CSC 8301- Design and Analysis of Algorithms

Lecture 8

Transform and Conquer II
Algorithm Design Technique

Transform and Conquer

This group of techniques solves a problem by a *transformation*

- **to a simpler/more convenient instance of the same problem (*instance simplification*)**

- **to a different representation of the same instance (*representation change*)**

- **to a different problem for which an algorithm is already available (*problem reduction*)**

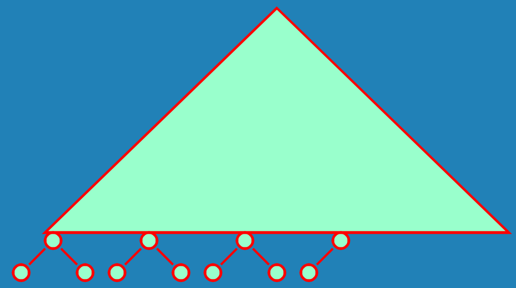
Representation Change

- Search Trees (binary, AVL, 2-3, 2-3-4, B-trees)
- Heaps
- Horner's rule for polynomial evaluation
- Computing a^n (binary exponentiation)

Heaps and Heapsort

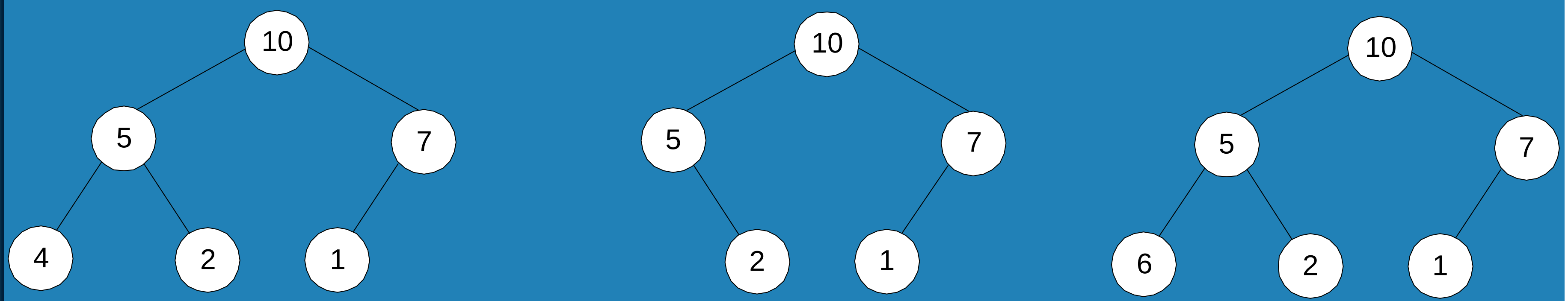
Definition **A *heap* is a binary tree with keys at its nodes (one key per node) such that:**

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



- **The key at each node is ≥ keys at its children (*heap property*)**

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

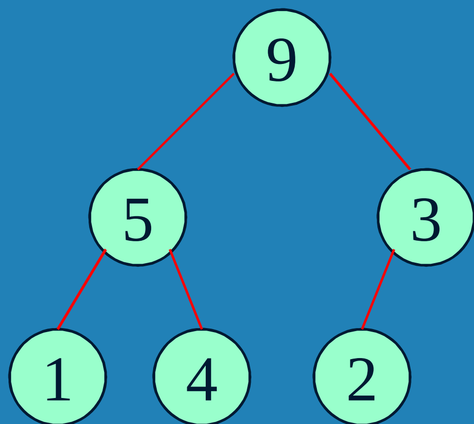
Some Important Properties of a Heap

- 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:



1	2	3	4	5	6
9	5	3	1	4	2

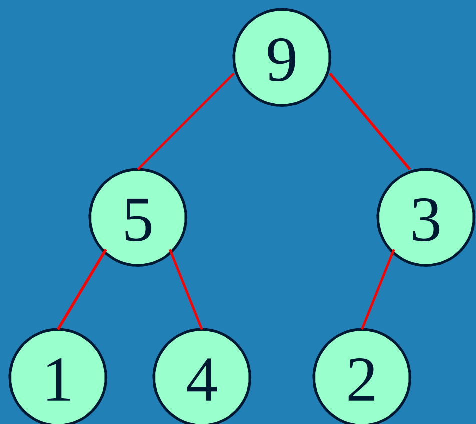
First level : index 1
2nd level : indices 2,3
3rd level : indices 4,5,6

