

# Final Exam

- Thursday, 7 August 2014, 19:00 – 22:00
- Closed Book
- Will cover whole course, with emphasis on material after midterm (hash tables, binary search trees, sorting, graphs)

# Suggested Study Strategy

- Review and understand the slides.
- Read the textbook, especially where concepts and methods are not yet clear to you.
- Do all of the practice problems provided.
- Do extra practice problems from the textbook.
- Review the midterm and solutions for practice writing this kind of exam.
- Practice writing clear, succinct pseudocode!
- Review the assignments
- See me or one of the TAs if there is anything that is still not clear.

# Assistance

- Regular office hours will not be held
- You may reach me by email / Skype

# End of Term Review

# Summary of Topics

1. Binary Search Trees
2. Sorting
3. Graphs

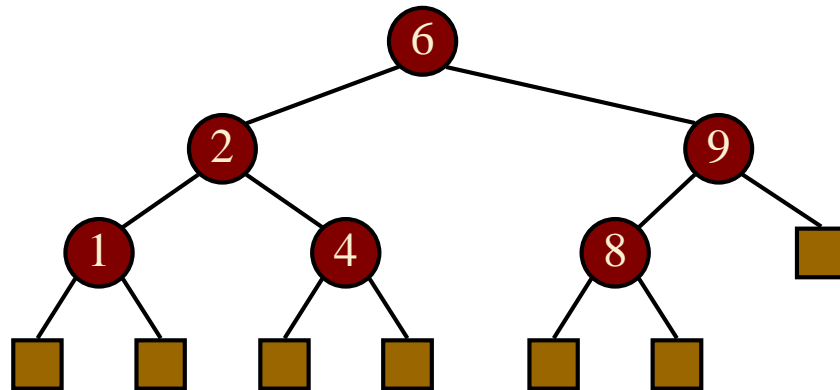
# Topic 1. Binary Search Trees

# Binary Search Trees

- Insertion
- Deletion
- AVL Trees
- Splay Trees

# Binary Search Trees

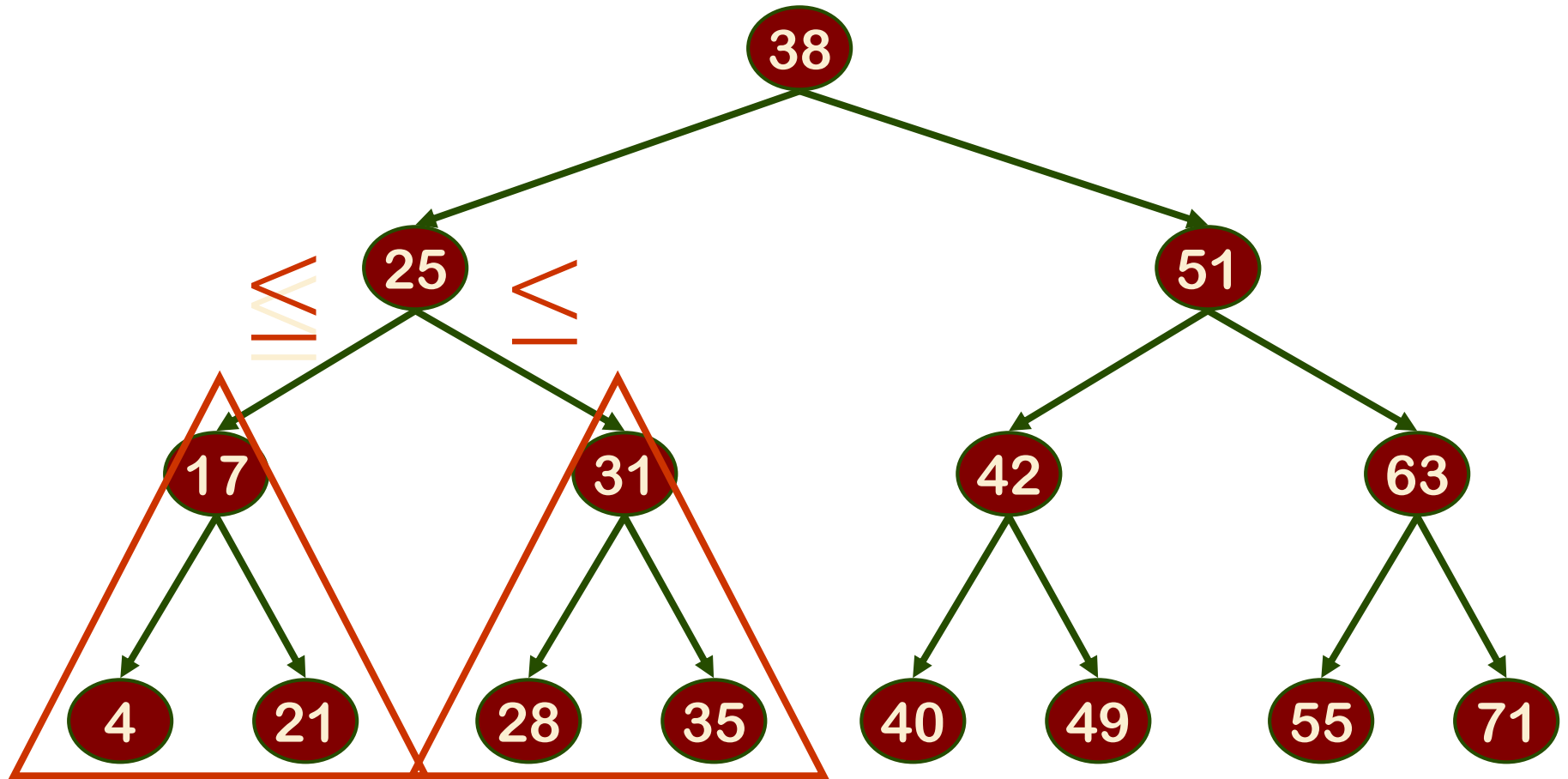
- A binary search tree is a binary tree storing key-value entries at its internal nodes and satisfying the following property:
  - Let  $u$ ,  $v$ , and  $w$  be three nodes such that  $u$  is in the left subtree of  $v$  and  $w$  is in the right subtree of  $v$ . We have  $key(u) \delta key(v) \delta key(w)$
- The textbook assumes that external nodes are 'placeholders': they do not store entries (makes algorithms a little simpler)
- An inorder traversal of a binary search tree visits the keys in increasing order
- Binary search trees are ideal for maps or dictionaries with ordered keys.





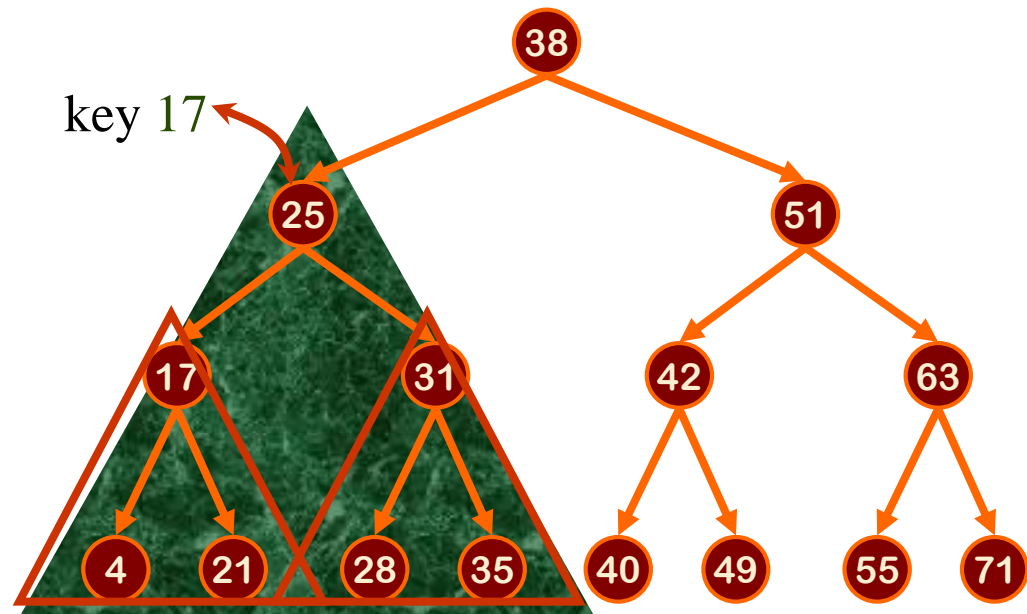
# Binary Search Tree

All nodes in left subtree  $\leq$  Any node  $\leq$  All nodes in right subtree



# Search: Define Step

- Cut sub-tree in half.
- Determine which half the key would be in.
- Keep that half.



