Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Jan. 18, 2023 (Lecture 2)

Growth of Functions

Asymptotic Notations

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

  $O \approx \leq$

  $\Omega \approx \geq$

  $\Theta \approx =$

  $o \approx <$

  $\omega \approx >$

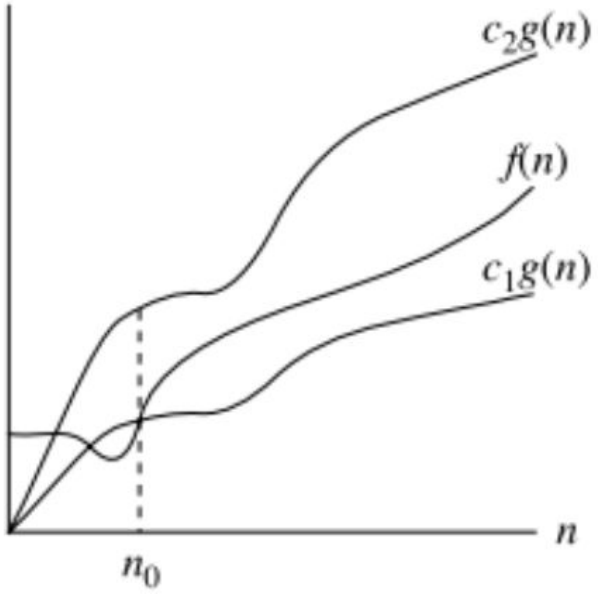
- We will mainly focus on Θ, O, Ω

Θ-Notation

For a given function $g(n)$, $\Theta(g(n))$ is defined as the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

If we can find out positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$, then we say that $f(n) \in \Theta(g(n))$. Usually, we will use the notation $f(n) = \Theta(g(n))$ to denote the same notation.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

An Example

Let $g(n) = n^2$, $f(n) = 5.5n^2 + 3n + 1$, prove that $f(n) = \Theta(n^2)$.

Proof:

- Let $c_1 = 5$, it is obvious that for all $n \geq 1$, $f(n) \geq 5g(n) \geq 0$.
- We may let $c_2 = 6$, next we need to find n_0 such that for all $n \geq n_0$,
- $5.5n^2 + 3n + 1 \leq 6n^2$, or equivalently $0.5n^2 \geq 3n + 1$.
- Dividing by $0.5n$ for both sides, we have $n \geq 6 + \frac{2}{n}$.
- So we can choose $n_0 = 7$.
- Overall, we have for $c_1 = 5$, $C_2 = 6$ and $n_0 = 7$,
- $0 \leq 5n^2 \leq 5.5n^2 + 3n + 1 \leq 6n^2$ for all $n \geq n_0$.
- Therefore $f(n) = \Theta(n^2)$.

In general, for a polynomial of n

$$f(n) = \sum_{i=0}^{t} a_i n^t = a_0 + a_1 n + \cdots + a_t n^t,$$

where $a_0, a_1, \ldots, a_t$ are constants and $a^t > 0$, we can use a similar method to prove that $f(n) = \Theta(n^t)$.

- $f(n) \in \Theta(g(n))$ means, there is positive integer n_0 such that when $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ within a constant factor.
- So we say that $g(n)$ is an asymptotically tight bound for $f(n)$.
- The definition of Θ(g(n)) requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative, i.e., $f(n)$ is nonnegative for sufficiently large n.
- It is obvious that $g(n) = \Theta(g(n))$.

Another Example

Use the formal definition to prove that $n^3 \neq \Theta(n^2)$.

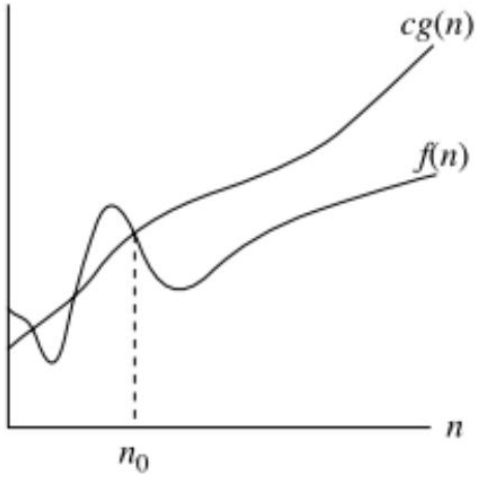
Proof:

- We should show that for any positive constant c and any integer n_0, we can always find some integer $n > n_0$, such that $cn^3 > n^2$.
- In fact, for any positive constant c and any integer n_0, we can choose $n > max\{\frac{1}{c}, n_0\}$
  Then $cn^3 > n^2$. Therefore $n^3 \neq \Theta(n^2)$.

