CSC 411
Design and Analysis of Algorithms

Chapter 6 Transfer and Conquer
- Part 3

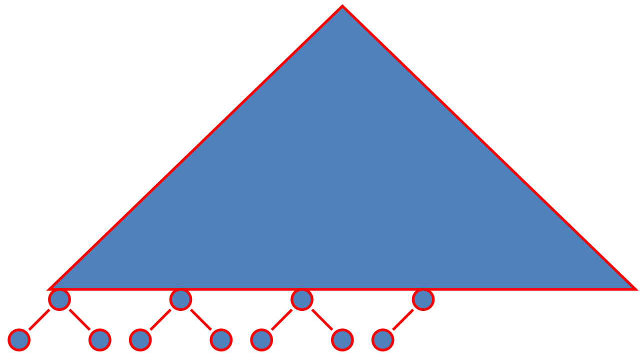
Instructor: Minhee Jun

Transfer-and-Conquer Examples

- Presorting (6.1)
- Gaussian Elimination (6.2)
- Balanced Search Trees (6.3)
  - AVL Trees
  - 2-3 Trees
- Heaps and Heapsort (6.4)
- Horner’s Rule and Binary Exponentiation (6.5)

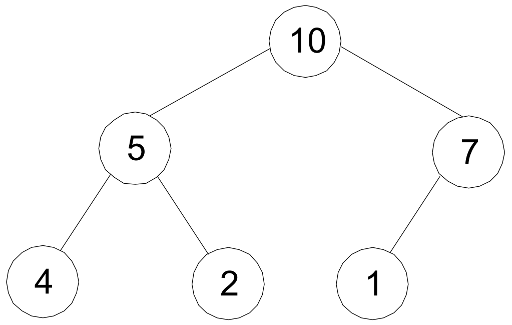
Heaps and Heapsort

- Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:
  - It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

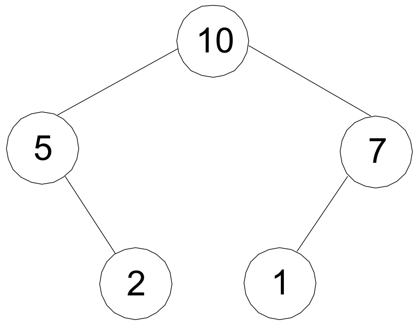


  - The key at each node is ≥ keys at its children
    - The value of parent is always greater than its children

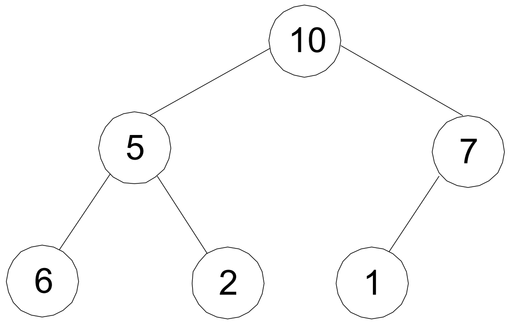
Illustration of the heap's definition



a heap



not a heap



not a heap

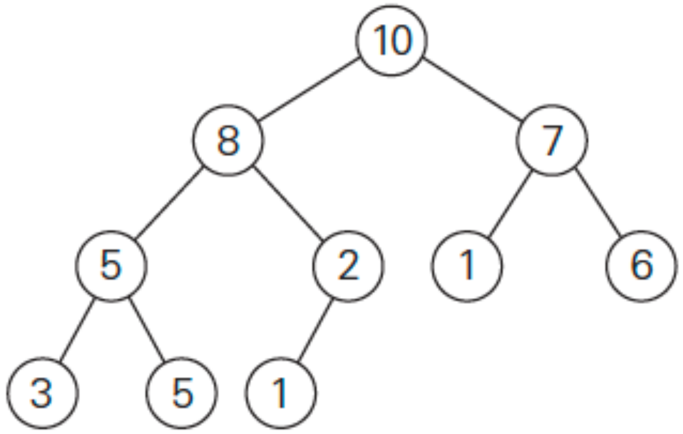
- Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

