DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

Lecture 25, 26

Transform-and-Conquer:
Heaps and Heap sort
Problem Reduction

Instructors:

Dr. Savitha G,
Assistant Professor, DSCA, MIT, Manipal

Dr. Abhilash K. Pai,
Assistant Professor, DSCA, MIT, Manipal

Recap of L24

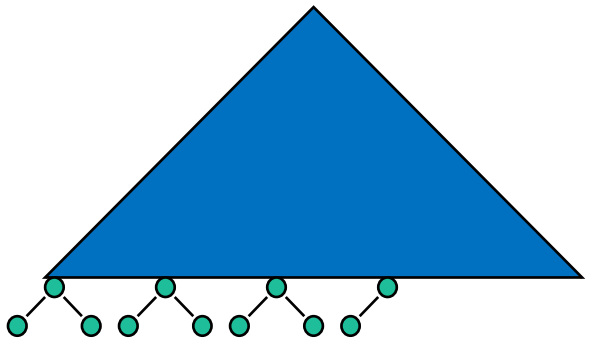
- AVL Trees

- 2-3 Trees

Heaps and Heapsort

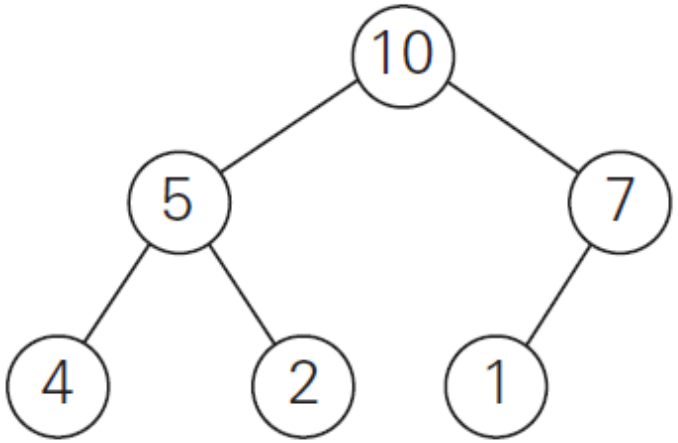
Definition : A *heap* is a binary tree with keys at its nodes (one key per node) such that:

1. It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing
2. The key at each node is ≥ keys at its children (this is called a *max-heap*)

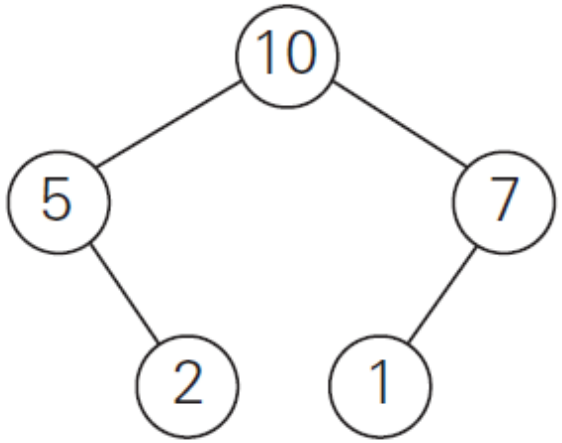


- Heaps are suitable for implementing priority queues.
- Applications:
  - Job scheduling in operating systems
  - Traffic management in communication networks
  - Prim's and Kruskal's algorithms, Huffman Coding
  - Sorting