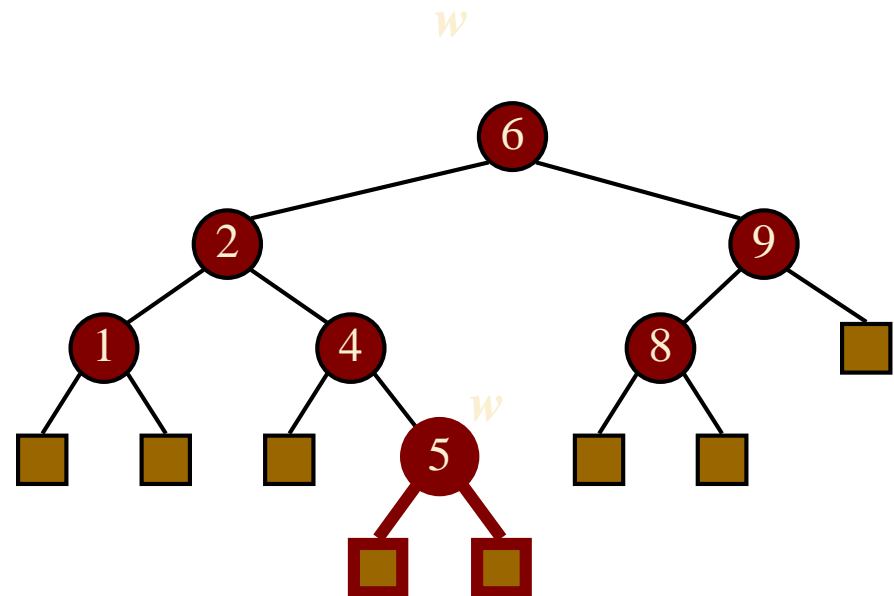
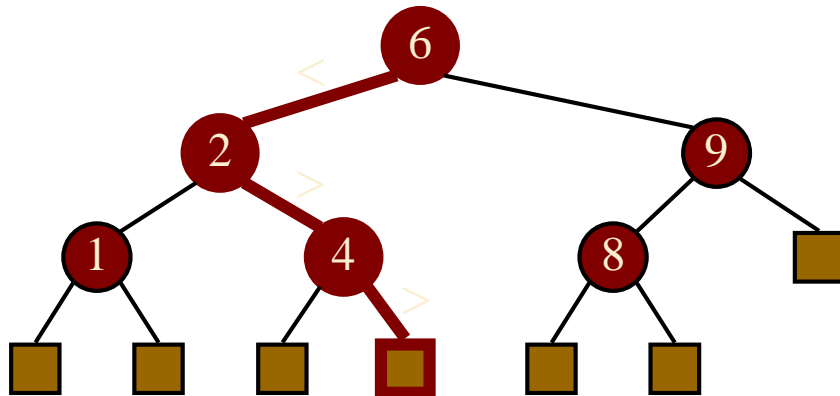
If  $\text{key} < \text{root}$ ,  
then key is  
in left half.

If  $\text{key} = \text{root}$ ,  
then key is  
found

If  $\text{key} > \text{root}$ ,  
then key is  
in right half.

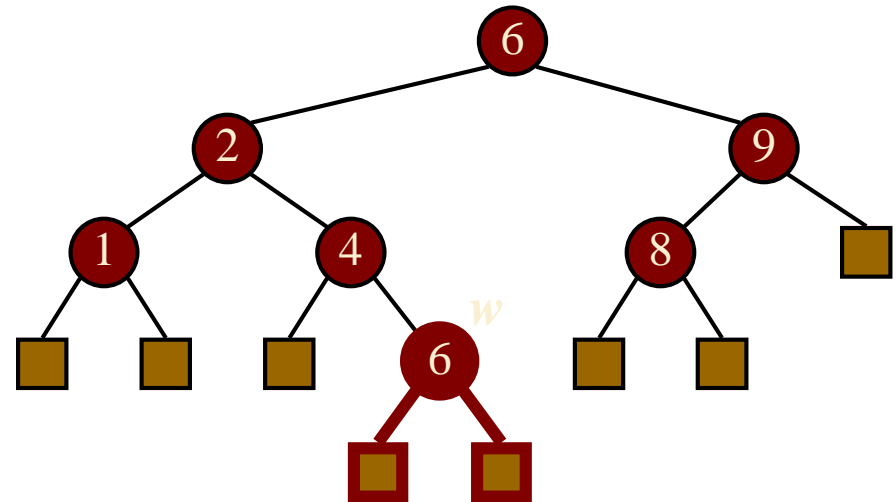
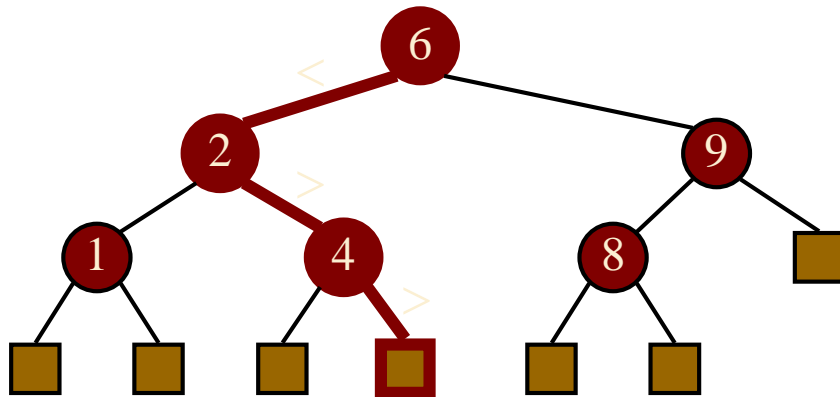
# Insertion (For Dictionary)

- To perform operation **insert**( $k$ ,  $o$ ), we search for key  **$k$**  (using TreeSearch)
- Suppose  **$k$**  is not already in the tree, and let  **$w$**  be the leaf reached by the search
- We insert  **$k$**  at node  **$w$**  and expand  **$w$**  into an internal node
- Example: insert 5



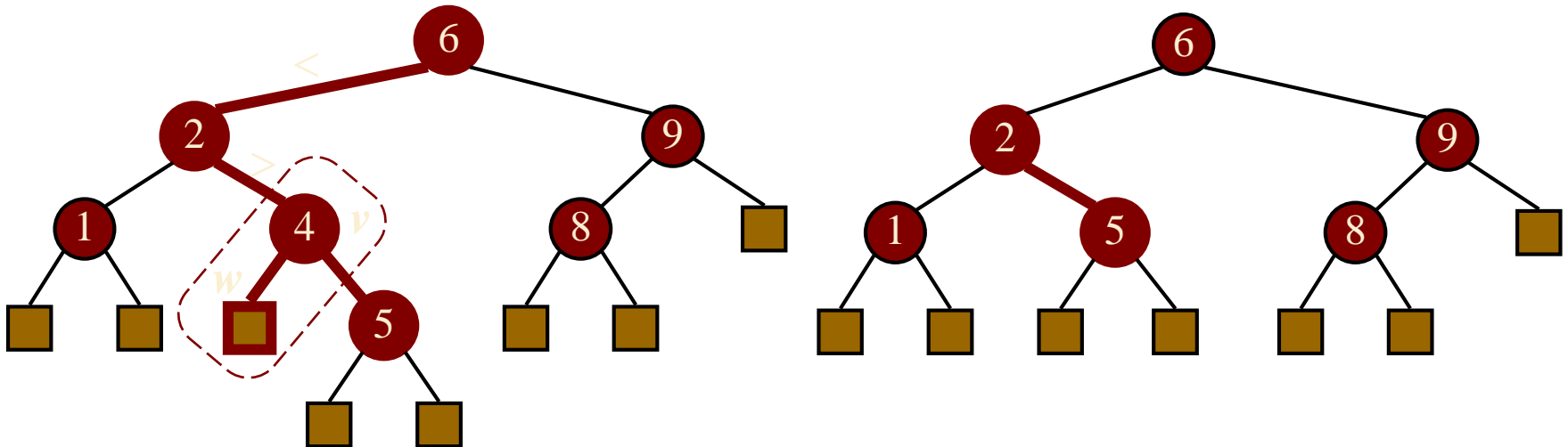
# Insertion

- Suppose **k** is already in the tree, at node **v**.
- We continue the downward search through **v**, and let **w** be the leaf reached by the search
- Note that it would be correct to go either left or right at **v**. We go left by convention.
- We insert **k** at node **w** and expand **w** into an internal node
- Example: insert 6



# Deletion

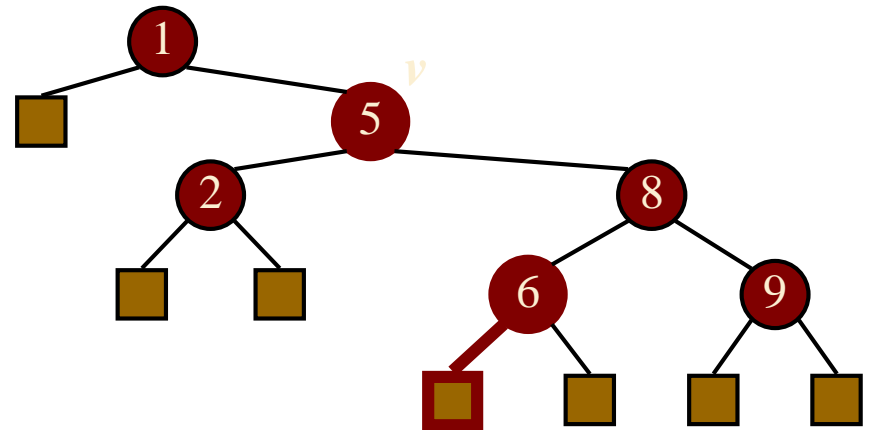
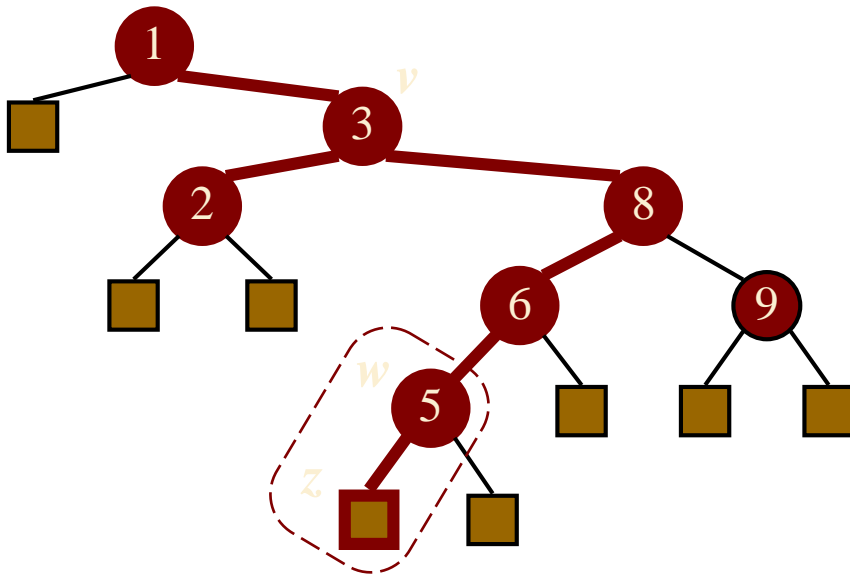
- To perform operation **remove**( $k$ ), we search for key  $k$
- Suppose key  $k$  is in the tree, and let  $v$  be the node storing  $k$
- If node  $v$  has a leaf child  $w$ , we remove  $v$  and  $w$  from the tree with operation **removeExternal**( $w$ ), which removes  $w$  and its parent
- Example: remove 4



