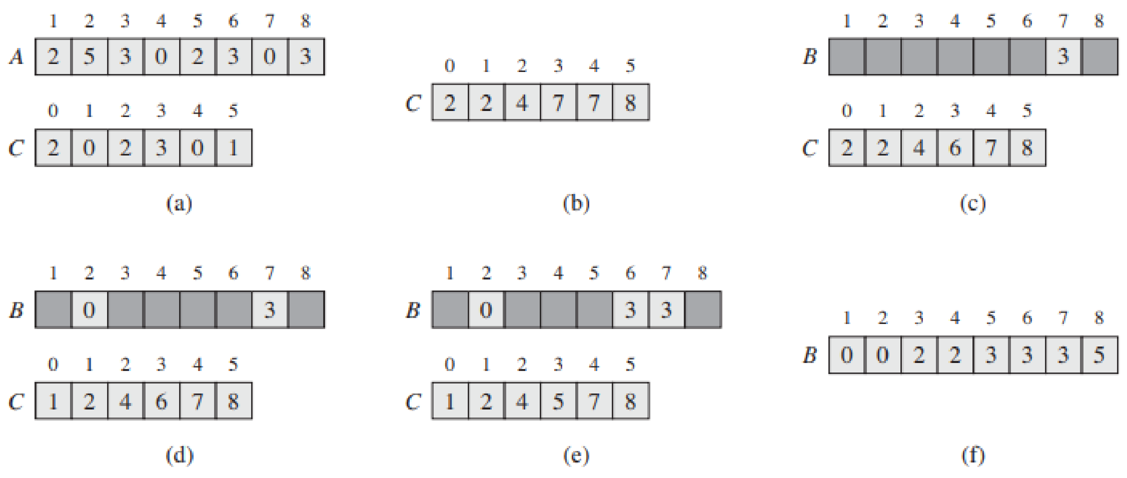


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1\,..\,8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Radix Sort

```
//A: Input Array
//d: dimension

Algorithm radixSort(A,d)
  for i := 1 to d do
  use some stable sort on digit d
```

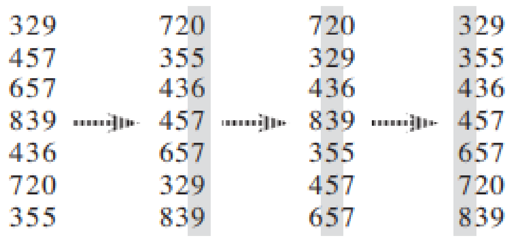


Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Bucket Sort

```
//A: Input Array with 0<=A[i] <= 1 for i = 1,...,A.length

Algorithm bucketSort(A):
  n := A.length
  new array B[0...n-1]
  for i := 0 to n-1 do
    make B[i] an empty list //init B
  for i := 1 to n do
    insert A[i] into list B[floor(n·A[i])]
  for i := 0 to n-1 do
    insertionSort(B[i])
  concatenate B[0],B[1],...,B[n-1]
```

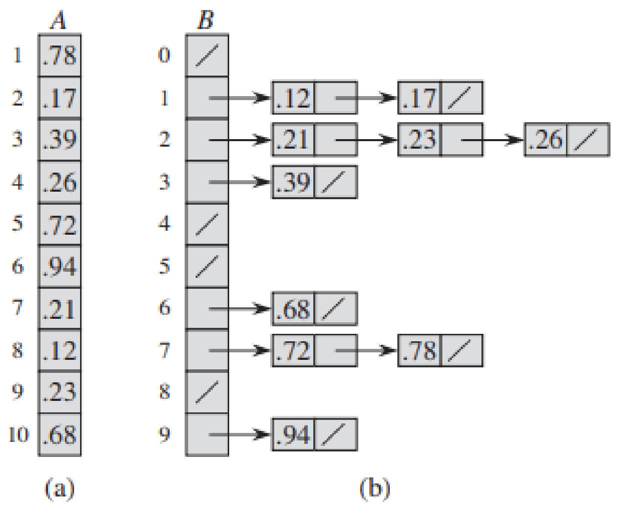


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Time complexity in detail

Inserstion Sort

> Worst Case

- we excute the whole-loop as many times as possible($A[i] > key$ always succeeds)
- The input is **inverse sorted**

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

```
INSERTION-SORT(A)                                  cost     times
1  for $j = 2$ to $A.length$                        $c_1$    $n$
2      $key = A[j]$                                 $c_2$    $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1\,..\,j-1]$.                $0$      $n-1$
4      $i = j - 1$                                  $c_4$    $n-1$
5      while $i > 0$ and $A[i] > key$               $c_5$    $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                          $c_6$    $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i - 1$                              $c_7$    $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                               $c_8$    $n-1$
```

