Computer Algorithms

Lecture 9: Transform-and-Conquer– Ch 6 – Cont'd

Dr. Manal Helal, Spring 2014.

http://moodle.manalhelal.com

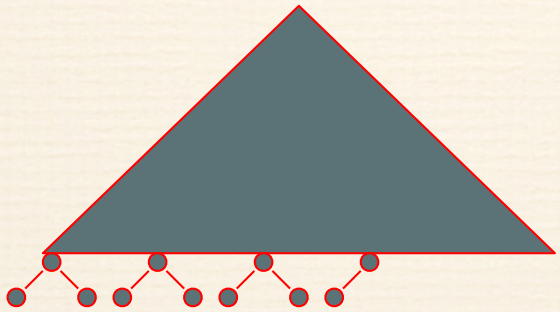
Lecture Learning Objectives

1. Use a Transform-and-Conquer algorithm design strategy to transform a problem to another easier representation such as Heap-sort, solving polynomials, and exponentiation.
2. Understand how to apply a Transform-and-Conquer algorithm design strategy to reduce a problem to another that has an existing solution.

Heaps and Heapsort

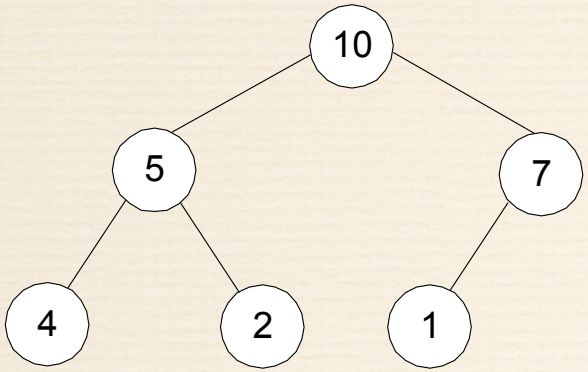
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