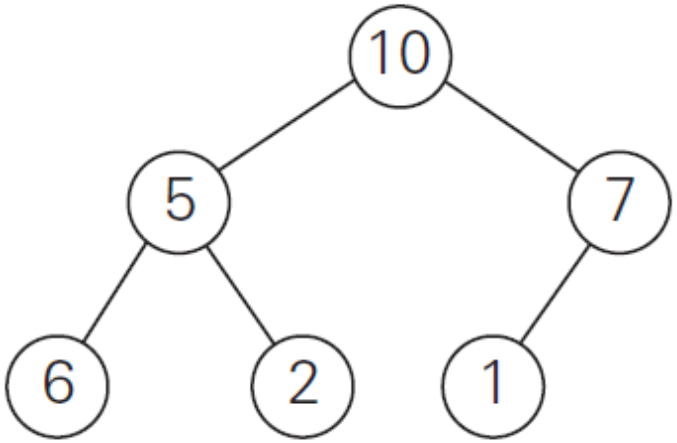
Illustration of the heap's definition



a heap



not a heap



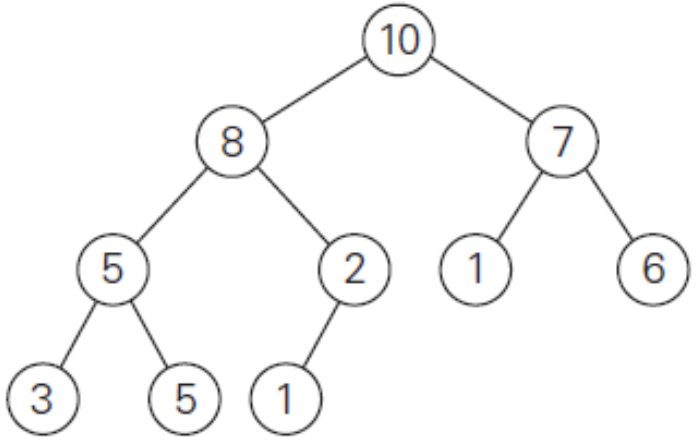
not a heap

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

1. There exists exactly one essentially complete binary tree with n nodes.
2. Its height is equal to $\lfloor \log_2 n \rfloor$
3. The root of a heap always contains its largest element.
4. A node of a heap considered with all its descendants is also a heap.
5. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion.

Heap's Array Representation



- 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 ⌊*j*/2⌋**
- Parental nodes are represented in the **first ⌊*n*/2⌋ locations**

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				

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

Heap Construction (bottom-up)

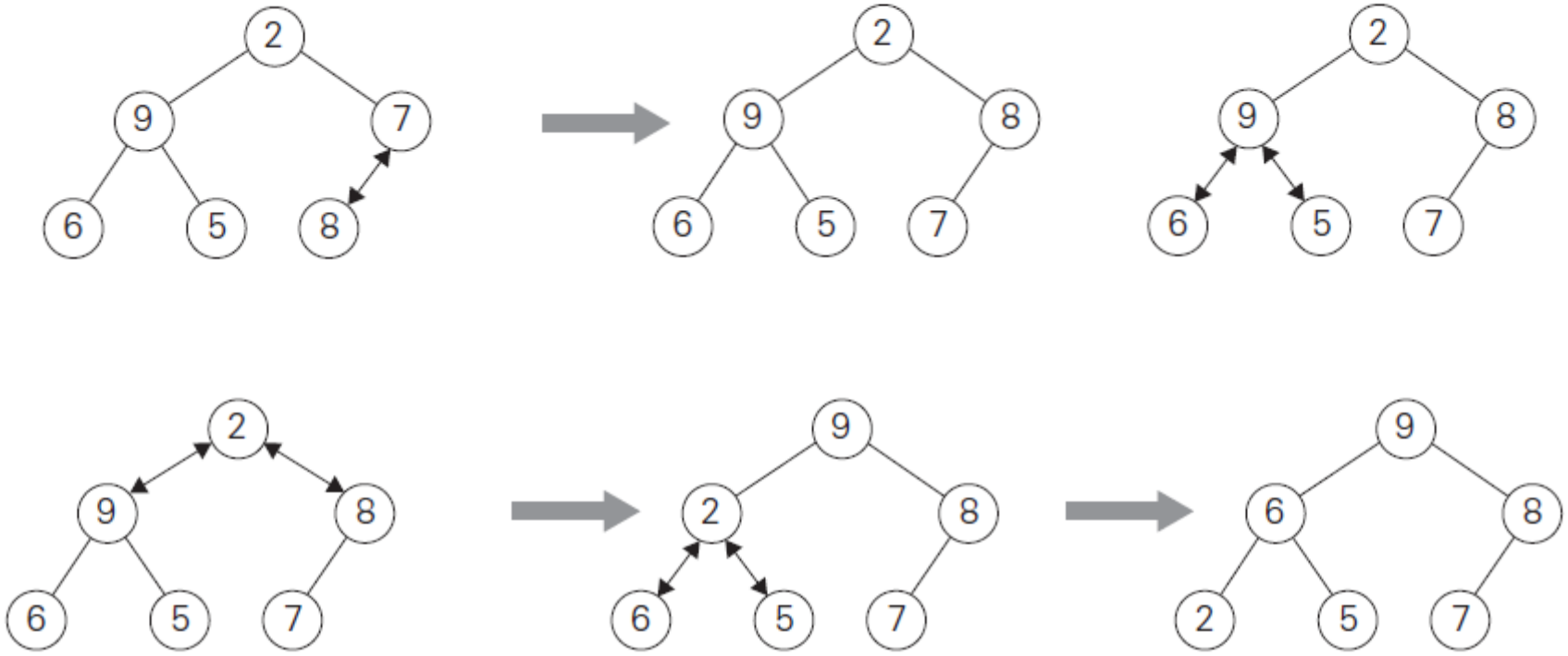
- **Step 0:** Initialize the structure with keys in the order given

- **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 larger child until the heap condition holds

- **Step 2:** Repeat Step 1 for the preceding parental node

Heap Construction (bottom-up)