O-Notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, because the definition of Θ notation is stronger than O notation. In the Θ notation, we need to find c_1, c_2 and n_0 and we can use c_2 and n_0 for the O notation.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Notes for O-Notation

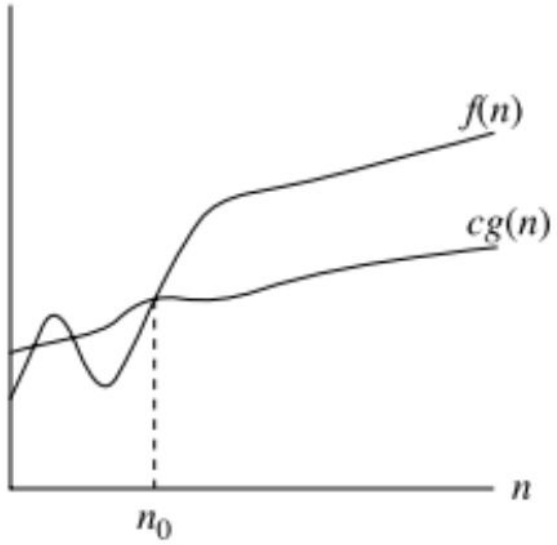
- For a linear function $f(n) = an + b$, we can prove that $f(n) = O(n)$. In fact, we can let $c = a + |b|$ and $n_0 = 1$. It is easy to see that $f(n) = O(n^2)$ by using the same constants. So, O-notation just gives an upper bound which may not be tight.
- When we use O-notation to describe the upper bound of the running time for the worst case of an algorithm, we have bounded on the running time of the algorithm for any input. But the Θ-notation does not have the similar property. For example, the insertion sort for the best case is $\Theta(n)$ but not $\Theta(n^2)$.
- Examples of functions in $O(n^2)$: $n^2, n^2 + 3n + 2$, also n, $n \log n$, $n^{1.99}$, etc.

Ω-Notation

Ω notation gives an asymptotic lower bound for a function. The definition of Ω notation is defined as follows.

$$\begin{aligned}\Omega(g(n)) &= \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ &\quad 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\end{aligned}$$

The running time of an algorithm being $\Omega(g(n))$ means that the running time, independent from the properties of input, in at least a constant times $g(n)$. Therefore this is also a lower bound of the best case.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

- Examples of functions in $\Omega(n^2)$:
  $n^2, n^2 + 3n + 2$, also n^3, $n^2 \log n$, $n^{2.001}$, etc.

From the definitions of the asymptotic notations we learned so far, the following theorem is not difficult to prove.

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Some Properties

Suppose that $f(n)$ and $g(n)$ are asymptotic positive. Then

- Transitivity

  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,

  $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,

  $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,.

- Reflexivity

  $f(n) = \Theta(f(n))$,

  $f(n) = O(f(n))$,

  $f(n) = \Omega(f(n))$.

- Transpose symmetry

  $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

o-Notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

ω-Notation

$$\omega(g(n)) = \{f(n) : \text{ for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$$

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Example – Merge Sort Analysis

Dividing and Conquer Approach

- In this approach ,we solve the sorting problem **recursively** by applying 3 steps:

1. **DIVIDE**-break the problem into several sub problems of smaller size.
2. **CONQUER**-solve the problem recursively.
3. **COMBINE**-combine these solutions to create a solution to the original problem.

Algorithm D&C (P) :

if small(P)

then return solution of P

else

divide P into smaller instances P_1 ,P_2 .....P_k
Apply D and C to each sub problem
Return combine D&C(P_1), D&C(P_2), D& C(P_k)

Merge Sort

Input: a sequence of n numbers, which we will assume is stored in an array A [1...n].
Output a permutation of this sequence, sorted in increasing order.

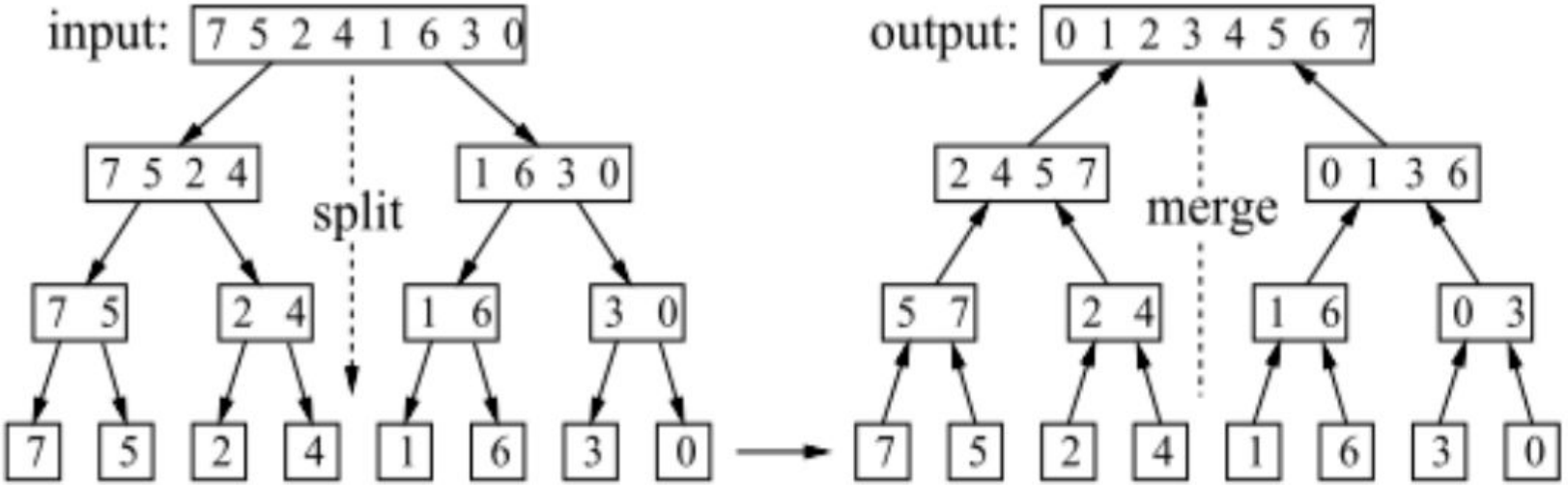
How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 .