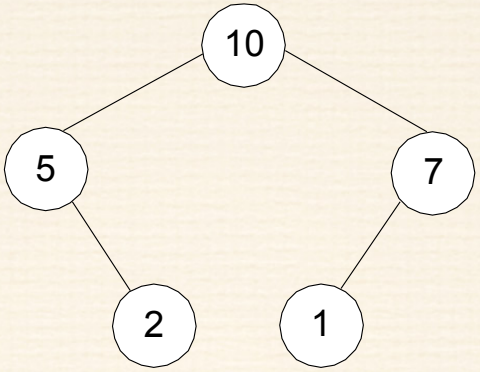


The key at each node is ≥ keys at its children

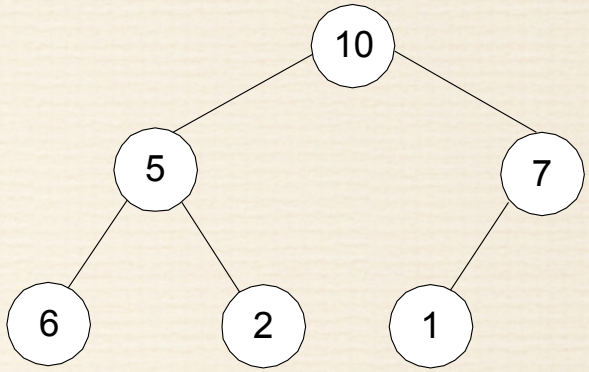
Illustration of the heap's definition



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

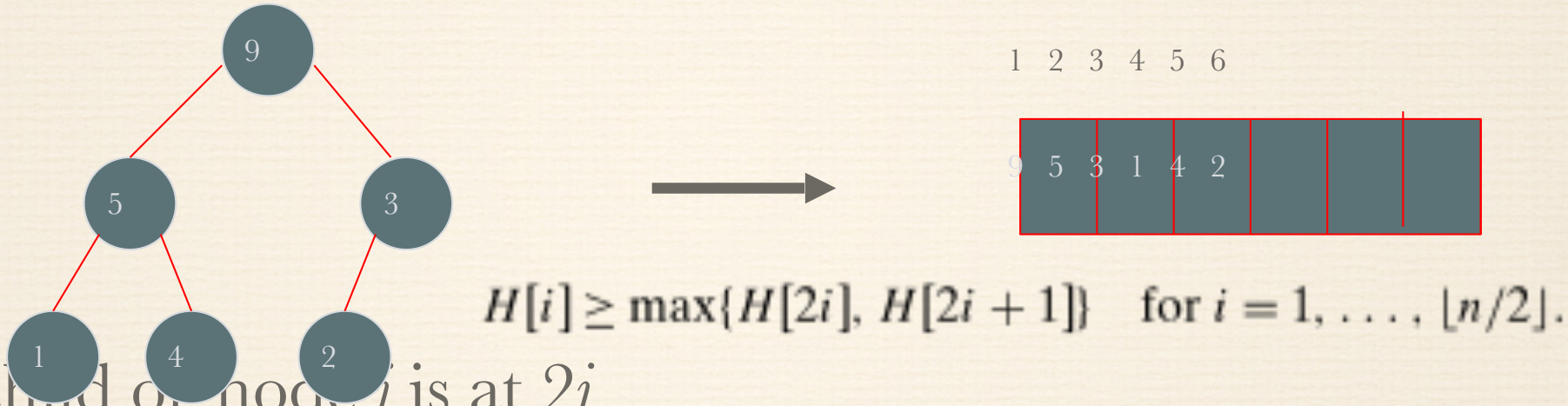
Some Important Properties of a Heap

- 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:



$$H[i] \geq \max\{H[2i], H[2i+1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor.$$

Left child of node j is at $2j$
Right child of node j is at $2j+1$
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)

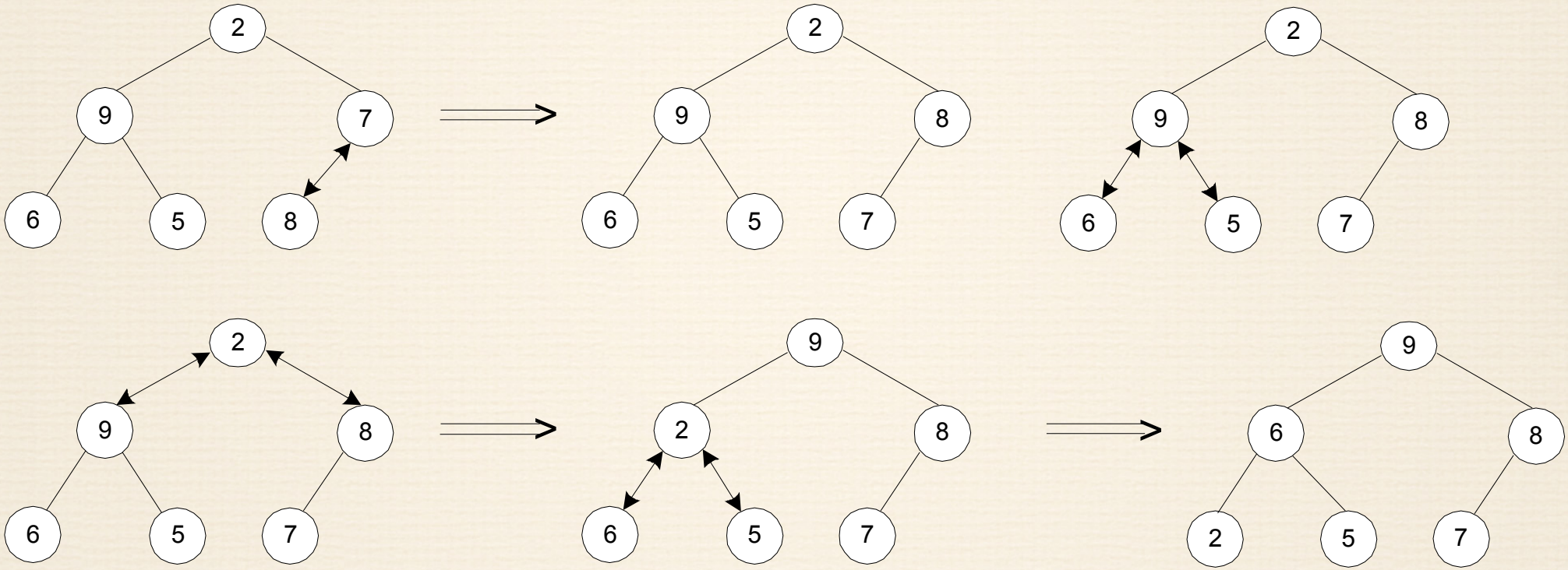
Step 0: Initialize the structure with keys in the order given