- **Left child of node *j* is at** 2j ______

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:



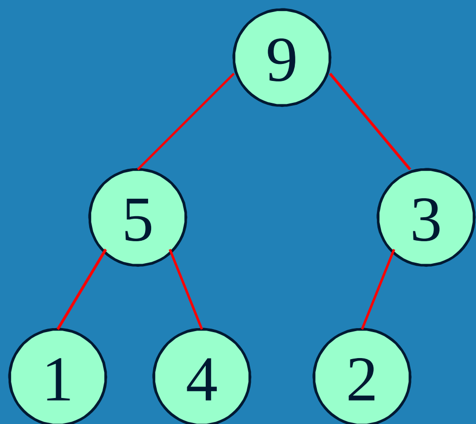
1	2	3	4	5	6
9	5	3	1	4	2

- **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 ________** $\lfloor j/2 \rfloor$

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:



1	2	3	4	5	6
9	5	3	1	4	2

- **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)

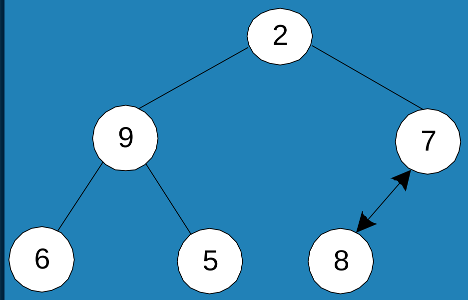
Step 0: Initialize the structure with keys in the order given

Step 1: (Heapify) 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

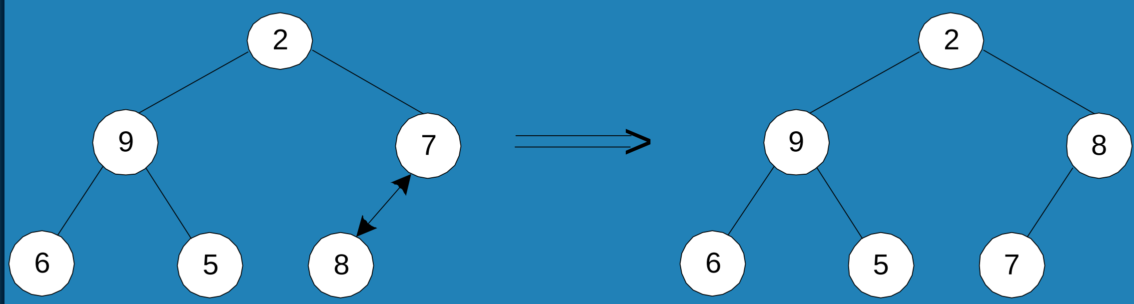
Example of Bottom-up Heap Construction

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



Example of Bottom-up Heap Construction

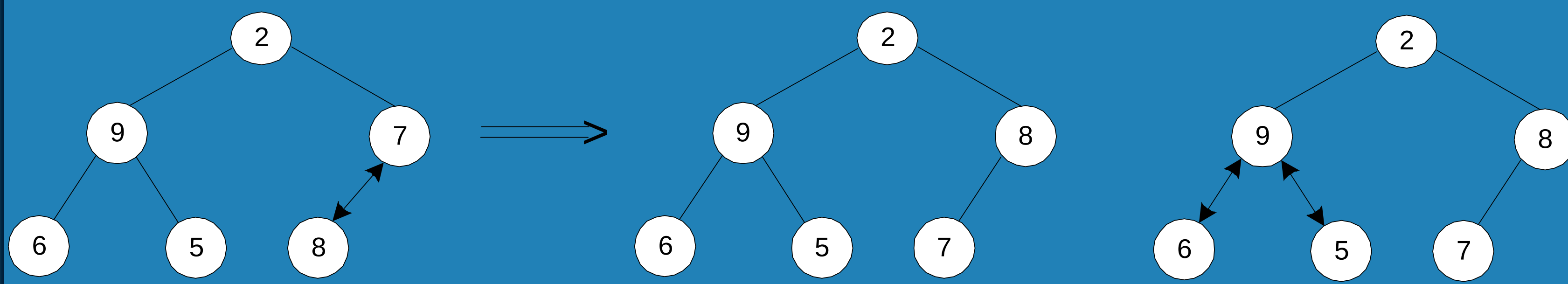
Construct a heap for the list 2, 9, 7, 6, 5, 8