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Best Case

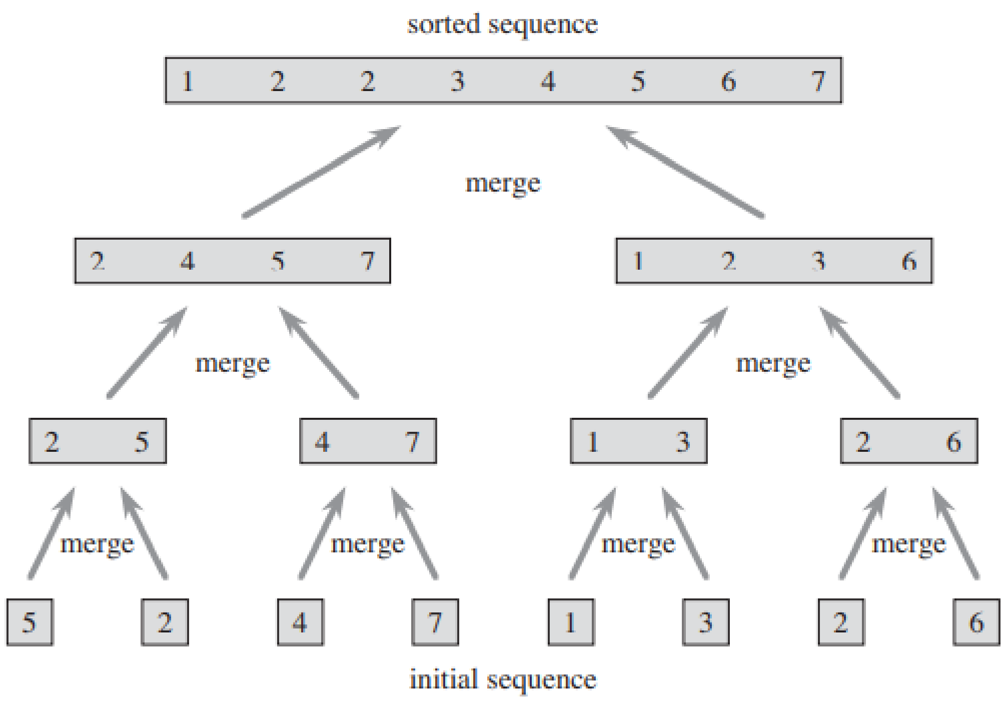
- the best case occurs if $A[i] > key$ always fails
- this is the case if the input is **sorted**

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

Merge Sort

`recursion tree`



- Time complexity of merging n elements is in $\Theta(n)$
- height is $logn$ for n is the length of the input-array
- number of levels is $(logn) + 1$
- number of leaves is n
- total: $c \cdot n \cdot logn$(from the upper n level) $+d(n)$(from the leaves)
- So the $T(n)$ is in the form of $anlogn + bn$ which is $\Theta(nlogn)$ (see recurrence)

`For small inputs, insertion sort may be faster because of the constant`

Heap Sort

MaxHeapify

- Time complexity of down-heap bubble determined by **height of the heap**
- so in $Olog(n)$

1. with h the height of the heap:
   $T(h) = T(h-1) + 1$ if $h > 0$ gives $T(h) \in O(h)$
   then use $h \in \Theta(logn)$ gives $T(n) \in O(logn)$
2. with n the number of nodes of the heap:
   $T(n) = T(\frac{2}{3}n) + 1$ if $n > 1$ gives $T(n) \in O(logn)$

`Because in the worst case the bottom level is exactly half full`

buildMaxHeap

- build MaxHeap is in $O(n)$

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by

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

We evalaute the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

> **heapsort**

- buildMaxHeap in $O(n)$
- $n-1$ calls of MaxHeapify with every call in $O(logn)$
- hence the worst-case running time of heapsort is in $O(n \cdot logn)$
- worst case happens if the list is already **sorted**

Quick Sort

> **partition**

```
Algorithm partition(A,p,r)
  x := A[r]
  i := p - 1
  for j = p to r - 1 do
    if A[j] <= x then
      i := i + 1
      exchange A[i] with A[j]
  exchange A[i+1] with A[r]
  return i + 1
```

- We do the test for the for-loop (line 4) $r - p + 1$ times
- We execute the inside of the for-loop $r - p$ times
- Worst-case running time proportional to $r - p$
- Linear in number of elements:
  on input $A[p...r]$ in $\Theta(n)$ with $n = r - p + 1$

> **Worst Case**

- Worst-case running time if no 'small ones' or no 'big ones'

$T(n) = T(0) + T(n-1) + \Theta(n)$ with $T(0) \in \Theta(1)$ (see recurrence)

we find: $T(n) \in \Theta(n^2)$

> **Best Case**

- Best-case running time if as many 'small ones' as 'big ones'

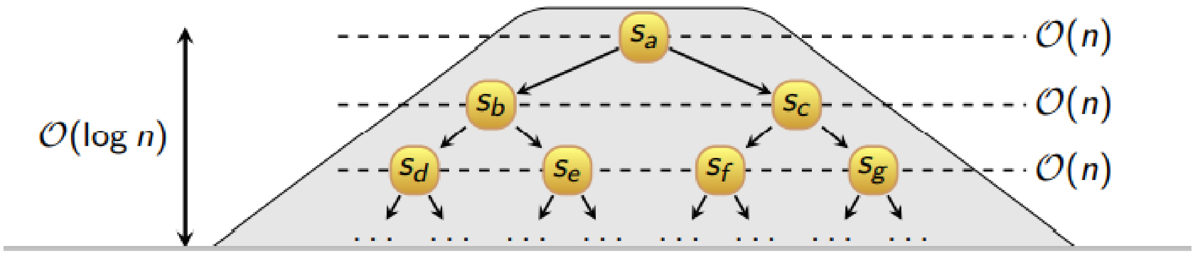
$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$ with $T(0) \in \Theta(1)$