# Deletion (cont.)

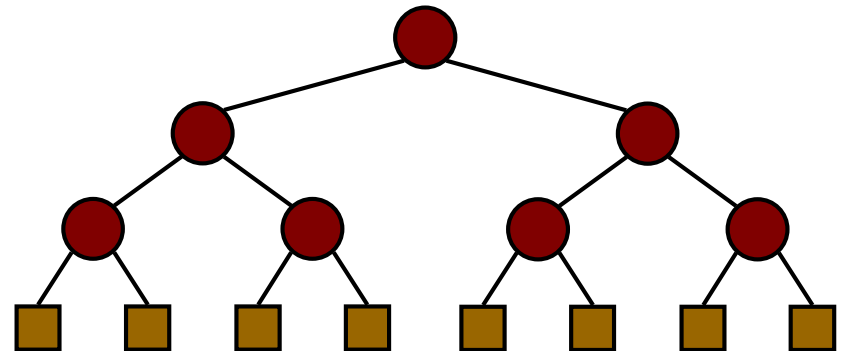
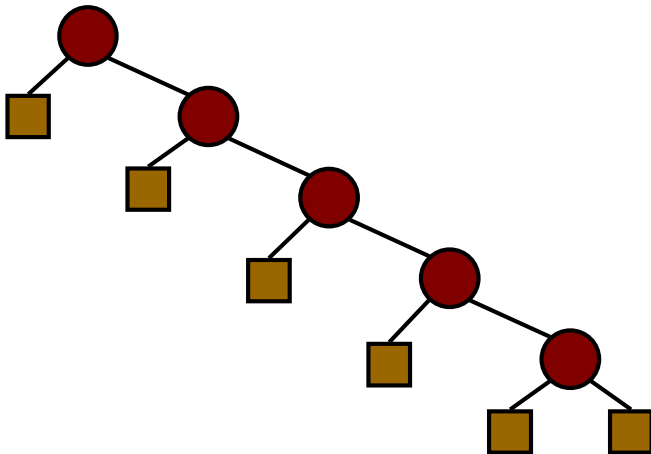
- Now consider the case where the key  $k$  to be removed is stored at a node  $v$  whose children are both internal
  - ❑ we find the internal node  $w$  that follows  $v$  in an inorder traversal
  - ❑ we copy the entry stored at  $w$  into node  $v$
  - ❑ we remove node  $w$  and its left child  $z$  (which must be a leaf) by means of operation **removeExternal**( $z$ )

➤ Example: remove 3



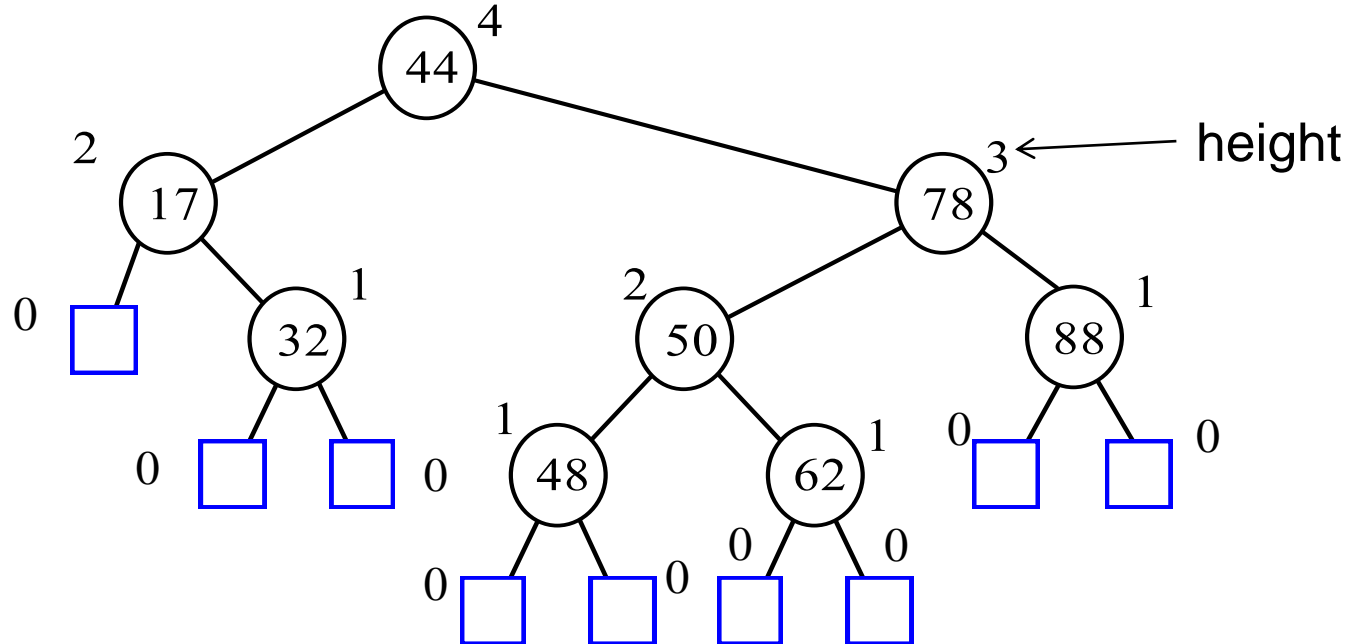
# Performance

- Consider a dictionary with  $n$  items implemented by means of a binary search tree of height  $h$ 
  - ❑ the space used is  $O(n)$
  - ❑ methods **find**, **insert** and **remove** take  $O(h)$  time
- The height  $h$  is  $O(n)$  in the worst case and  $O(\log n)$  in the best case
- It is thus worthwhile to balance the tree (next topic)!



# AVL Trees

- **AVL trees are balanced.**
- An AVL Tree is a **binary search tree** in which the heights of siblings can differ by at most 1.



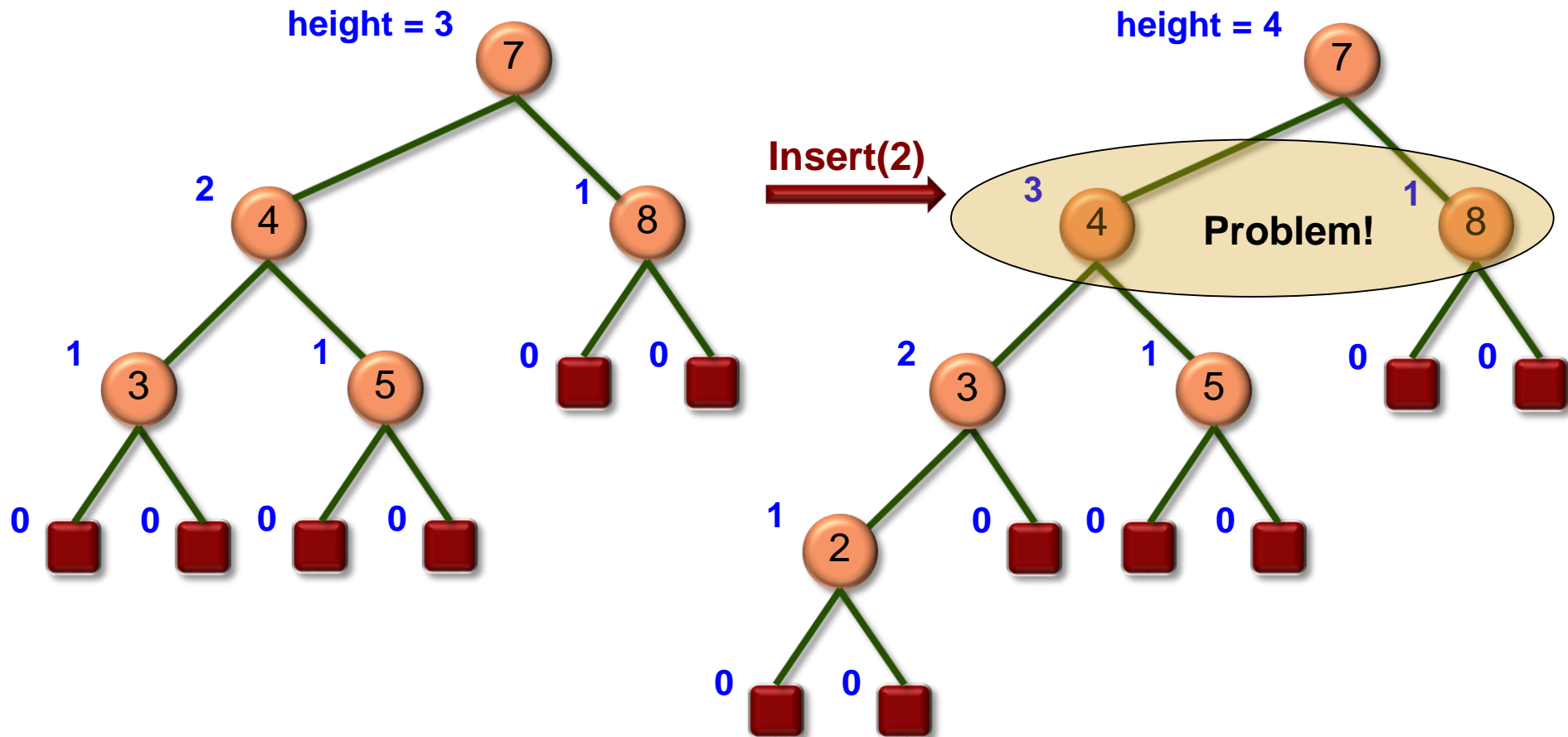


# Height of an AVL Tree

- **Claim:** The *height* of an AVL tree storing  $n$  keys is  $O(\log n)$ .