2, 9, 8, 6, 5, 7

Example of Bottom-up Heap Construction

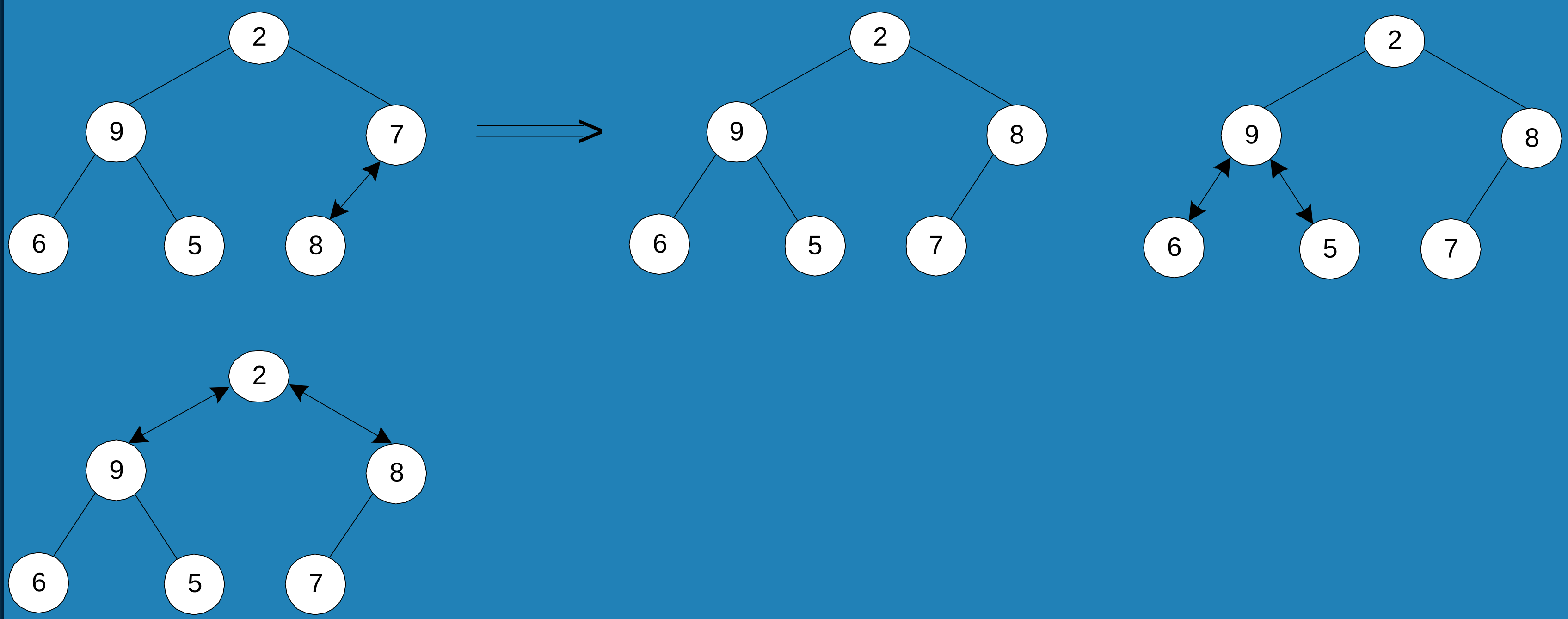
Construct a heap for the list 2, 9, 7, 6, 5, 8