$2 \cdot T(\frac{n}{2})$ is from the recursive calls and $\Theta(n)$ from the partition

we find: $T(n) \in \Theta(n \cdot logn)$ (see recurrence)

> **Average Case**

- consider 'good' splits (we get $\frac{1}{4}$ and $\frac{3}{4}$ or better) and 'bad' splits'
- suppose probability of 'good' split is $\frac{1}{2}$
- expectation for node in recursion tree on depth i:
  $\frac{i}{2}$ ancestors are calls with good pivots
- hence: length array on depth i is $\leq (\frac{3}{4})^{\frac{i}{2}} \cdot n$
  at least half cases are good
- length 1 is reached for $i = 2log_{\frac{4}{3}}n$
- hence: height is in $O(logn)$
- work per depth: in $\Theta(n)$



`with random key as pivot from input sequence`

Average time complexity of quicksort is in $O(nlogn)$
Expected time complexity of quicksort is in $O(nlogn)$

Recurrence

Insertion Sort

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n + 1 & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + (n+1) \\ &= T(n-2) + (n-1) + 1 + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-2) + 1 + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-3) + 1 + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &= \ldots \\ &= T(n-i) + (n-i+2) + (n-i+3) + \ldots + (n-i+i) + (n-i+(i+1)) \end{aligned}$$

We hit the base case if $n - i = 1$, that is, $i = n - 1$. We substitute and find:

$$\begin{aligned} T(n) &= 1 + 3 + 4 + \ldots + n + (n+1) \\ &= (\Sigma_{i=1}^{n+1} i) - 2 \\ &= \tfrac{(n+1)(n+2)}{2} - 2 \\ &= \tfrac{1}{2}n^2 + \tfrac{3}{2}n - 1 \end{aligned}$$

Hence $T(n)$ is in $\Theta(n^2)$.

Merge Sort

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n \\ &= 2 \cdot (2 \cdot T(\tfrac{n}{4}) + \tfrac{n}{2}) + n \\ &= 4 \cdot T(\tfrac{n}{4}) + 2n \\ &= 4 \cdot (2 \cdot T(\tfrac{n}{8}) + \tfrac{n}{4}) + 2n \\ &= 8 \cdot T(\tfrac{n}{8}) + 3n \\ &= \ldots \\ &= 2^i \cdot T(\tfrac{n}{2^i}) + i \cdot n \end{aligned}$$

The base case is found if $\frac{n}{2^i} = 1$, that is, if $i = \log n$. We substitute this in the last equation found above: $T(n) = n \cdot 1 + n \cdot \log n = n + n \cdot \log n$. Hence T is in $\Theta(n \cdot \log n)$.

Quick Sort

`Worst Case (almost the same as Insertion Sort)`

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n + 1 & \text{if } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= T(n-1) + (n+1) \\
&= T(n-2) + (n-1) + 1 + (n+1) \\
&= T(n-2) + n + (n+1) \\
&= T(n-3) + (n-2) + 1 + n + (n+1) \\
&= T(n-3) + (n-1) + n + (n+1) \\
&= T(n-4) + (n-3) + 1 + (n-1) + n + (n+1) \\
&= T(n-4) + (n-2) + (n-1) + n + (n+1) \\
&= \ldots \\
&= T(n-i) + (n-i+2) + (n-i+3) + \ldots + (n-i+i) + (n-i+(i+1))
\end{aligned}$$

We get to the base case if $n - i = 1$ so if $i = n - 1$. Substituting this back in yields $T(n) = T(0) + T(1) + 2 + \ldots + n$. This gives $T(n)$ in $\Theta(n^2)$.

`Best Case (almost the same as Merge Sort)`

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T(\tfrac{n}{2}) + n \\
&= 2 \cdot (2 \cdot T(\tfrac{n}{4}) + \tfrac{n}{2}) + n \\
&= 4 \cdot T(\tfrac{n}{4}) + 2n \\
&= 4 \cdot (2 \cdot T(\tfrac{n}{8}) + \tfrac{n}{4}) + 2n \\
&= 8 \cdot T(\tfrac{n}{8}) + 3n \\
&= \ldots \\
&= 2^i \cdot T(\tfrac{n}{2^i}) + i \cdot n
\end{aligned}$$

The base case is found if $\frac{n}{2^i} = 1$, that is, if $i = \log n$. We substitute this in the last equation found above: $T(n) = n \cdot 1 + n \cdot \log n = n + n \cdot \log n$. Hence T is in $\Theta(n \cdot \log n)$.