# Insertion

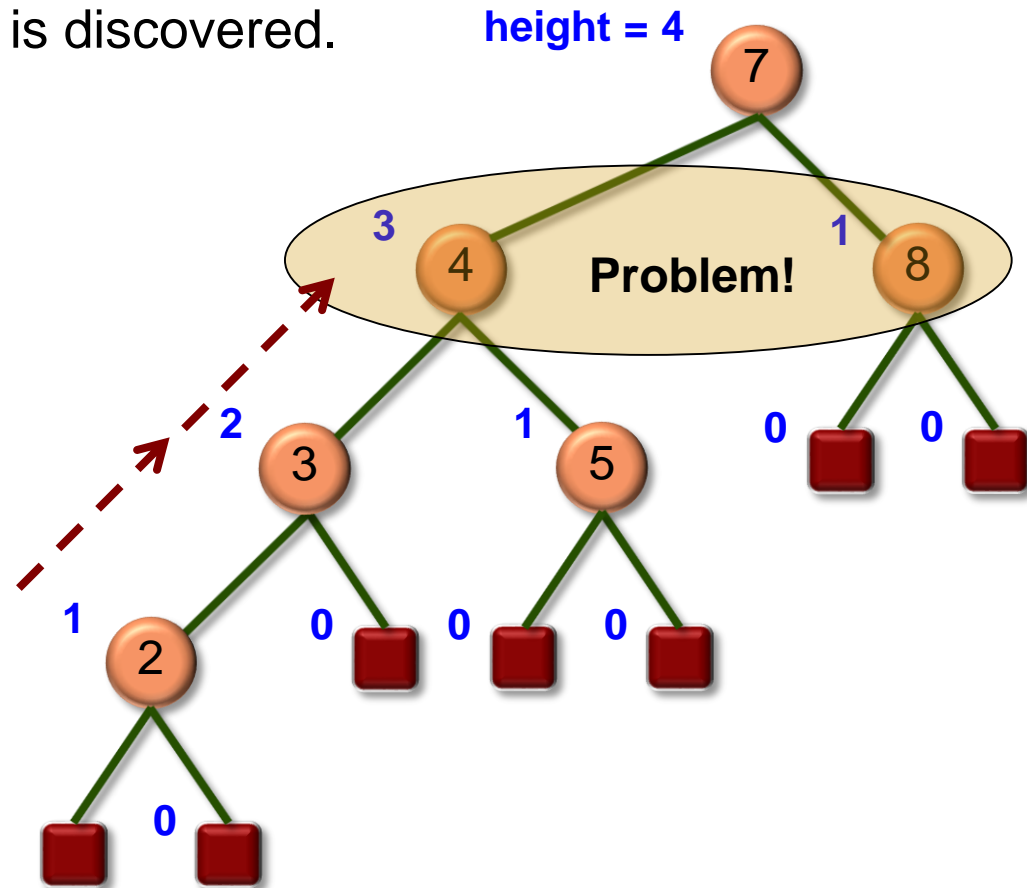
- Imbalance may occur at any ancestor of the inserted node.



# Insertion: Rebalancing Strategy

## ➤ Step 1: Search

- Starting at the inserted node, traverse toward the root until an imbalance is discovered.



# Insertion: Rebalancing Strategy

## ➤ Step 2: Repair

❑ The repair strategy is called **trinode restructuring**.

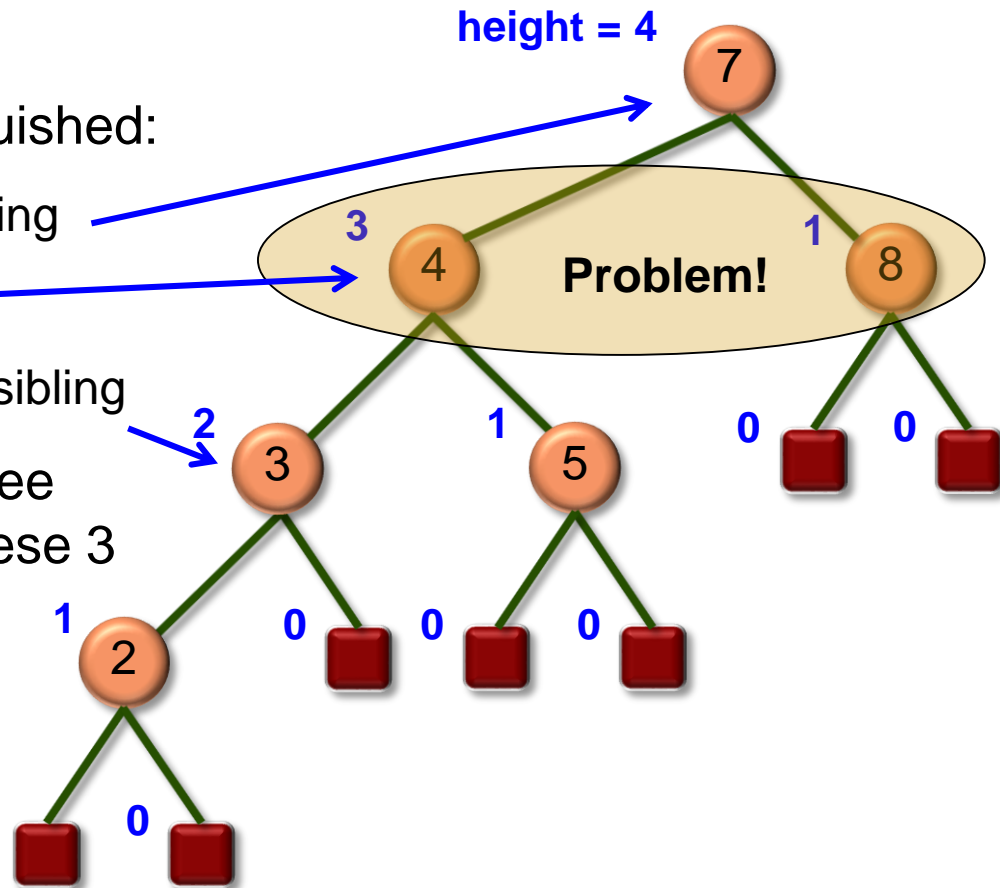
❑ 3 nodes x, y and z are distinguished:

✧ z = the parent of the high sibling

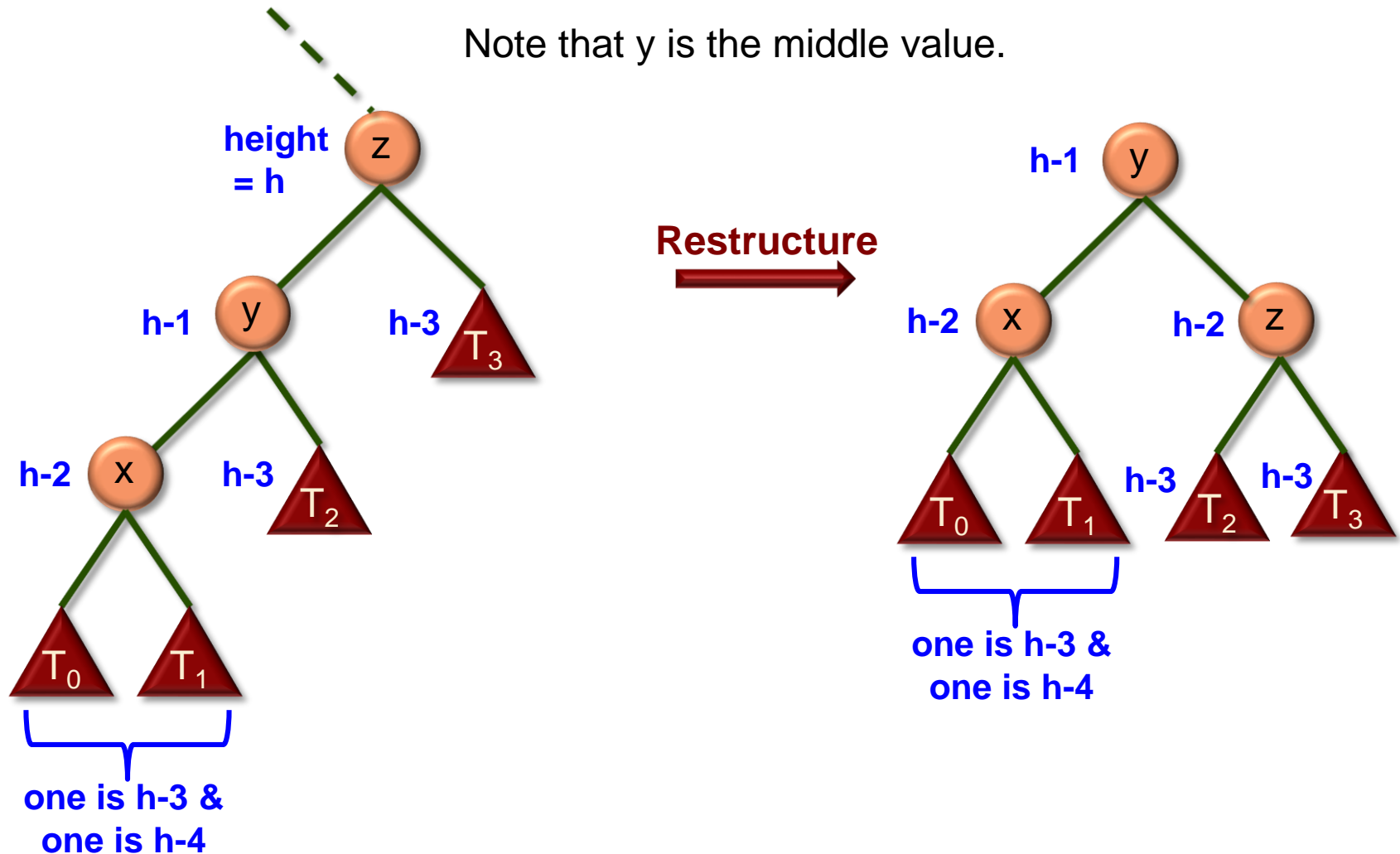
✧ y = the high sibling

✧ x = the high child of the high sibling

❑ We can now think of the subtree rooted at z as consisting of these 3 nodes plus their 4 subtrees

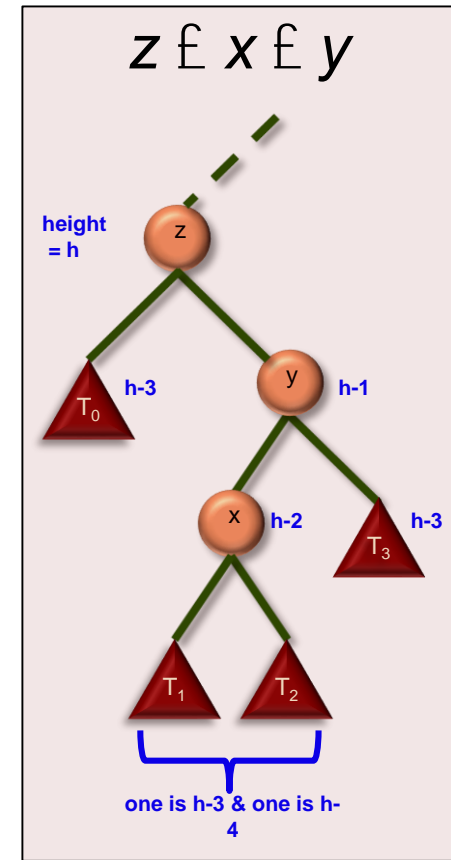
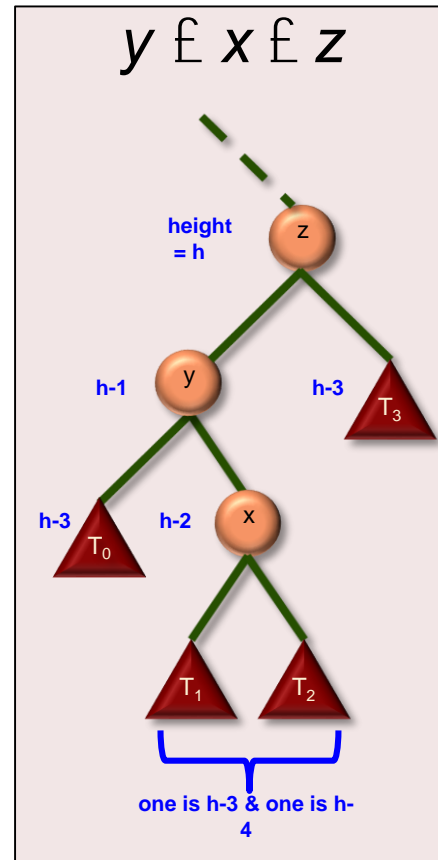
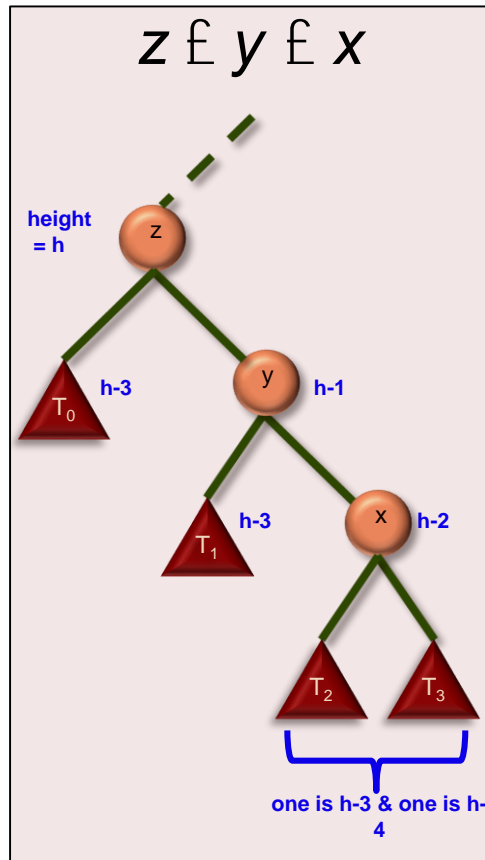
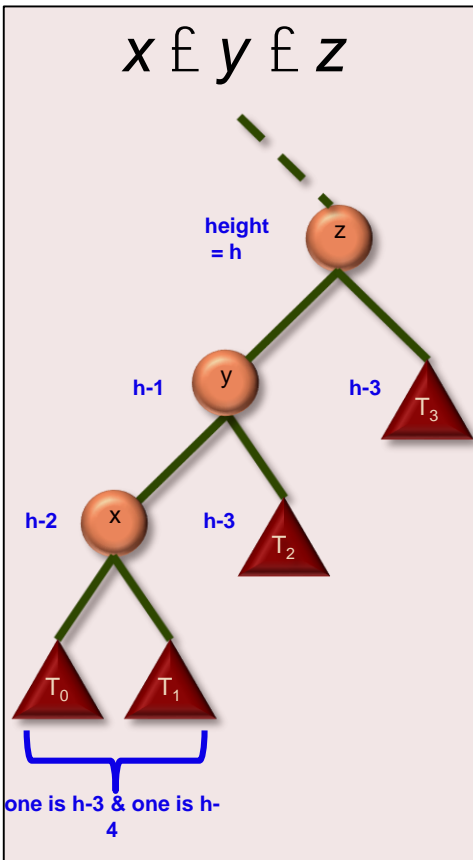


# Insertion: Trinode Restructuring Example



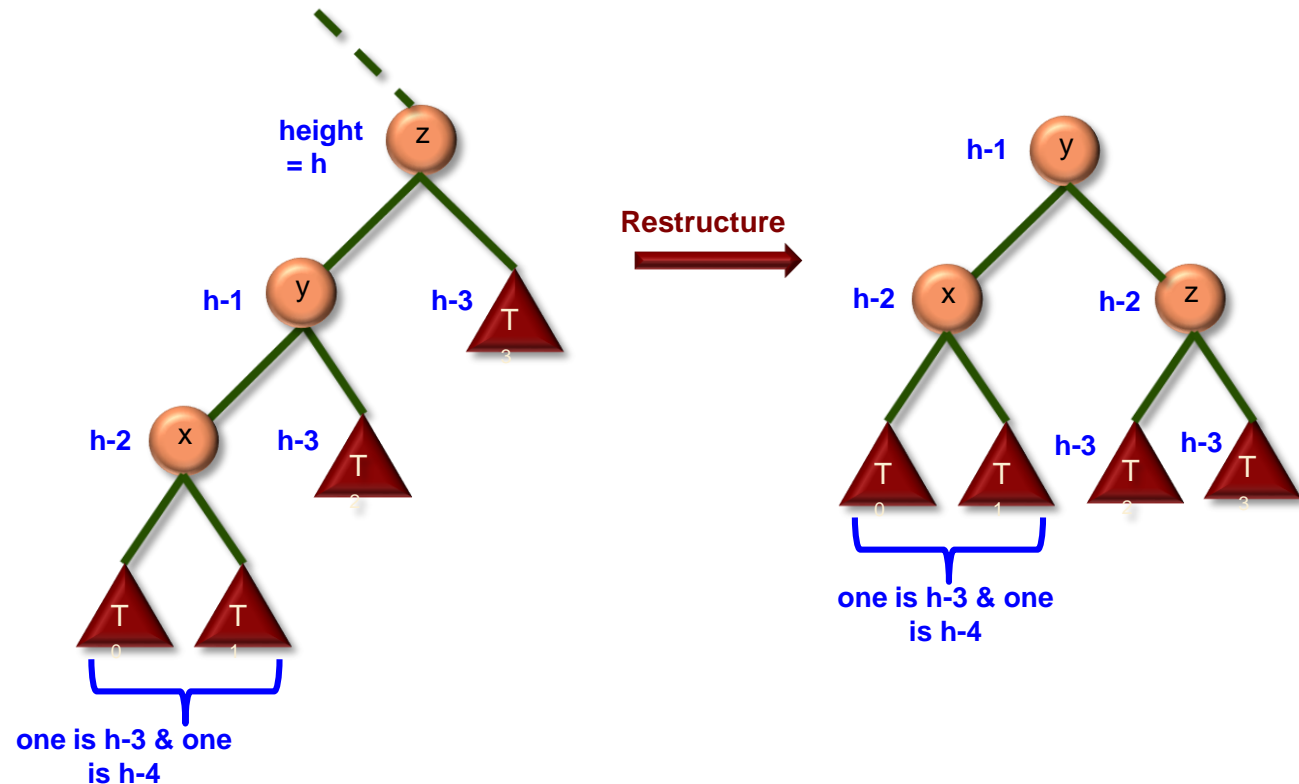
# Insertion: Trinode Restructuring - 4 Cases