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

Merge Sort



Merge Sort

Pseudocode

```
MERGE-SORT(A, p, r)
1       if p<r
2               q= lower bound of (p +r)/2
3               MERGE-SORT(A, p, q)
4               MERGE-SORT(A, q +1, r)
5               MERGE(A, p, q, r)
6       else
7               return A[p]
```

Running Time Analysis

- The running time of MERGE(A, p, q, r) can be proved that it is $\Theta(n)$ where $n=r-p+1$;

- Assume MERGE-SORT(A, p, r) running time is T(n) where $n=r-p+1$:

  Case 1: when n is a constant, including only 1 number, then $T(1)= \Theta(1)$;
  Case 2: when n is a general array, including more than 1 number;
  then

  $T(n) = 2T(n/2)+n\ \Theta(1)$
  $= 2T(n/2)+cn$
  $= 2T(n/2)+ \Theta(n)$

  for some constant c

Recurrence and Solutions

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more simpler instances of the same problem.

Example: merge sort and whose running time is

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

Several ways to solve T(n) :

1. Substitution method
2. Iterative method
3. Recursion tree method
4. master method

Substitution Method

1. Guess the form of the solution
2. Verify by induction
3. Solve for constants

Example : solution for merge sort T(n)

Step 1: For the example above, first we guess that $T(n)=O(n^2)$ which is tight upper bound

Step 2: Assume,$T(k)<= c\ k^2$ for k<n so,we should prove that $T(n)<=c\ n^2$

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

$<= 2c\ (n/2)^2+ dn$ (for some constant d)

$<= cn^2$ (for some c)

Step 3: solve for c: from $cn^2/2+ dn <= cn^2$, we let c = 2d

Iteration Method

Example:

$T(n)=2T(n/2)+n$
$= 2[2T(n/4) + n/2]+n$
$<= 2^2 T(n/4)+2n$
$<= 2^3 T(n/8)+3n$

After k iterations ,$T(n)<=2^k T(n/2^k)+kn$
Sub problem size is 1 after $n/2^k =1 => k= \log n$
So,after logn iterations,the sub-problem size will be 1.
So,when k=logn is put in equation
$T(n)<= nT(1)+n\log n$
So $T(n)= O(n \log n)$.

In fact, we can also prove $T(n)= \Omega(n \log n)$ and therefore $T(n)= \Theta(n \log n)$.

Iteration Method

Example:

$T(n)=2T(n/2)+n$
$= 2[2T(n/4) + n/2]+n$
$<= 2^2 T(n/4)+2n$
$<= 2^3 T(n/8)+3n$

After k iterations ,$T(n) <= 2^k T(n/2^k)+kn$
Sub problem size is 1 after $n/2^k =1 => k= \log n$
So,after logn iterations,the sub-problem size will be 1.
So,when k=logn is put in equation
$T(n)<= nT(1)+n\log n$
So $T(n)= O(n \log n)$.

In fact, we can also prove $T(n)= \Omega(n \log n)$ and therefore $T(n)= \Theta(n \log n)$.

Iteration Method

Example:

$T(n)=2T(n/2)+n$
$= 2[2T(n/4) + n/2]+n$
$<= 2^2 T(n/4)+2n$
$<= 2^3 T(n/8)+3n$

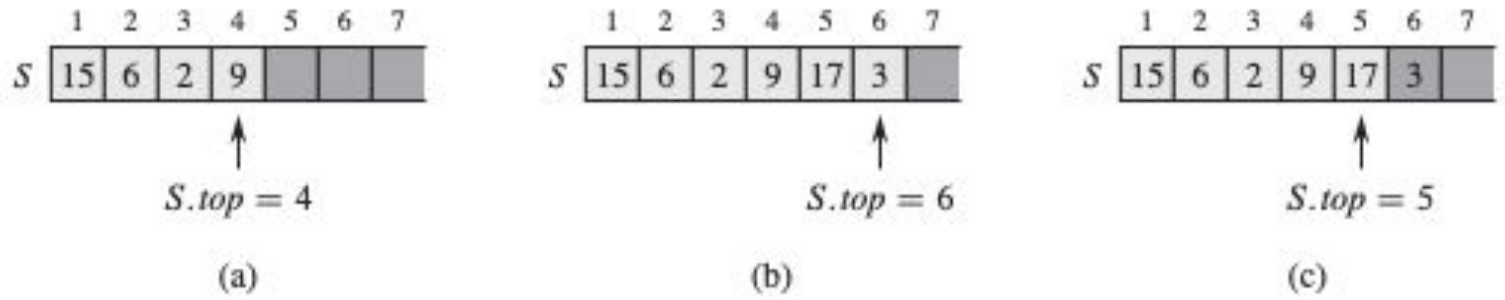
After k iterations ,$T(n)<=2^k T(n/2^k)+kn$
Sub problem size is 1 after $n/2^k =1 => k= \log n$
So,after logn iterations,the sub-problem size will be 1.
So,when k=logn is put in equation
$T(n)<= nT(1)+n\log n$
So $T(n)= O(n \log n)$.

In fact, we can also prove $T(n)= \Omega(n \log n)$ and therefore $T(n)= \Theta(n \log n)$.

Data Structure Review

Stack

- A stack is a linear dynamic set with the property last-in first-out (LIFO). The operations of this data structure are at the top of the stack.
- We can use an array to implement a stack. A stack consists of elements S[1 . . . S.top], where S[1] is the element at the bottom of the stack and S[S.top] is the element at the top. When S.top == 0, the stack is empty.