$\boxed{2}$ 9, 8, 6, 5, 7

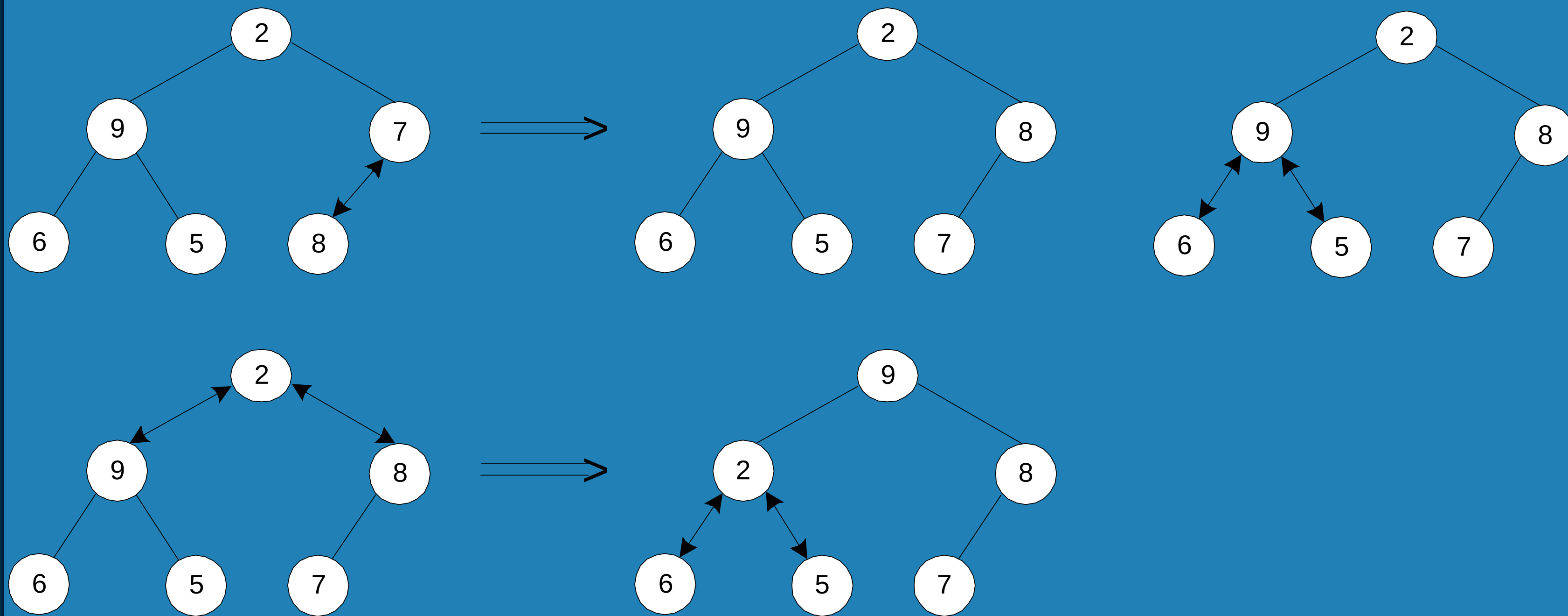
Example of Bottom-up Heap Construction

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



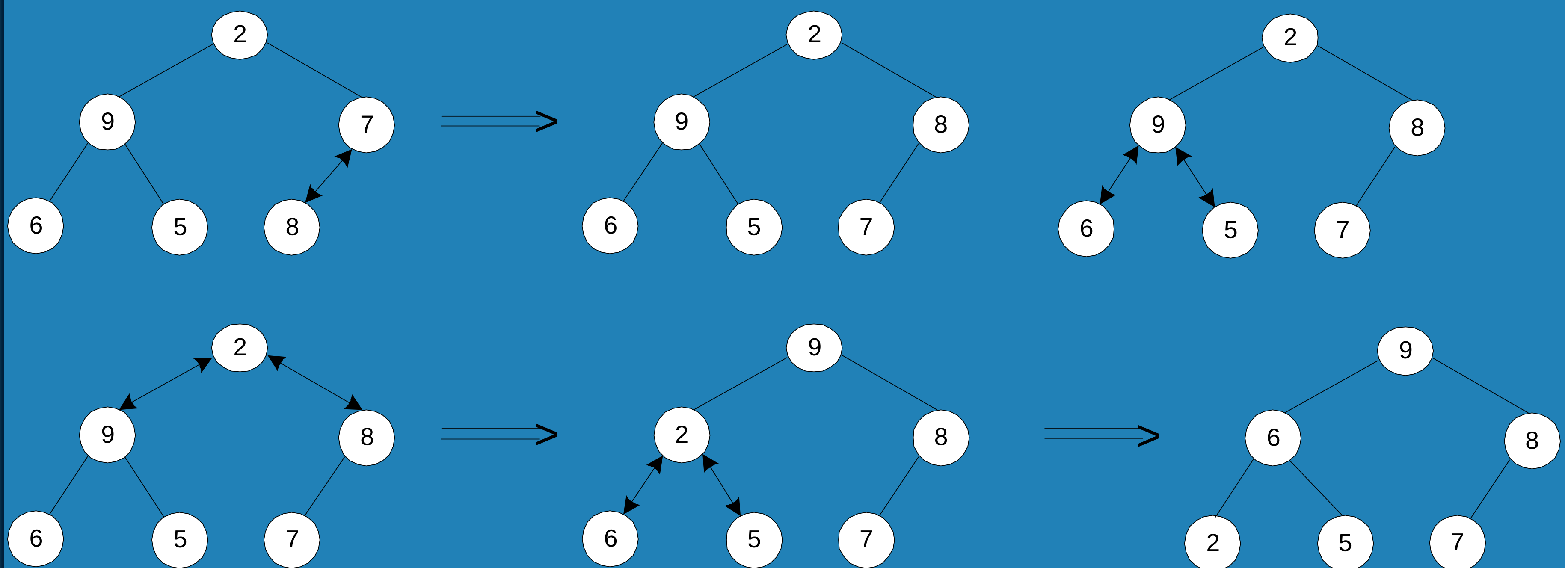
Example of Bottom-up Heap Construction

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



Example of Bottom-up Heap Construction

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



$$C(n) = \sum_{\ell=0}^{h-1} 2^{\ell} * 2(h-\ell) < 2n$$

Bottom-up Heap Construction Algorithm

Construct a heap for the list

1	2	3	4	5	6
2	9	7	6	5	8

H

$\underbrace{2\ 9\ 7}_{\text{parent nodes}}$

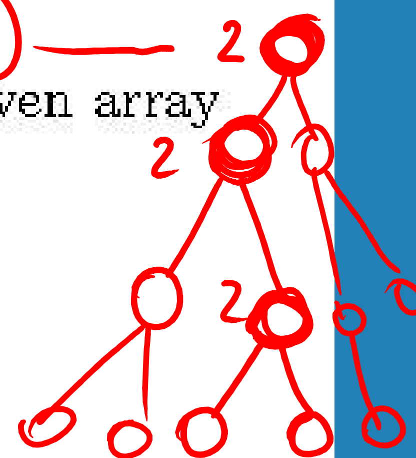
```
for i = ⌊n/2⌋ downto 1
    k = i
    while (2*k <= n)
        j ← 2*k
        if (j<n) and H[j]<H[j+1]
            j++;
        if (H[k] > H[j])
            break;
        swap(H[k], H[j]);
        k ← j;
```

```
for i = ⌊n/2⌋ downto 1
    Heapify(H, i)
```

```
Heapify(H, i)
    j ← 2*i
    if ( j > n) return
    if ( j < n) and H[j] < H[j+1]
        j++;
    if (H[i] > H[j]) return;
    swap(H[i], H[j]); Heapify(H, j);
```

Pseudopodia of Bottom-up Heap Construction

Algorithm $HeapBottomUp(H[1..n])$ level ℓ



//Constructs a heap from the 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 $H[j] < H[j+1]$ $j \leftarrow j + 1$ 1
        if $v \ge H[j]$ 2
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]$; $k \leftarrow j$
    $H[k] \leftarrow v$

Each key on level ℓ needs to travel distance $h-\ell$ in the worst case

$$C(n) = \sum_{\ell=0}^{h-1} 2^{\ell} * 2(h-\ell) < 2n$$

Heapsort

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

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

- **Exchange keys in the root and in the last (rightmost) leaf**
- **Decrease heap size by 1**
- **“Heapify” the tree: if necessary, swap new root with larger child until the heap condition holds**