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



Pseudopodia of bottom-up heap construction

ALGORITHM *HeapBottomUp*$(H[1..n])$

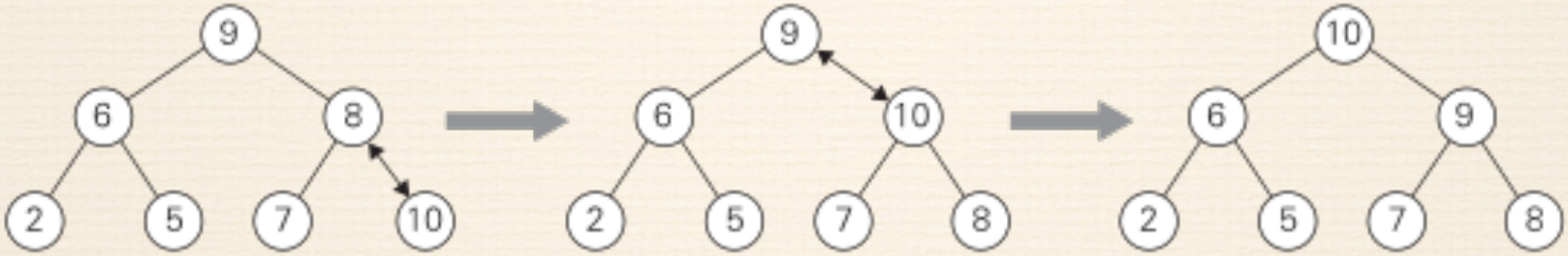
```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

Insertion of a New Element into a 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



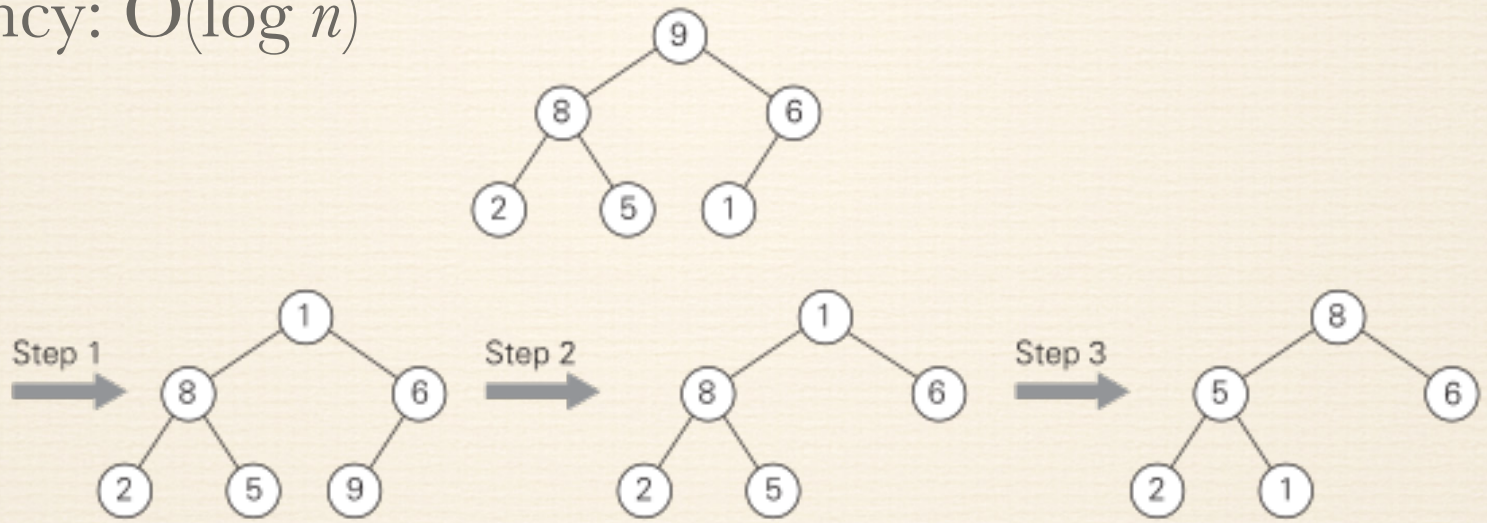
Example: Insert key 10

Efficiency: O(log n)

Heap Delete

- 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

Efficiency: O(log *n*)



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: Construct a heap for a given list of n keys

Stage 2: Repeat operation of root removal n-1 times:

Example of Sorting by Heapsort

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

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

Analysis of Heapsort

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

worst-case

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

nodes at level i

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

worst-case

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

Both worst-case and average-case efficiency: $\Theta(n \log n)$

In-place: yes

Stability: no (e.g., 1 1)

Priority Queue

A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:

- find element with highest priority
- delete element with highest priority
- insert element with assigned priority (see below)

Heap is a very efficient way for implementing priority queues

Two ways to handle priority queue in which highest priority = smallest number

Horner's Rule For Polynomial Evaluation

Given a polynomial of degree *n*

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

and a specific value of x, find the value of p at that point.

Two brute-force algorithms:

```
p ← 0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + a_i * power
return p
```

$\Theta(n^2)$, n^2 multiplications, n additions

```
p ← a_0;  power ← 1
for i ← 1 to n do
        power ← power * x
        p ← p + a_i * power
    return p
```

$\Theta(n)$ with 2n multiplications and n additions

Horner's Rule

Example: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$

$= x(2x^3 - x^2 + 3x + 1) - 5$

$= x(x(2x^2 - x + 3) + 1) - 5$

$= x(x(x(2x - 1) + 3) + 1) - 5$

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:

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

S*ynthetic division* of of $p(x)$ by $(x\text{-}x_0)$

Example: Let $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$. Find $p(x)$:$(x\text{-}3)$

Computing a^n (revisited)

Left-to-right binary exponentiation

Initialize product accumulator by 1.

Scan n's binary expansion from left to right and do the following:

If the current binary digit is 0, square the accumulator (S);

if the binary digit is 1, square the accumulator and multiply it by a (SM).

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

binary rep. of 13:		1	1	0	1
		SM	SM	S	SM
accumulator:	1	$1^2*a=a$	$a^2*a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2*a= a^{13}$

(computed left-to-right)

Efficiency: $\lfloor \log_2 n \rfloor \leq M(n) \leq 2\lfloor \log_2 n \rfloor$

Left-to-Right Binary Exponentiation

ALGORITHM *LeftRightBinaryExponentiation*$(a, b(n))$

//Computes a^n by the left-to-right binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_l, \ldots, b_0$
// in the binary expansion of a positive integer n
//Output: The value of a^n
$product \leftarrow a$
for $i \leftarrow I - 1$ **downto** 0 **do**
  $product \leftarrow product * product$
  if $b_i = 1$ $product \leftarrow product * a$
return $product$

Computing a^n (cont.)

Right-to-left binary exponentiation

Scan n's binary expansion from right to left and compute a^n as the product of terms a^{2^i} corresponding to 1's in this expansion.

Example Compute a^{13} by the right-to-left binary exponentiation.

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

Right-to-Left Binary Exponentiation

ALGORITHM *RightLeftBinaryExponentiation*$(a, b(n))$

//Computes a^n by the right-to-left binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_l, \ldots, b_0$
// in the binary expansion of a nonnegative integer n
//Output: The value of a^n
$term \leftarrow a$ //initializes a^{2^i}
if $b_0 = 1$ $product \leftarrow a$
else $product \leftarrow 1$
for $i \leftarrow 1$ **to** l **do**
    $term \leftarrow term * term$
    if $b_i = 1$ $product \leftarrow product * term$
return $product$

Horner's rule By-products

Self Study for a Bonus 3 marks: how to use Horner's rule as a division algorithm:

Hint: ***synthetic division***

Problem Reduction

This variation of transform-and-conquer solves a problem by transforming it into a 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.

Examples of Solving Problems by Reduction

- computing lcm(m, n) via computing gcd(m, n)
- counting number of paths of length n in a graph by raising the graph's adjacency matrix to the n-th power
- transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- linear programming
- reduction to graph problems (e.g., solving puzzles via state-space graphs)

Assignment 2

a. Describe an algorithm to heapify an array using the top down approach, and compare its performance to the bottom up algorithm discussed.
b. If the data structure of the heap is required to be a tree designed using pointers to left and right children, describe the bottom up heapify, insert a key, and delete a key algorithms, and analyse their efficiency.