Stack Basic Operations

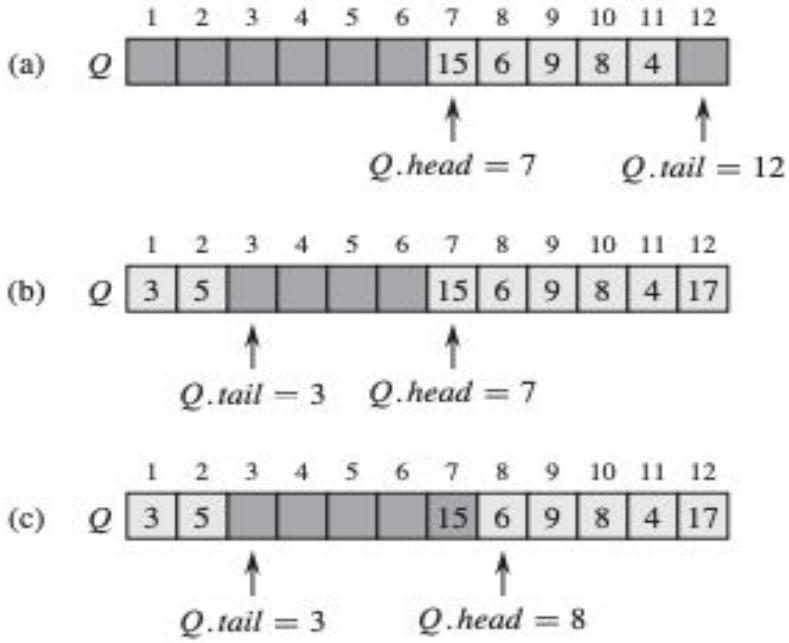
```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Queues

- The queues have the property first-in first-out (FIFO). The operations on the queues are at two ends, the head and the tail.
- We also use an (circular) array to implement a queue.



Queues Basic Operations

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Linked List

- Linked lists are more flexible representation for dynamic sets.
- One example implementation is doubly linked list. Each element of a doubly linked list is an object with and attribute “key” and two other pointer attributes: next and prev.
- The object may also contain other data. In general a linked list may be single linked, doubly linked or circular list.
- The first element of the list is called the head and the last element is called the tail.
- The list can be sorted or unsorted.

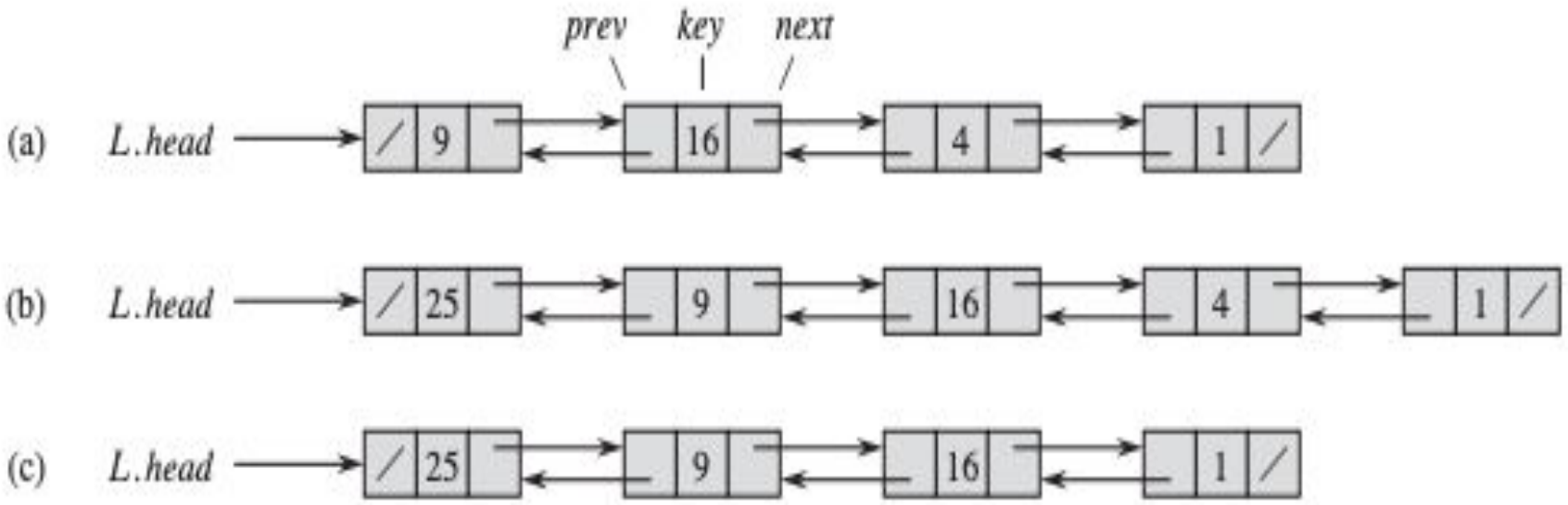
Linked List Operations

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

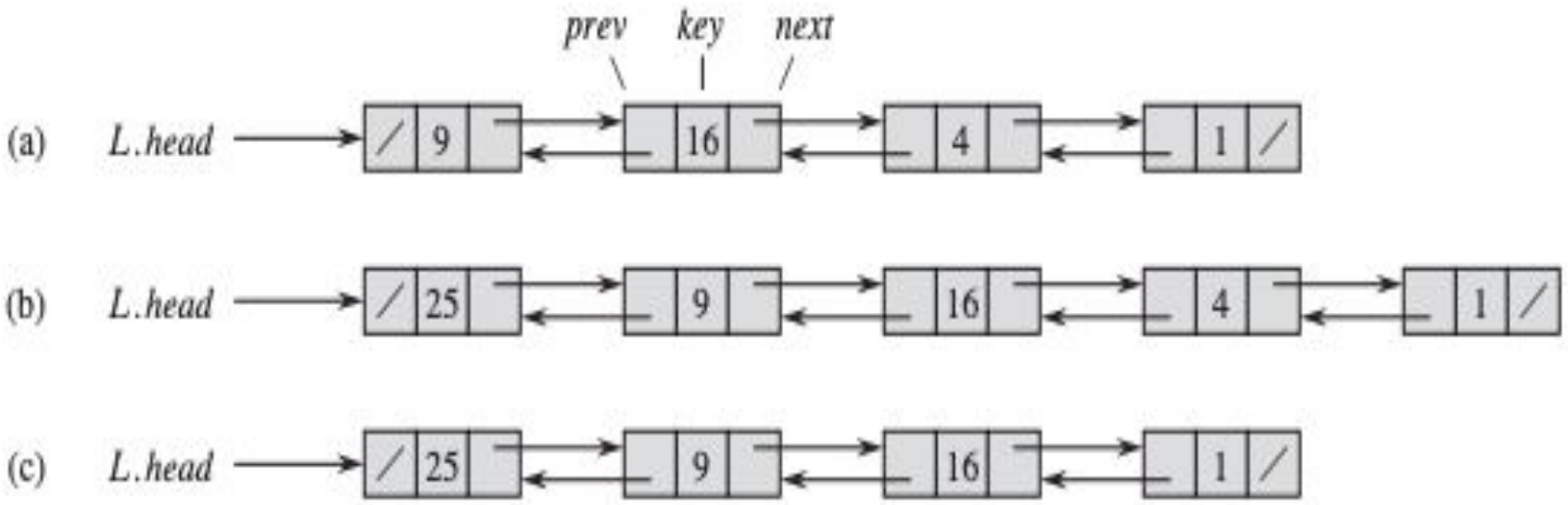
```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