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



Pseudocode 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
```

- In the worst case, each key on level ***i*** of the tree will travel to the leaf level ***h.***
- Moving to the next level down requires **two comparisons.**
- The total number of key comparisons involving a key on level ***i*** will be ***2(h – i)***

The total no. of key comparisons in the worst case :

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

i=0 to h-1

O(n)

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

Hint: For a binary tree of n nodes and height h
$\mathbf{n = 2^{h+1} - 1}$

Deletion of root (maximum key) in a heap

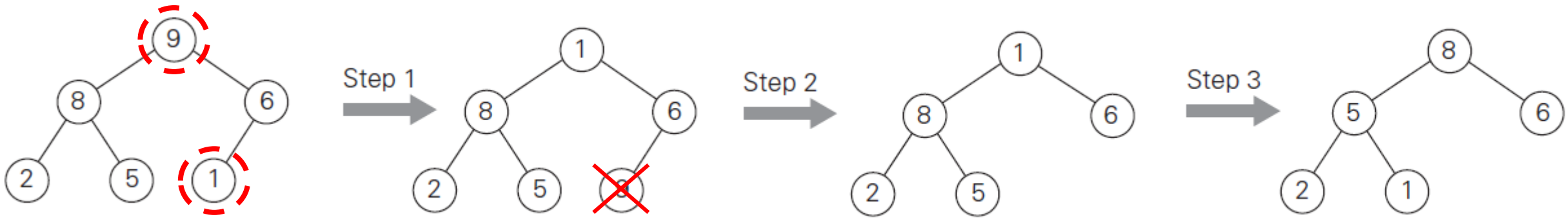
Maximum Key Deletion from a heap

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.

- Deletion depends on the height of the tree
- Efficiency: **O(log *n*)**



Heap Sort

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

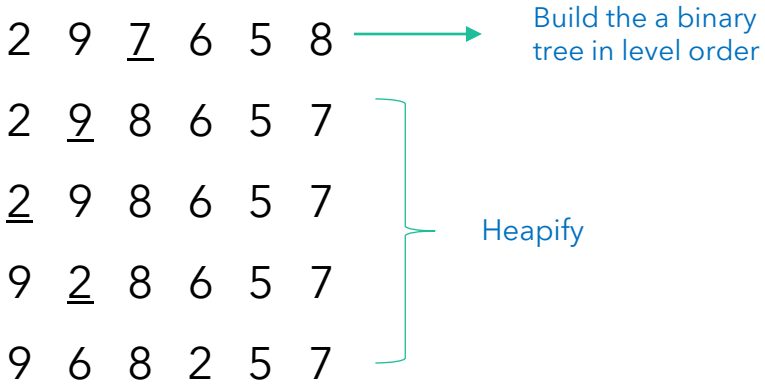
Stage 2: Repeat operation of root removal *n*-1 times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds

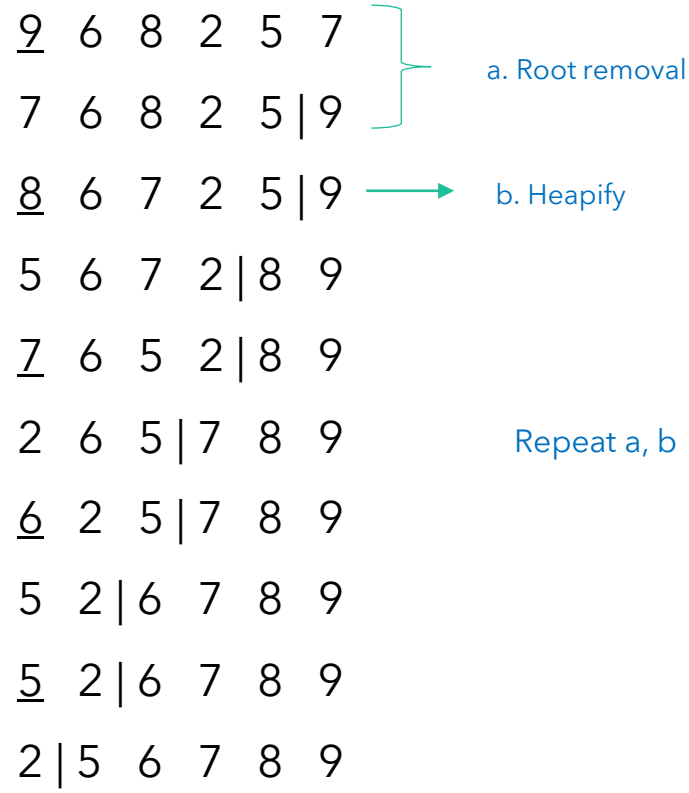
Heap Sort

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

Stage 1 (heap construction)



Stage 2 (root/max removal)



Heap Sort : Analysis

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

Worst-case

$C_1(n) = 2\,(n - \log_2(n+1)) \in O(n)$