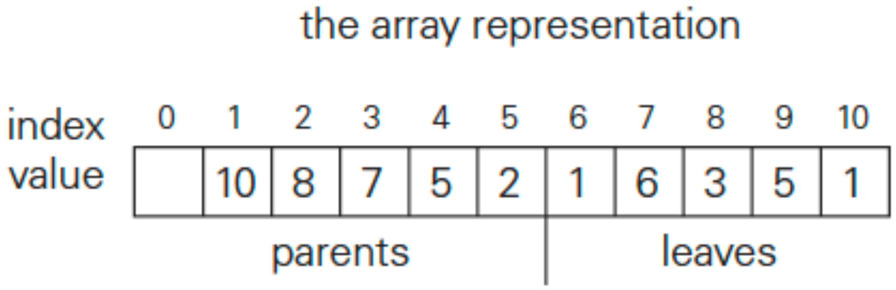
Some Important Properties of a Heap

- Given *n*, there exists a **unique** binary tree with *n* nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

Heap's Array Representation

- Store heap's elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order
- Example:





the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

parents | leaves

- Left child of node *j* is at 2*j*, Right child of node *j* is at 2*j*+1
- Parent of node *j* is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

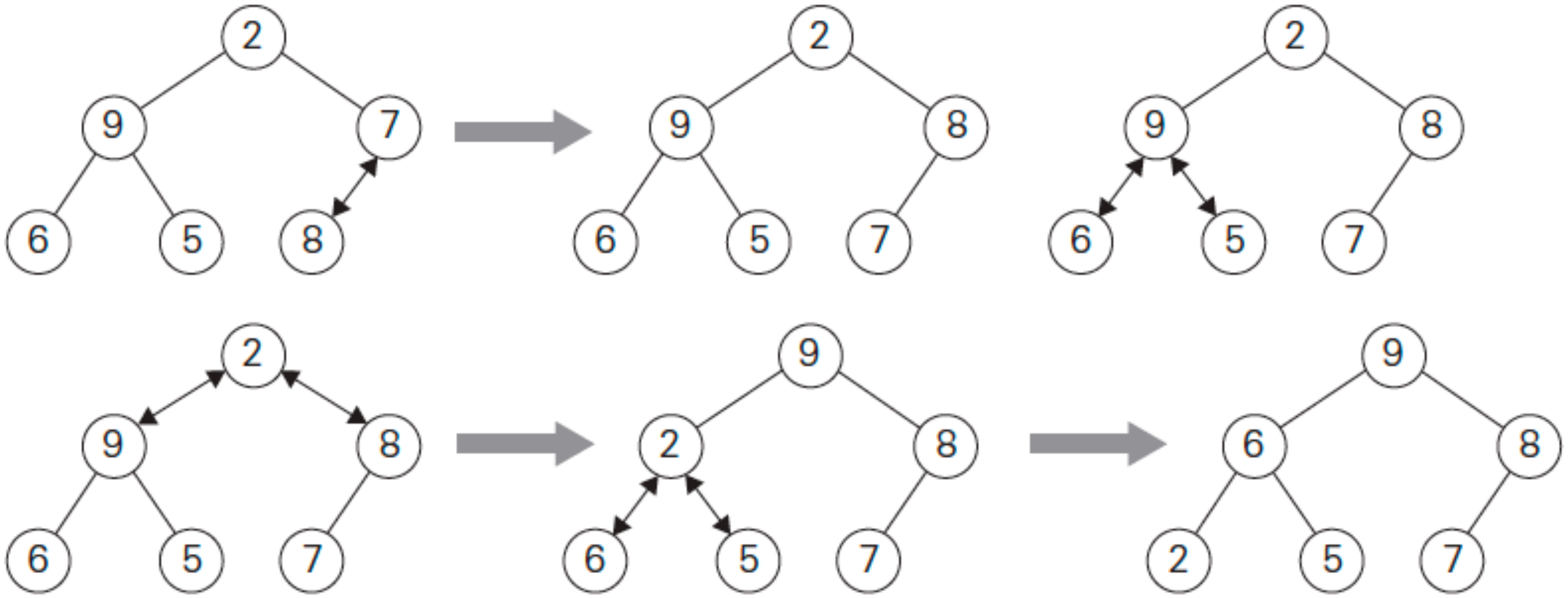
HEAP CONSTRUCTION (BOTTOM-UP)