Linked List Operations



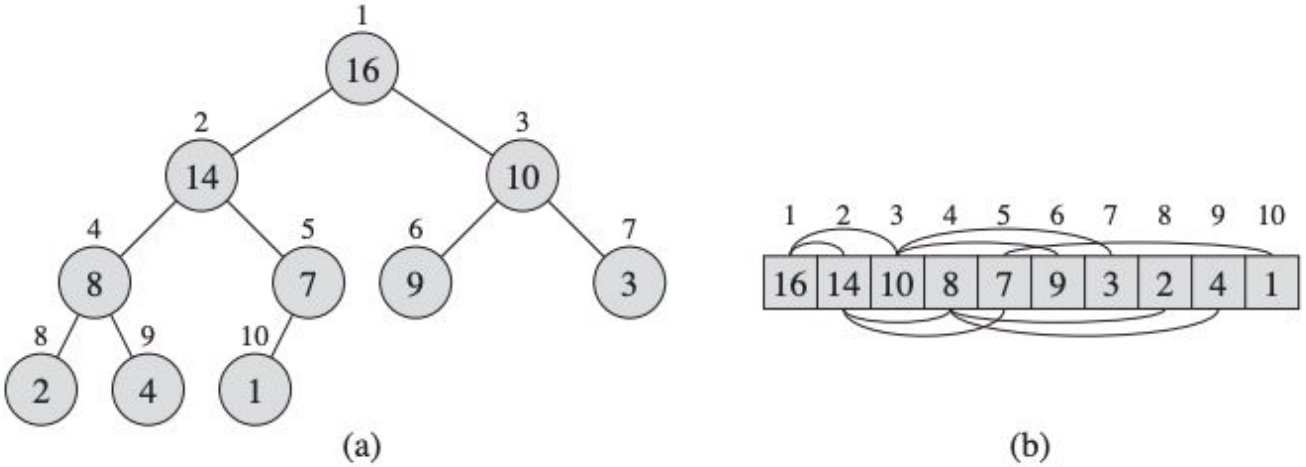
Linked List Operations



Heap

- The (binary) heap data structure is an array that we can view as a nearly complete binary tree.
- An array *A* that represent a heap is an object with two attributes:
  - A.length which gives the number of elements in the array, and
  - A.heap-size, which represents how many elements in the heap are stored within array A.
  - Although A[1.. A.length] may contain numbers, only the elements in A[1.. A.heap-size], where 0<A.heap-size <= A.length, are valid elements of the heap. The root of the tree is A[1].

Heap



A max-heap viewed as (a) a binary tree and (b) an array.

The number within the circle at each node in the tree is the value stored at that node.

The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

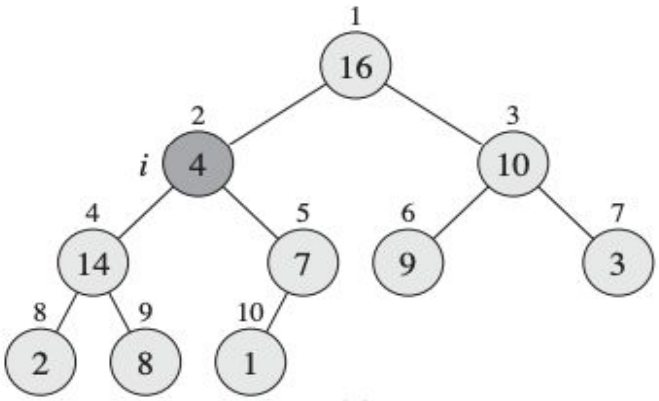
The tree has height three; the node at index 4 (with value 8) has height one.

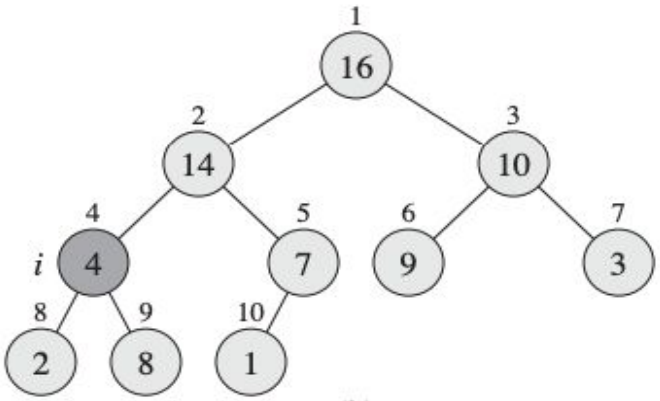
Notes for Heaps

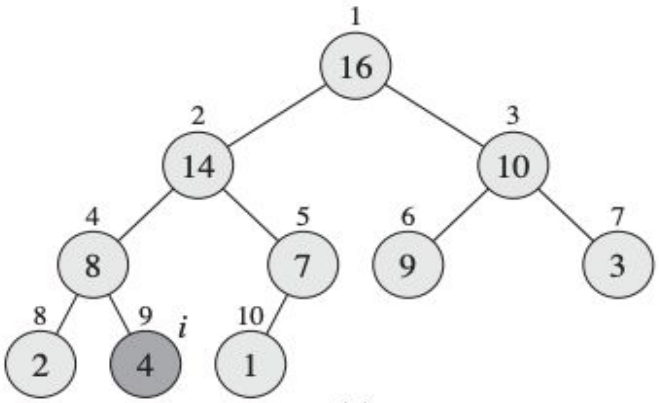
- On most computers, it is very efficient to compute 2i or ⌊i/2⌋, just shift the binary representation of i left, or right, by one bit position.
- There are two kinds of binary heaps: max-heaps and min-heaps. In a max-heap, A[Parent(i)] ≥ A[i], while in a min-heap, A[Parent(i)] ≤ A[i] for any node A[i].
- Next we consider main operations (use max-heap as example).