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

Worst-case

$C_2(n) <= \sum_{i=1}^{n-1} 2\log_2 i \in O(n\log n)$

Proof

$$C_2(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$
$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

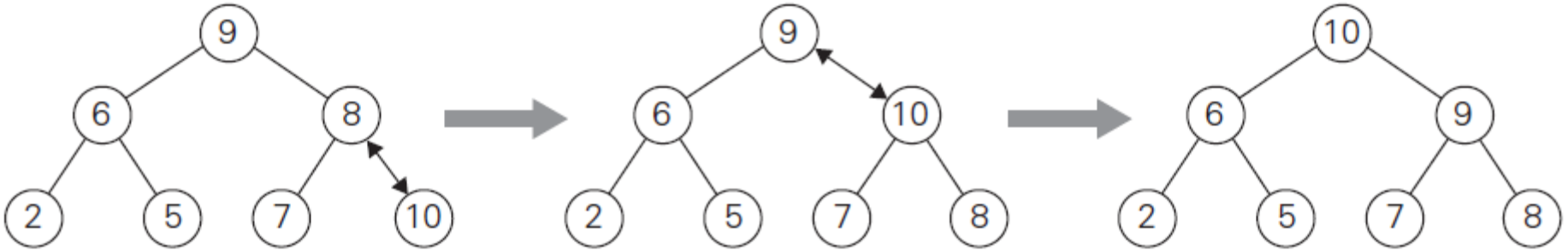
$C(n) = C_1(n) + C_2(n) = O(n\log n)$

- Both worst-case and average-case efficiency: **O(nlogn)**

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: Inserting a key 10



- Efficiency: **O(log *n*)**

Problem Reduction

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

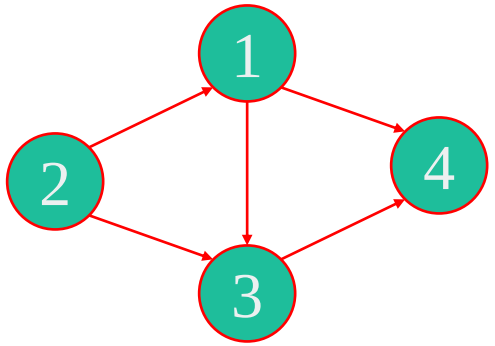
- To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

Examples of Solving Problems by Reduction

1. **Computing lcm(*m*, *n*)** via computing gcd(*m*, *n*)

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

2. **Counting number of paths of length *n* in a graph** by raising the graph's adjacency matrix to the *n*-th power



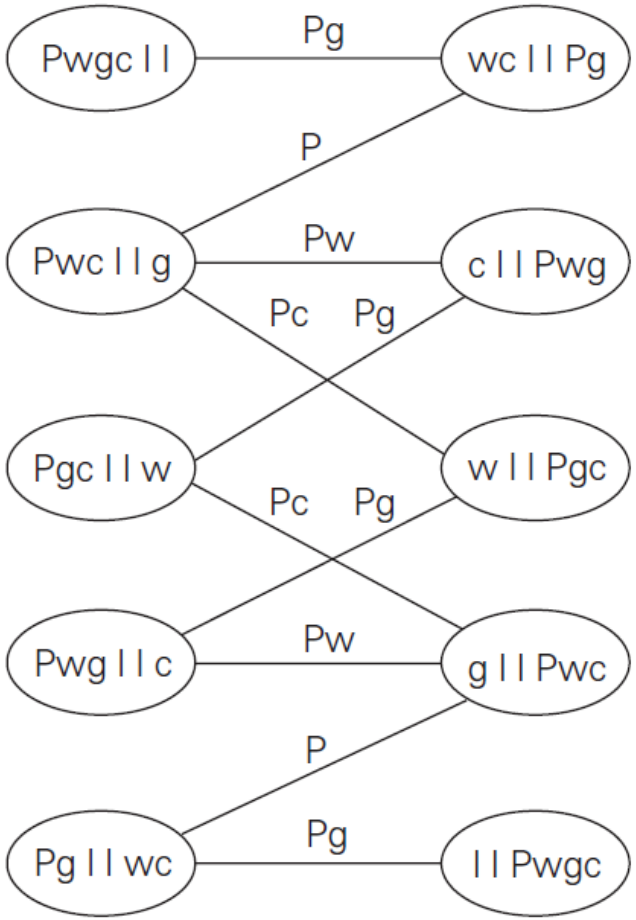
$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Examples of Solving Problems by Reduction

3. Reduction to graph problems

Problem: A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room only for the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Find a way for the peasant to solve his problem.

Examples of Solving Problems by Reduction



State-space graph for the peasant, wolf, goat, and cabbage puzzle.

Thank you!

Any queries?