- How can we construct a heap from a given list of keys
  - Step 0: Initialize the essentially complete binary tree with n nodes by placing keys in the order given
  - Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds
  - Step 2: Repeat Step 1 for the preceding parental node

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8

Bottom-up algorithm:

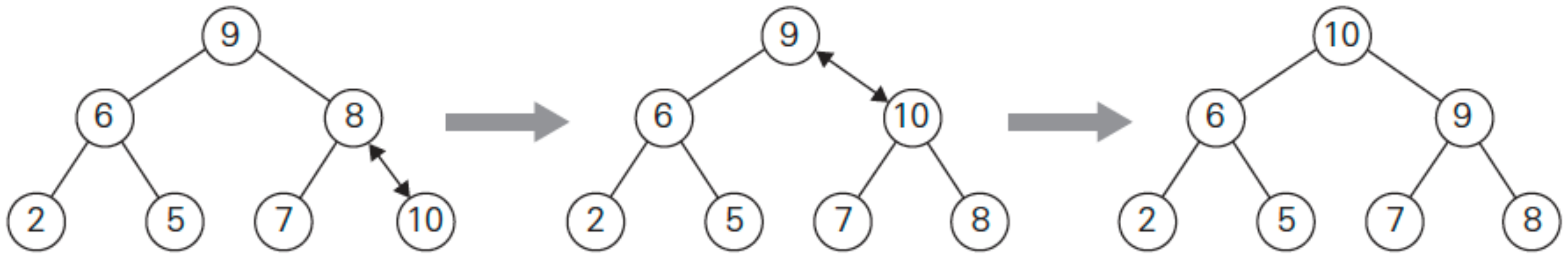


Bottom-up heap construction: pseudocode

```
ALGORITHM  $HeapBottomUp(H[1..n])$
    //Constructs a heap from elements of a given array
    // by the bottom-up algorithm
    //Input: An array $H[1..n]$ of orderable items
    //Output: A heap $H[1..n]$
    for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do
        $k \leftarrow i$;    $v \leftarrow H[k]$
        $heap \leftarrow$ false
        while not $heap$ and $2 * k \le n$ do
            $j \leftarrow 2 * k$
            if $j < n$    //there are two children
                if $H[j] < H[j+1]$  $j \leftarrow j + 1$
            if $v \ge H[j]$
                $heap \leftarrow$ true
            else $H[k] \leftarrow H[j]$;    $k \leftarrow j$
        $H[k] \leftarrow v$
```

Heap Construction (Top-Down)

- Insertion of a new element into a previously constructed heap
  - Insert the new element at last position in heap.
  - Compare it with its parent and, if it violates heap condition, exchange them
  - Continue comparing the new element with nodes up the tree until the heap condition is satisfied
- Example: Insert key 10



- Efficiency: O(log *n*)