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

The running time for Max-Heapify is O(log n), where n is the heap size.


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(a)

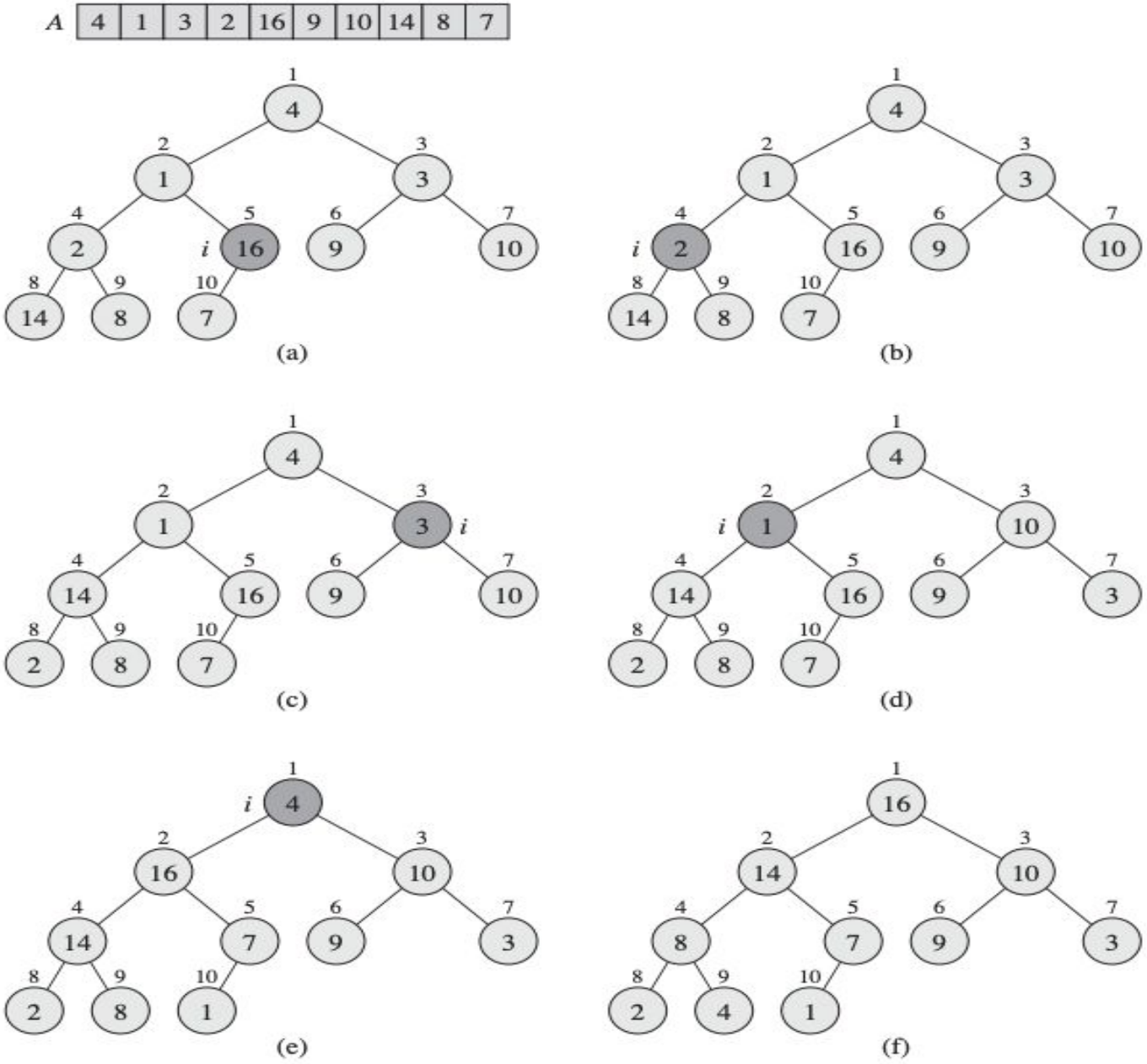

1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1
(c)

Build Max Heaps

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A into a max-heap.A more careful discussion can show that the asymptotically running time is O(n). We omitted the proof here.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