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 $\boxed{9}$ 8 6 5 7

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

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

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

1	2	3	4	5	6
2	9	7	6	5	8

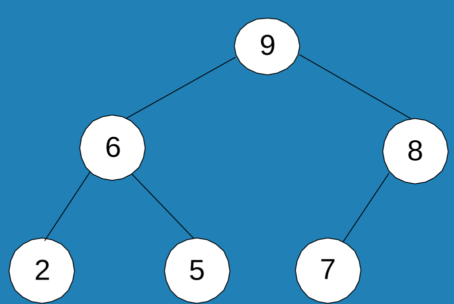
Example of Sorting by Heapsort

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

1	2	3	4	5	6
9	6	8	2	5	7

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 (root/max removal)

9 6 8 2 5 7

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

8 6 7 2 5

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

7 6 5 2

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

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

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

2 5 6 7 8 9

Example of Sorting by Heapsort

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

Stage 1 (heap construction)

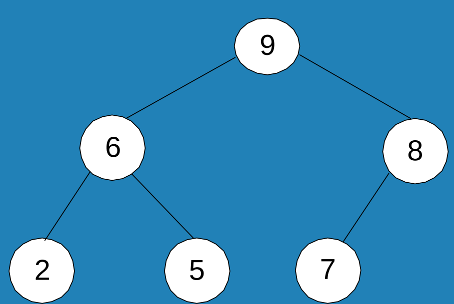
2 9 7 6 5 8

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 (root/max removal)

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5 | 9

5 6 7 2 | 8 9

7 6 5 2 | 8 9

2 6 5 | 7 8 9

6 2 5 | 7 8 9

5 2 | 6 7 8 9

5 2 | 6 7 8 9

2 | 5 6 7 8 9

Analysis of Heapsort

$h \approx \log_2 n$

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

worst-case

Key on level ℓ requires $2(h-\ell)$ comparisons

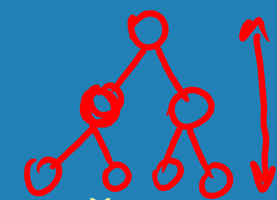
$$C(n) = \sum_{\ell=0}^{h-1} 2^{\ell} \cdot 2(h-\ell) < 2n$$

Analysis of Heapsort

All levels full:

$1 + 2 + 2^2 + \dots + 2^{k-1} = 2^k - 1 = n$

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



height 2

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*

$2^k - 1 = n$

$2^k = n+1$

$k = \log_2(n+1)$

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

$h = (k-1) = \log_2(n+1) - 1$

$h \approx \log_2 n$

worst-case

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

Removing 1^{st} root: $2 \cdot \log(n-1)$ comparisons to "heapify" $n-1$ nodes

2^{nd} root: $2\log(n-2)$, ...

Analysis of Heapsort

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

worst-case

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

\# 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)

Review of Major Sorting Algorithms

	Selection sort	Insertion sort	Mergesort	Quicksort	Heapsort
strategy					
worst time	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
avg. time					
in-place			NO		YES
stability					

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)

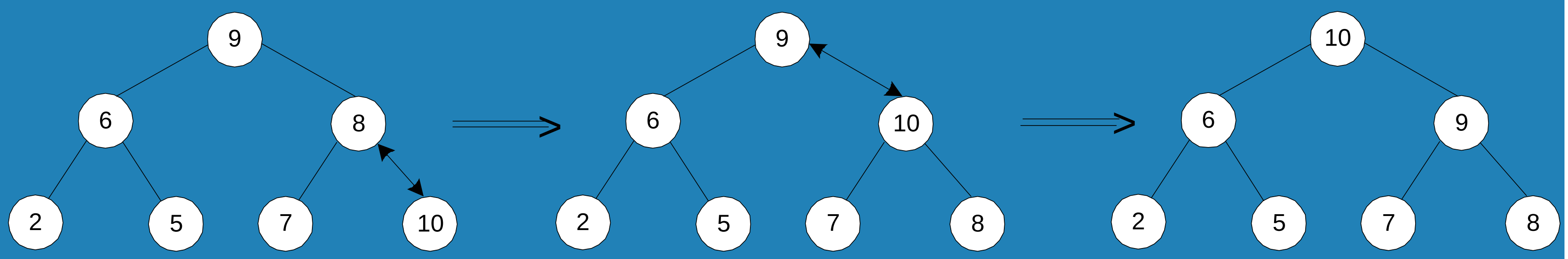
- Heap is a very efficient way for implementing priority queues