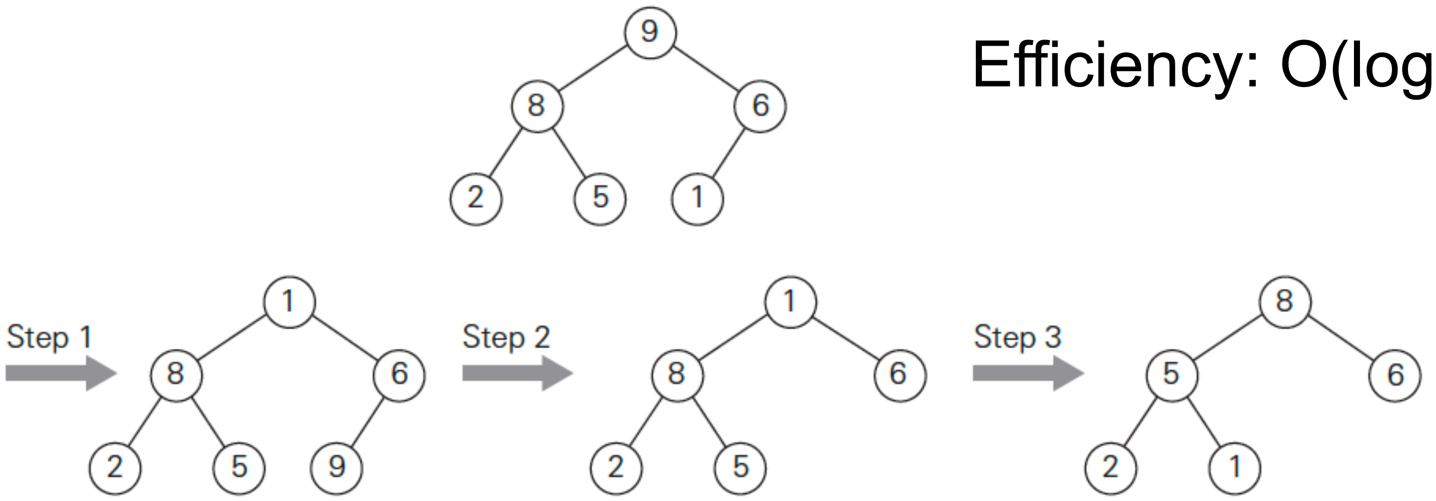
Maximum Key Deletion from a heap

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

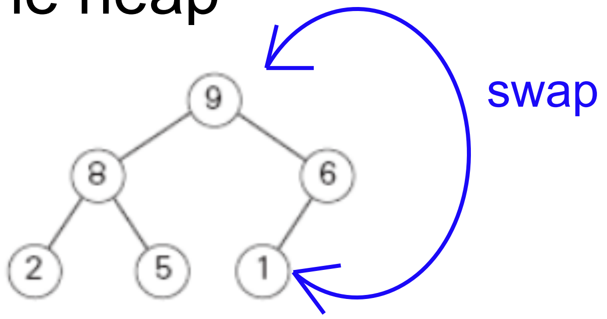
Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.



Efficiency: O(log n)

Deleting the root from a heap

- Delete "9" from the heap



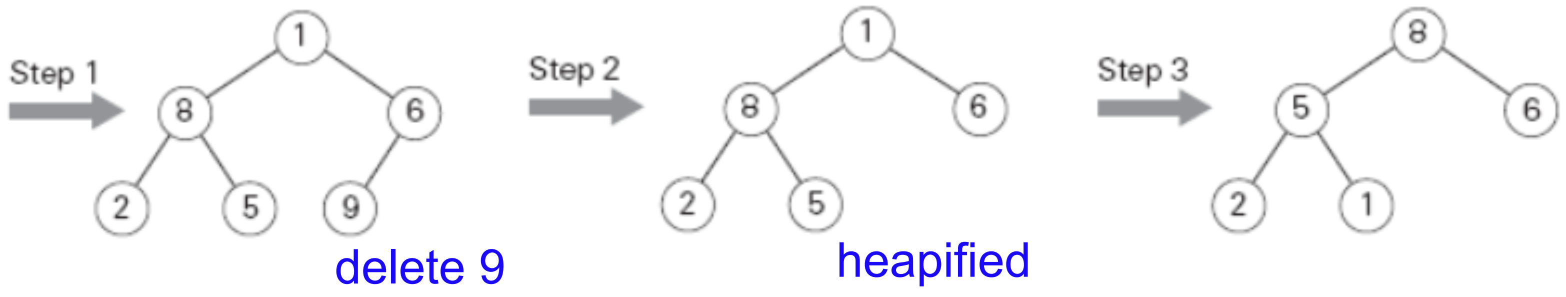


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Heapsort

Stage 1: heap construction

- Construct a heap for a given list of *n* keys

Stage 2: maximum deletions

- Apply the root-deletion operation n-1 times to the remaining heap.
    - Exchange keys in the root and in the last (rightmost) leaf
    - Decrease heap size by 1
    - If necessary, swap new root with larger child until the heap condition holds