- There are 4 different possible relationships between the three nodes x, y and z before restructuring:



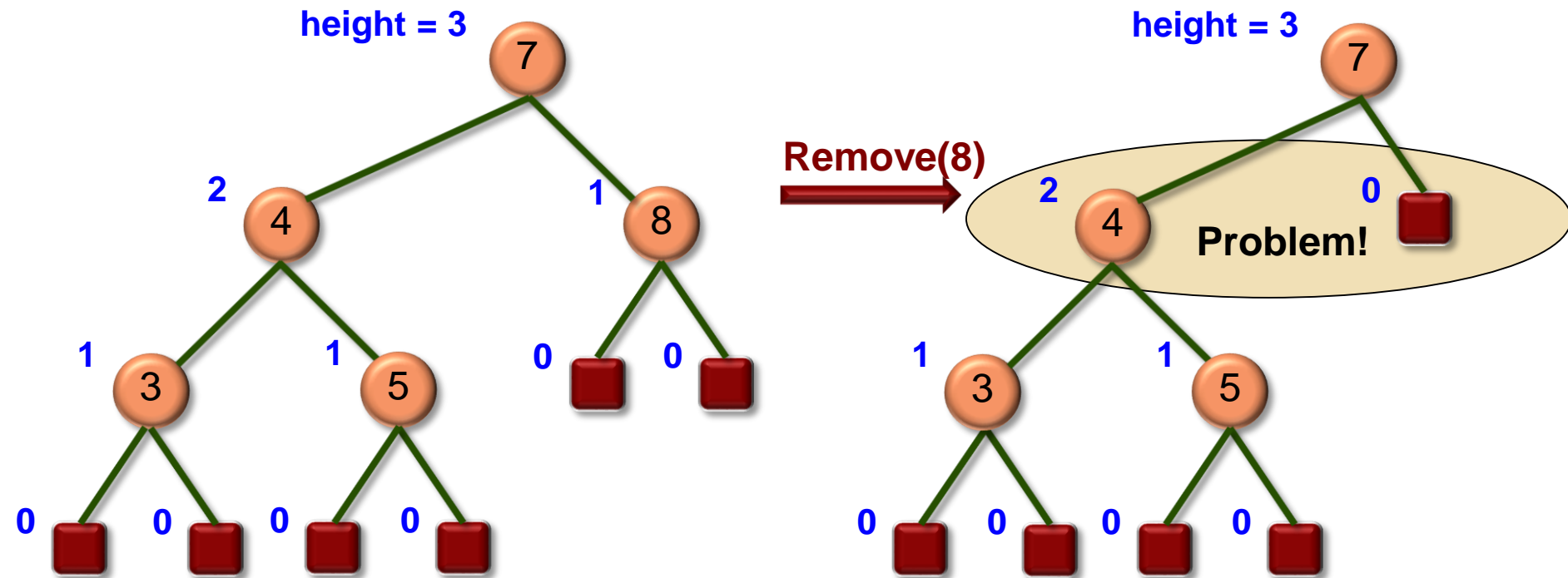
# Insertion: Trinode Restructuring - The Whole Tree

- Do we have to repeat this process further up the tree?
- No!
  - ❑ The tree was balanced before the insertion.
  - ❑ Insertion raised the height of the subtree by 1.
  - ❑ Rebalancing lowered the height of the subtree by 1.
  - ❑ Thus the whole tree is still balanced.



# Removal

- Imbalance may occur at an ancestor of the removed node.

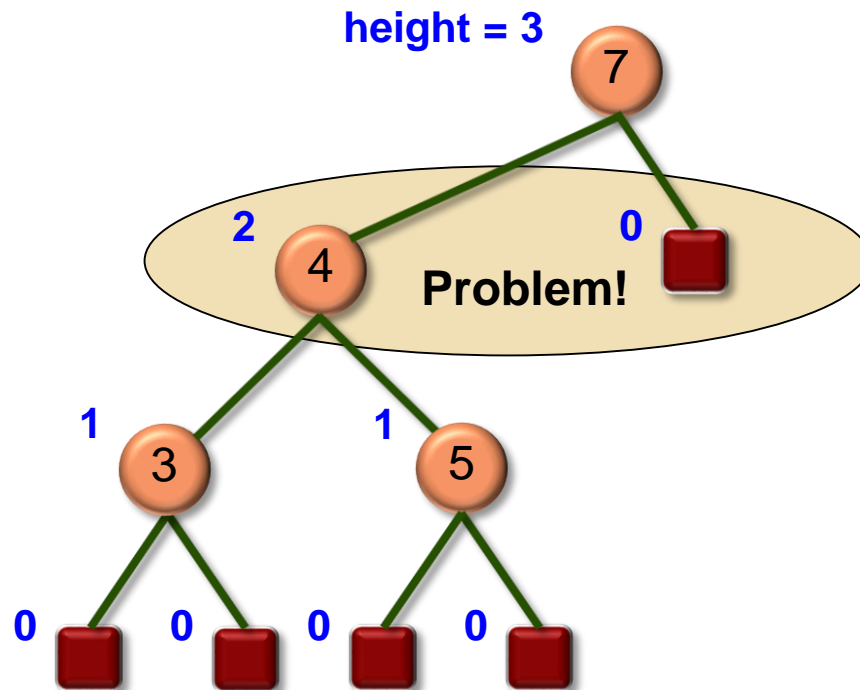




# Removal: Rebalancing Strategy

## ➤ Step 1: Search

- Starting at the location of the removed node, traverse toward the root until an imbalance is discovered.



# Removal: Rebalancing Strategy

## ➤ Step 2: Repair

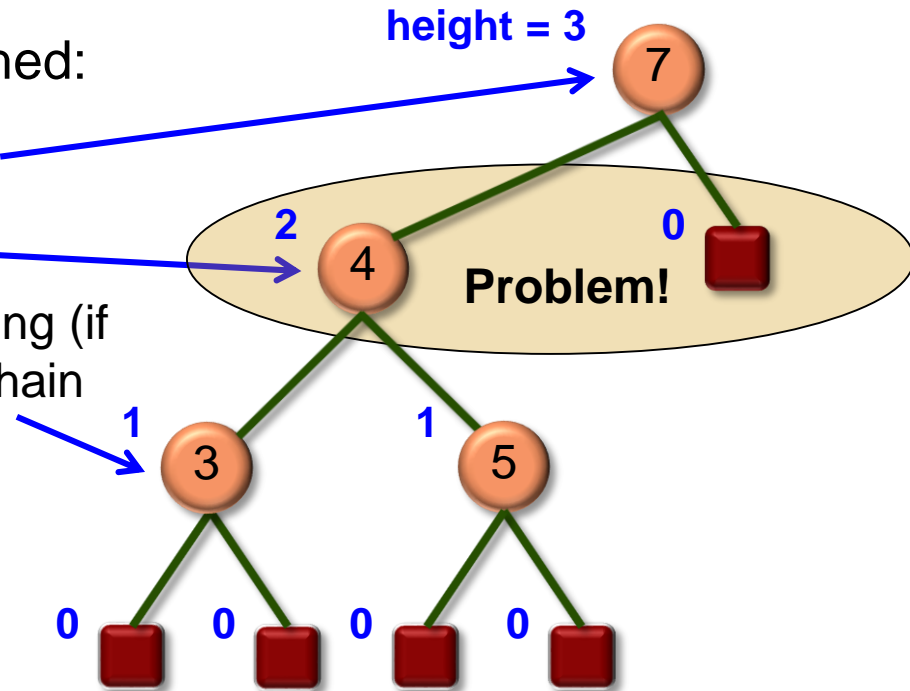
❑ We again use **trinode restructuring**.