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

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*)

Representation Change

- Search Trees (binary, AVL, 2-3, 2-3-4, B-trees)
- Heaps
- Horner's rule for polynomial evaluation
- Computing a^n (binary exponentiation)

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

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

Horner's Rule

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

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

$$= x(x\underbrace{(2x^2 - x + 3)} + 1) - 5$$

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

2 -1 3 1 -5

Horner's Rule

Example: $p(\mathrm{x}) = 2x^4 - x^3 + 3x^2 + x - 5 =$

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

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

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

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		$3*2-1$	$3*5+3$	$3*18+1$	$3*55-5$
		5	18	55	160

$\boxed{p(3) = 160}$

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

$2x^4 - x^3 + 3x^2 + x - 5$

	0	1	2	3	4
P	-5	1	3	-1	2

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 $p(x)$:$(x-3)$

$3*2-1$ 5 $3*5+3$ 18 $3*18+1$ 55

$\frac{p(x)}{x-3} = 2x^3 + 5x^2 + 18x + 55$

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	$\frac{1^2 * a}{= a}$	$\frac{a^2 * a}{a^3}$	$\frac{(a^3)^2}{a^6}$	$\frac{(a^6)^2 * a}{a^{13}}$

Computing a^n (revisited)

<u>Left-to-right binary exponentiation</u>

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

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		
a^8	a^4	a^2	a	:	a^{2^i} terms

term = term * term

result = a if bit0 = 1
result = 1 if bit0 = 0
for i = 1 to log n
term = term * term
if bit = 1
result = result * term

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**		
a^8		a^4		a^2	a	:	a^{2^i} **terms**
a^8	*	a^4	*		a	:	**product**

(computed right-to-left)

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

Transform and Conquer
Problem Reduction

Problem Reduction

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

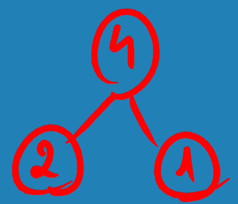
Examples of Solving Problems by Reduction

- computing lcm(m, n) via computing gcd(m, n)

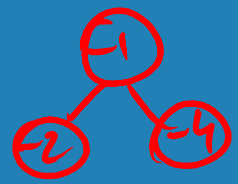
  $\boxed{lcm(m,n)} * gcd(m,n) = n * m$

- 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)

  max-heap on -4, -2, -1

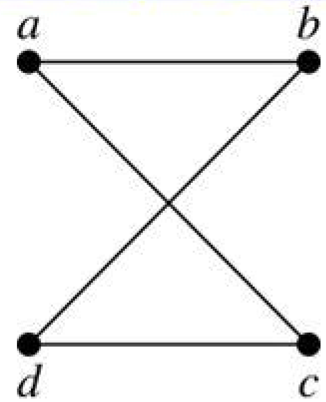
  

- linear programming

- reduction to graph problems (e.g., solving puzzles via state-space graphs)

Example: How many paths of length four are there from *a* to *d* in the graph *G*.

G



adjacency matrix A of G
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$\mathbf{A}^4 = \begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix}$$

Solution: The adjacency matrix of *G* is given above. Hence the number of paths of length four from *a* to *d* is the $(1, 4)$th entry of $\mathbf{A}^4$. The eight paths are as:

a, b, a, b, d	*a, b, a, c, d*
a, b, d, b, d	*a, b, d, c, d*
a, c, a, b, d	*a, c, a, c, d*
a, c, d, b, d	*a, c, d, c, d*

Homework

Read Sec. 6.4, 6.5, and 6.6

Exercises 6.4: 3, 6, 7, 8

Exercises 6.5: 4, 7, 9

Exercises 6.6: 2, 9