Heapsort: stage 1 - heap construction

Construct a heap for the list 2, 9, 7, 6, 5, 8

Stage 1 (heap construction)

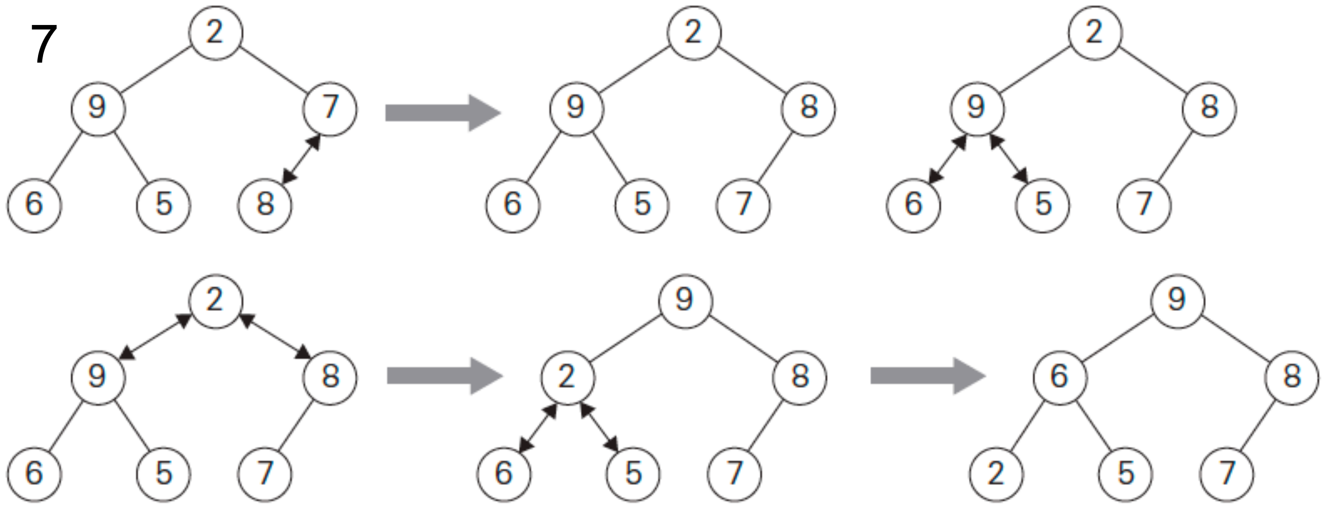
2 9 $\underline{7}$ 6 5 8

2 $\underline{9}$ 8 6 5 7

$\underline{2}$ 9 8 6 5 7

9 $\underline{2}$ 8 6 5 7

9 6 8 2 5 7



Heapsort: stage 2 - maximum deletions

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2 9 $\underline{7}$ 6 5 8

2 $\underline{9}$ 8 6 5 7

$\underline{2}$ 9 8 6 5 7

9 $\underline{2}$ 8 6 5 7

9 6 8 2 5 7

Stage 2 (root/max removal)

$\underline{9}$ 6 8 2 5 7

7 6 8 2 5 | 9

$\underline{8}$ 6 7 2 5 | 9

5 6 7 2 | 8 9

$\underline{7}$ 6 5 2 | 8 9

2 6 5 | 7 8 9

$\underline{6}$ 2 5 | 7 8 9

5 2 | 6 7 8 9

$\underline{5}$ 2 | 6 7 8 9

2 | 5 6 7 8 9

Analysis of Heapsort

- Stage 1: Build heap for a given list of *n* keys

$$C_{worst}(n) = \sum_{i=0}^{h-1} 2^{h-i} \cdot 2 = 2(n - \log_2(n+1)) \in \Theta(n)$$

# nodes at level *i*

- Stage 2: Repeat operation of root removal *n*-1 times (fix heap)

$$C_{worst}(n) = \sum_{i=1}^{n-1} 2 \log_2 i \in \Theta(n \log n)$$

- *Efficiency:* $\Theta(n \log n)$ *in Both worst-case and average-case*

Horner’s Rule For Polynomial Evaluation

- Given a polynomial of degree n

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

  - Find the value of $p(x)$ at a specific value of x.