❑ 3 nodes x, y and z are distinguished:

✧ z = the parent of the high sibling

✧ y = the high sibling

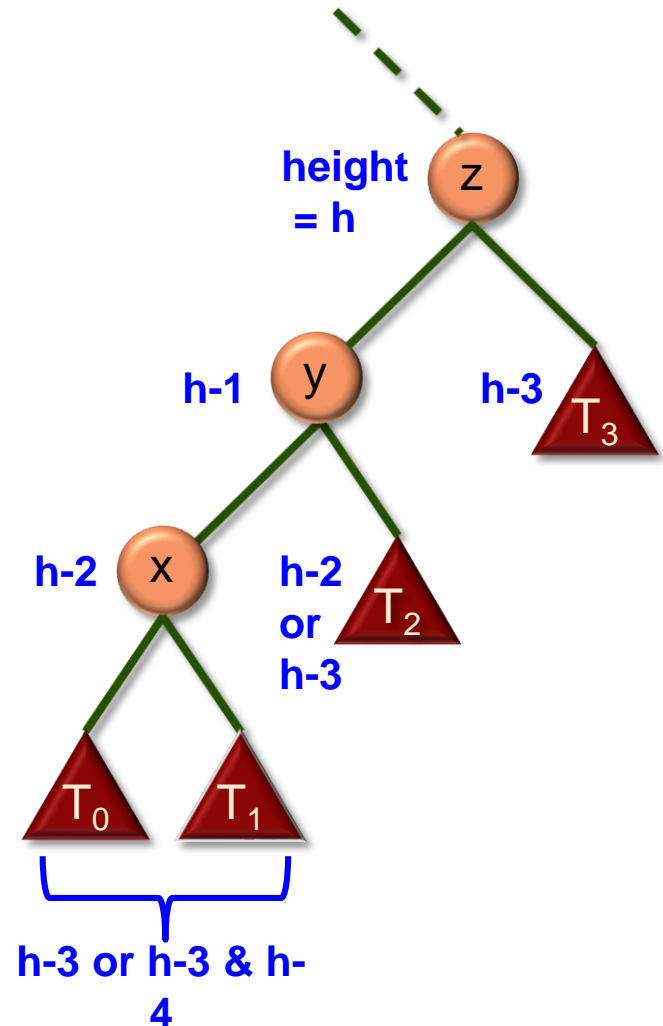
✧ x = the high child of the high sibling (if children are equally high, keep chain linear)



# Removal: Rebalancing Strategy

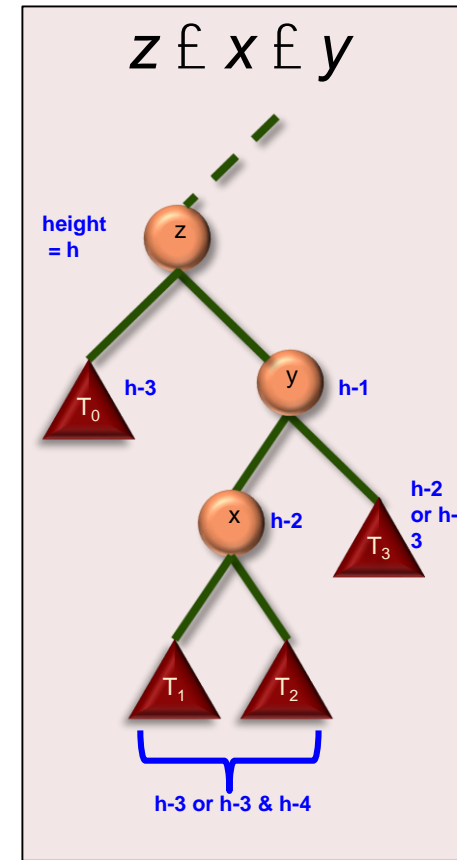
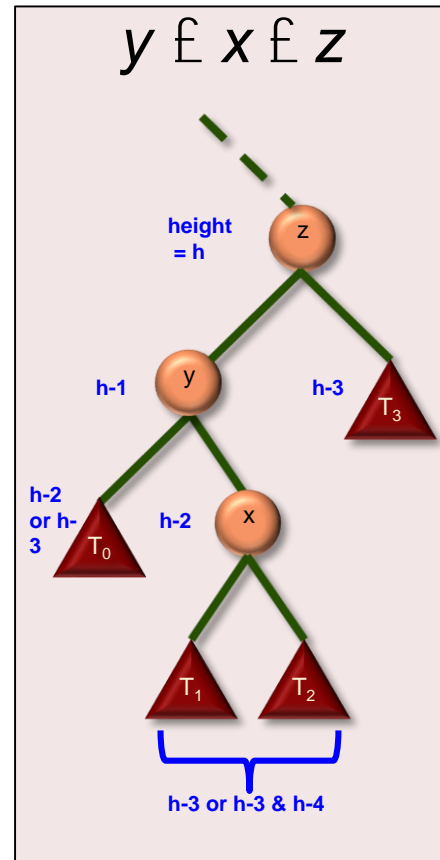
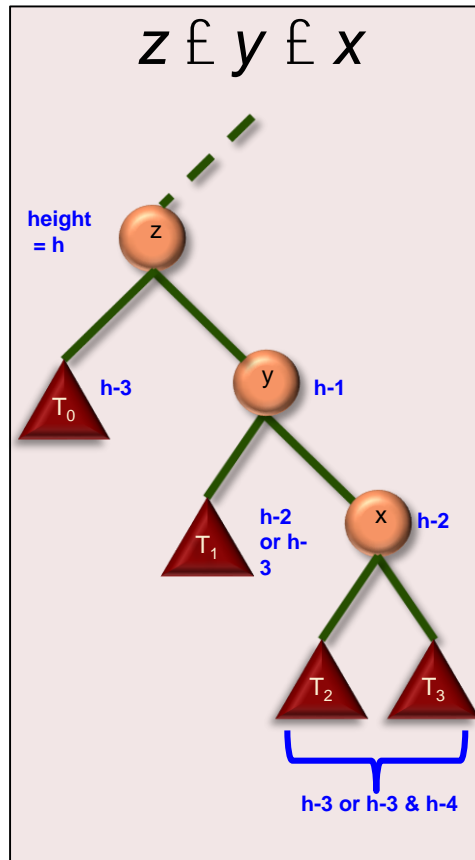
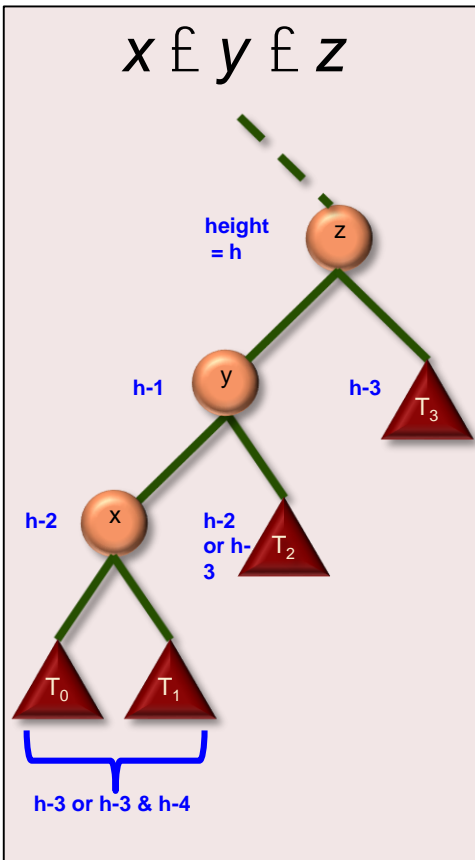
## ➤ Step 2: Repair

- ❑ The idea is to rearrange these 3 nodes so that the middle value becomes the root and the other two becomes its children.
- ❑ Thus the linear **grandparent – parent – child** structure becomes a triangular **parent – two children** structure.
- ❑ Note that **z** must be either bigger than both **x** and **y** or smaller than both **x** and **y**.
- ❑ Thus either **x** or **y** is made the root of this subtree, and **z** is lowered by 1.
- ❑ Then the subtrees **T<sub>0</sub> – T<sub>3</sub>** are attached at the appropriate places.
- ❑ Although the subtrees **T<sub>0</sub> – T<sub>3</sub>** can differ in height by up to 2, after restructuring, sibling subtrees will differ by at most 1.