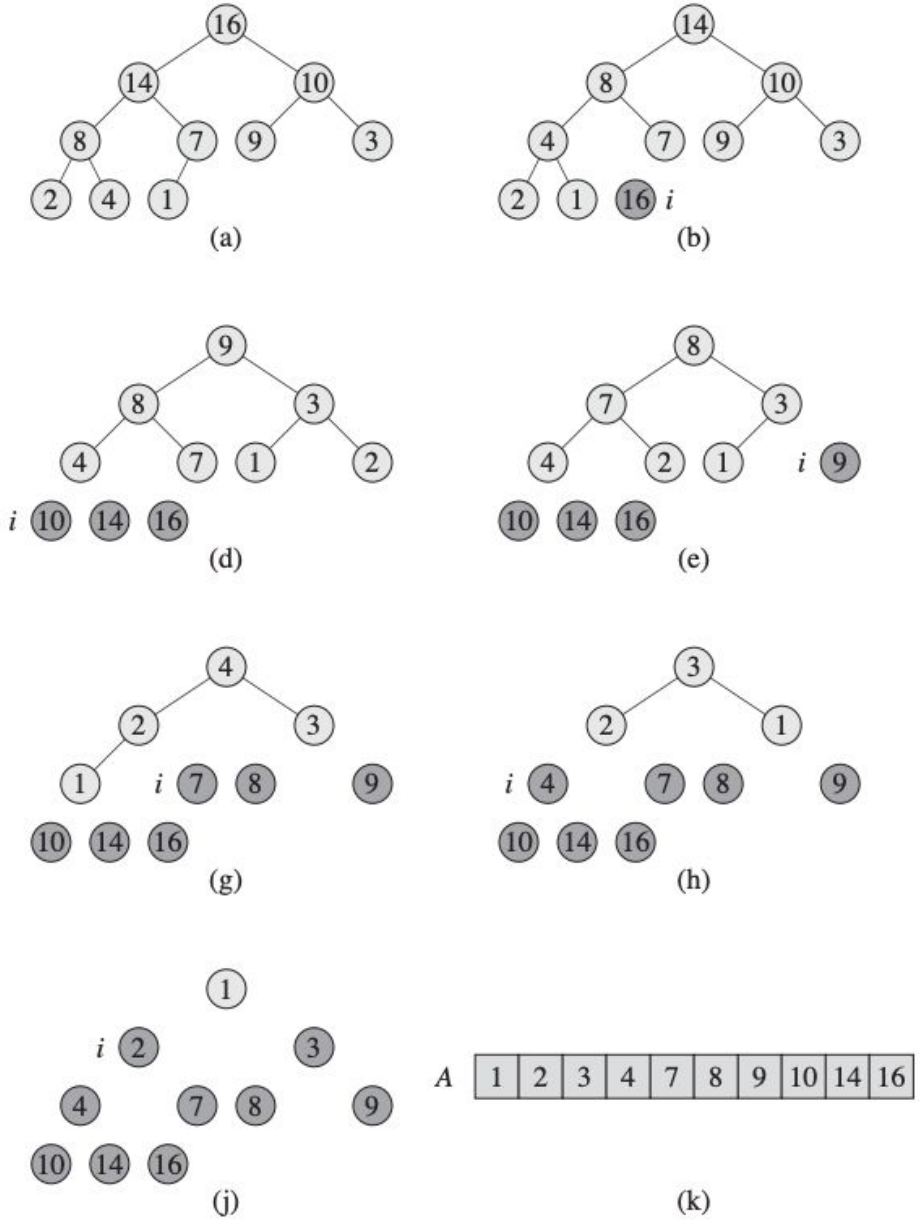
After Class

Read: Part 1 Chapter 3, Chapter 4.1-4.3, Chapter 10.1-10.2

Heap Sort

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A with length n. The Heapsort procedure takes time O(n log n).

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```


16 14 10 8 7 9 3 2 4 1
(a)
14 8 10 4 7 9 3 2 1 16 i
(b)
10 8 9 4 7 1 3 i 2 14 16
(c)
9 8 3 4 7 1 2 i 10 14 16
(d)
8 7 3 4 2 1 i 9 10 14 16
(e)
7 4 3 1 2 8 i 9 10 14 16
(f)
4 2 3 1 i 7 8 9 10 14 16
(g)
3 2 1 i 4 7 8 9 10 14 16
(h)
2 1 3 i 4 7 8 9 10 14 16
(i)
1 i 2 3 4 7 8 9 10 14 16
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Application – priority queues

One important application for the heap is priority queues.

There are two types of priority queues: max-priority queues and min-priority queues. We use a max-priority queue to explain the main ideas. The min-priority queue is similar.

A max-priority queue maintains a set S and supports the following operations.

- **Insert(S, x)** inserts the element x into the set S, which is equivalent to the set operation $S = S \cup \{x\}$.
- **Maximum(S)** returns the element 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.

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

The procedure HEAP-MAXIMUM has running time in Θ(1).

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

The procedure HEAP-EXTRACT-MAX has running time O(log n).

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

The procedure HEAP-INCREASE-KEY has running time in O(log n).

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

The procedure MAX-HEAP-INSERT has running time O(log n).