- Horner’s Rule
  - Successively take x as a common factor in the remaining polynomials of diminishing degrees:

  - $p(x) = (\ldots + (a_n x + a_{n-1})x + \cdots) + a_0$

  - Substitution into the last formula leads to a faster algorithm
  - Same sequence of computations are obtained by simply arranging the coefficient in a table and proceeding.

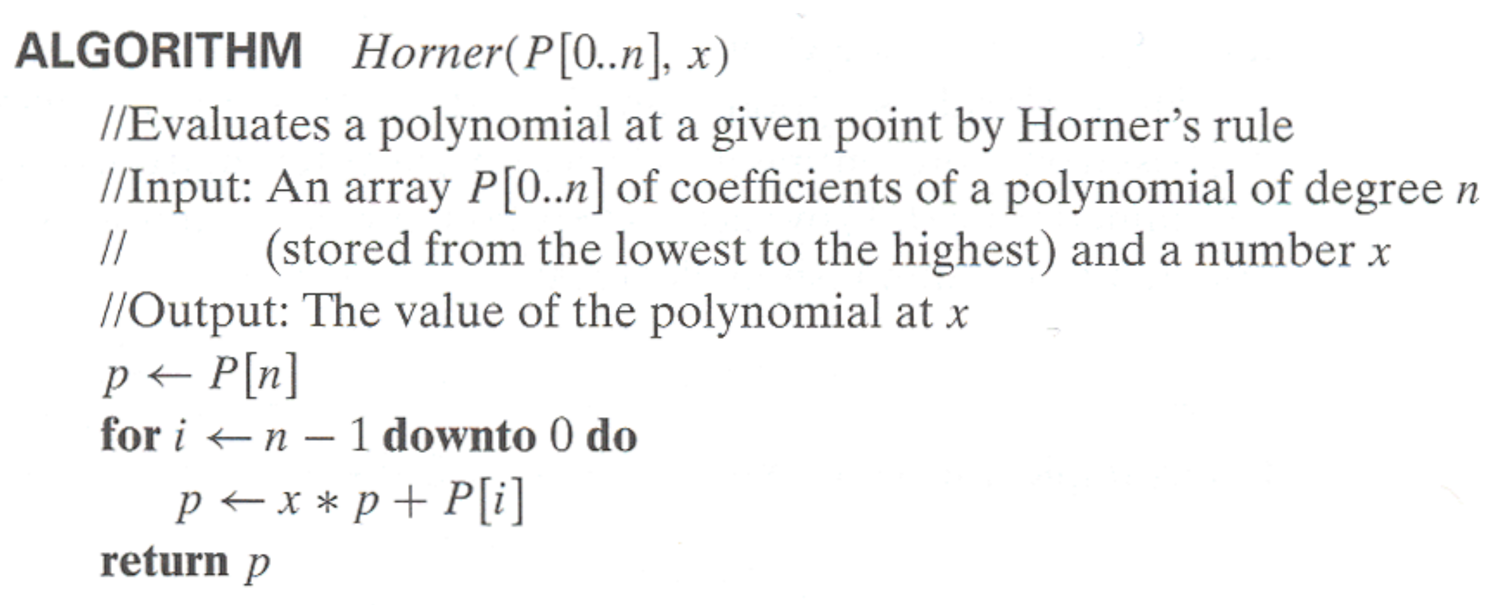
Horner’s Rule

- Substitution into the last formula leads to a faster algorithm
- Same sequence of computations are obtained by simply arranging the coefficient in a table and proceeding as follows:
- Example:

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned}$$

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Horner’s Rule : Pseudocode



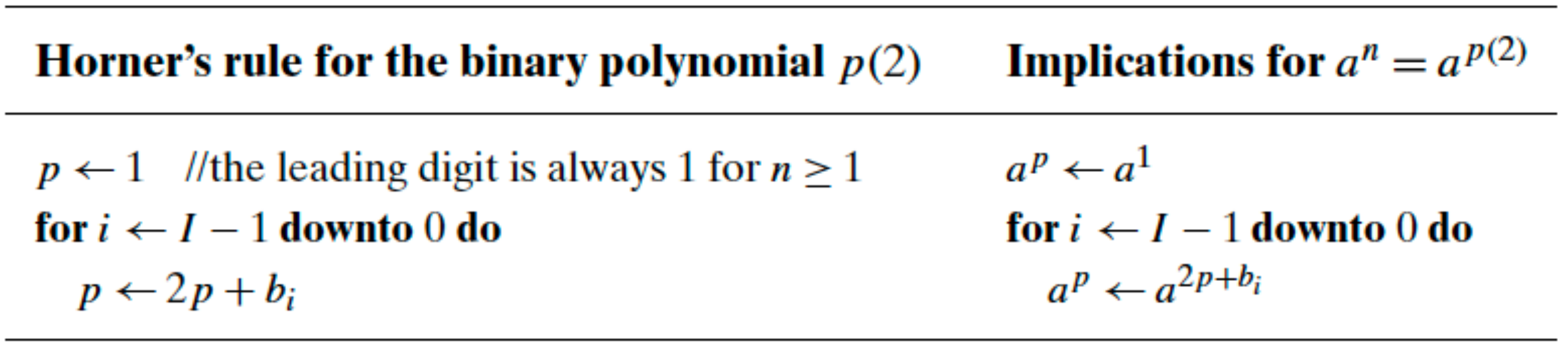
```
ALGORITHM  Horner(P[0..n], x)
    //Evaluates a polynomial at a given point by Horner's rule
    //Input: An array P[0..n] of coefficients of a polynomial of degree n
    //        (stored from the lowest to the highest) and a number x
    //Output: The value of the polynomial at x
    $p \leftarrow P[n]$
    for $i \leftarrow n-1$ downto 0 do
        $p \leftarrow x * p + P[i]$
    return $p$
```

- Efficiency of Horner’s Rule: # multiplications = # additions = n
- *Synthetic division* of of $p(x)$ by $(x - x_0)$
  - Example: Let $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$.
    - Find $p(x)$: $(x - 3)$

Binary Exponentiation: Computing a^n

- *The amazing effect of Horner's rule fades for $p(x) = x^n$.*
  - Like Brute-force multiplication.
  - Based on the representation-change, computing a^n:
    - $n = b_I \ \cdots \ b_i \ \cdots \ b_0$ (x is a base)
    - $n = p(x) = b_I x^I + \cdots + b_i x^i + \cdots + b_0$
    - $a^n = a^{p(2)} = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0}$ (the base $x = 2$)

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases}$$



Horner's rule for the binary polynomial $p(2)$	**Implications for $a^n = a^{p(2)}$**
$p \leftarrow 1$ //the leading digit is always 1 for $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ **downto** 0 **do**	**for** $i \leftarrow I - 1$ **downto** 0 **do**
$\quad p \leftarrow 2p + b_i$	$\quad a^p \leftarrow a^{2p+b_i}$

Binary Exponentiation: Left-to-right

- *<u>Left-to-right binary exponentiation</u>*
  - Initialize product accumulator by 1.
  - *Scan n's binary expansion from <u>left to right</u> and do the following:*

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0, \\ (a^p)^2 \cdot a & \text{if } b_i = 1. \end{cases}$$

- Example: Compute a^{13}. Here, $n = 13 = 1101_2$.

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

- Efficiency: $(b-1) \le M(n) \le 2(b-1)$ where $b = \lfloor \log_2 n \rfloor + 1$

Binary Exponentiation: Right-to-left

- *Right-to-left binary exponentiation*
  - *Scan n's binary expansion from right to left and compute a^n*

$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1, \\ 1 & \text{if } b_i = 0, \end{cases}$$

- Example: Compute a^{13}. Here, n = 13 = 1101_2.

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

- Efficiency: same as that of left-to-right binary exponentiation

$$(b-1) \leq M(n) \leq 2(b-1) \text{ where } b = \lfloor \log_2 n \rfloor + 1$$

Problem Reduction

- This variation of transform-and-conquer solves a problem by a transforming it into different problem for which an algorithm is already available.

- To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.