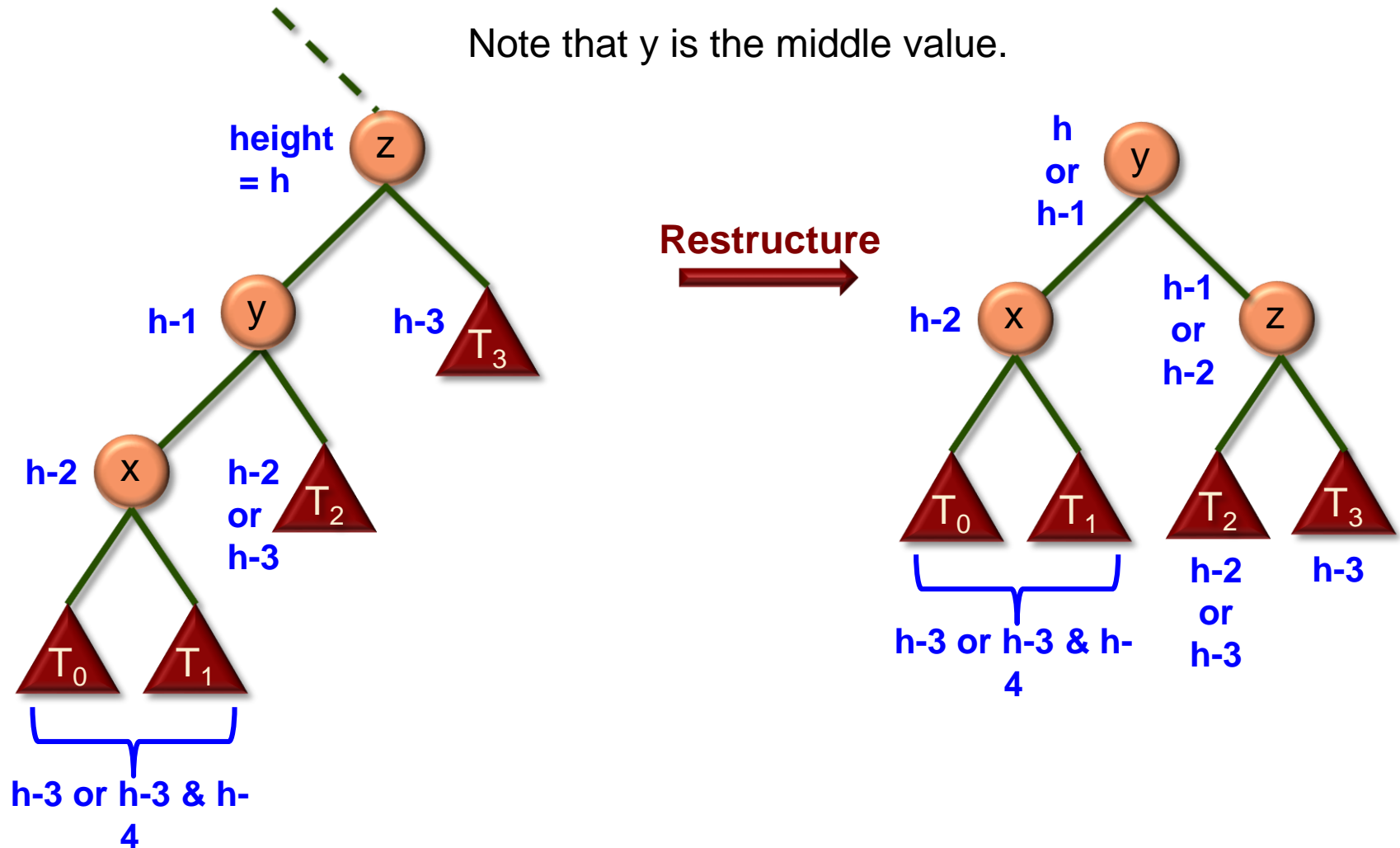


# Removal: Trinode Restructuring - 4 Cases

- There are 4 different possible relationships between the three nodes x, y and z before restructuring:



# Removal: Trinode Restructuring - Case 1



# Removal: Rebalancing Strategy

## ➤ Step 2: Repair

- ❑ Unfortunately, trinode restructuring may reduce the height of the subtree, causing another imbalance further up the tree.
- ❑ Thus this search and repair process must be repeated until we reach the root.

## Topic 2. Sorting

# Sorting Algorithms

## ➤ Comparison Sorting

- ☐ Selection Sort
- ☐ Bubble Sort
- ☐ Insertion Sort
- ☐ Merge Sort
- ☐ Heap Sort
- ☐ Quick Sort

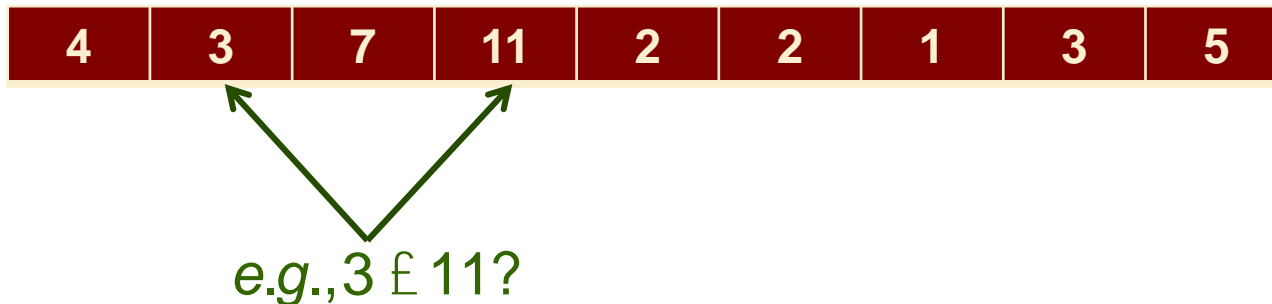
## ➤ Linear Sorting

- ☐ Counting Sort
- ☐ Radix Sort
- ☐ Bucket Sort



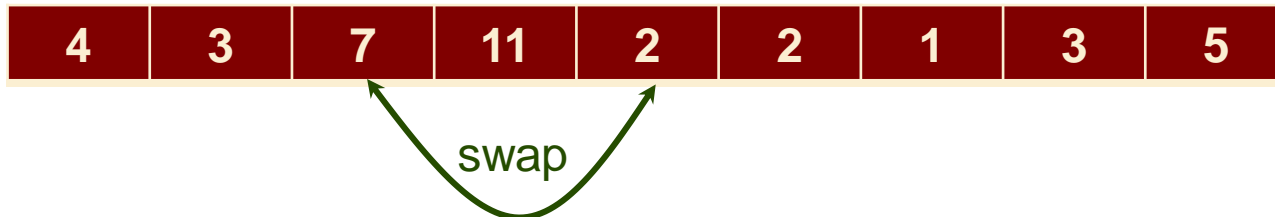
# Comparison Sorts

- Comparison Sort algorithms sort the input by successive comparison of pairs of input elements.
- Comparison Sort algorithms are very general: they make no assumptions about the values of the input elements.



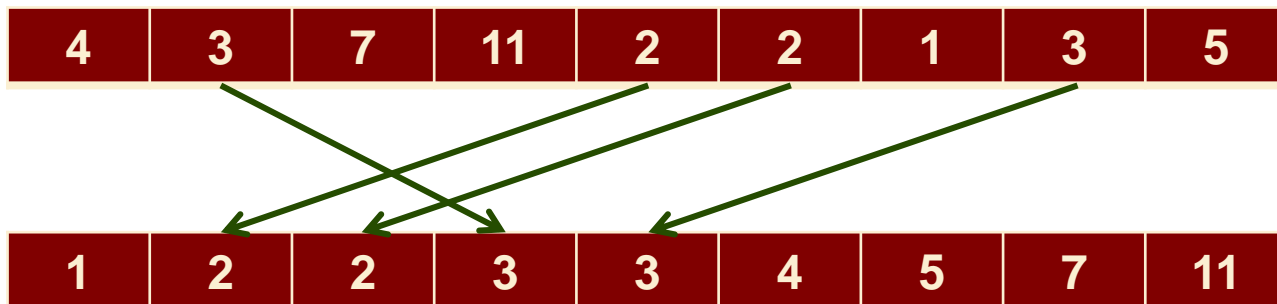
# Sorting Algorithms and Memory

- Some algorithms sort by swapping elements within the input array
- Such algorithms are said to **sort in place**, and require only  $O(1)$  additional memory.
- Other algorithms require allocation of an output array into which values are copied.
- These algorithms do not sort in place, and require  $O(n)$  additional memory.



# Stable Sort

- A sorting algorithm is said to be **stable** if the ordering of identical keys in the input is preserved in the output.
- The stable sort property is important, for example, when entries with identical keys are already ordered by another criterion.
- (Remember that stored with each key is a record containing some useful information.)



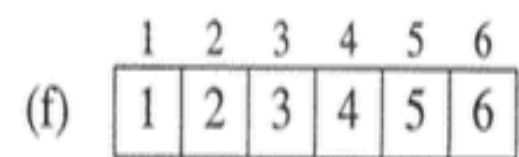
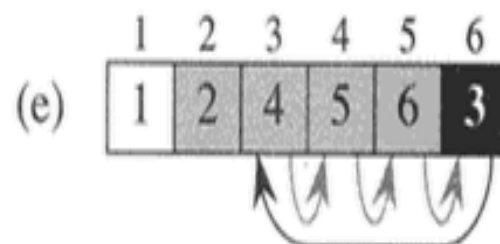
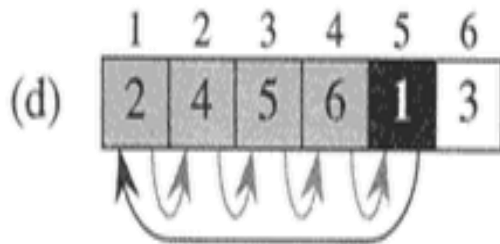
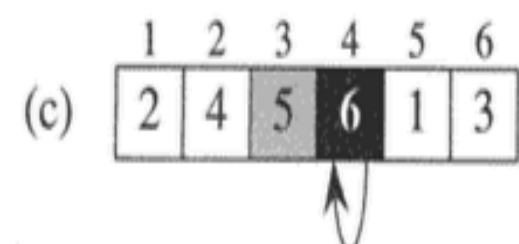
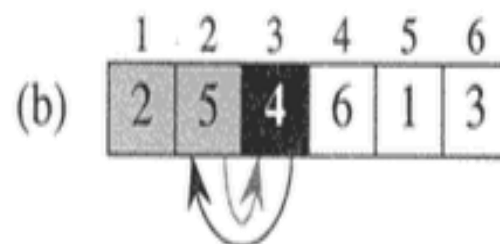
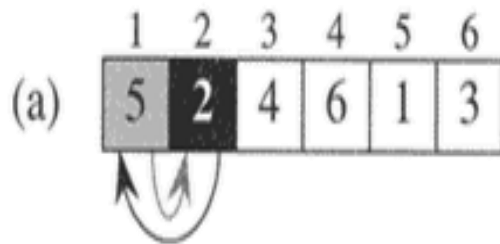
# Selection Sort

- Selection Sort operates by first finding the smallest element in the input list, and moving it to the output list.
- It then finds the next smallest value and does the same.
- It continues in this way until all the input elements have been selected and placed in the output list in the correct order.
- Note that every selection requires a search through the input list.
- Thus the algorithm has a nested loop structure
- Selection Sort Example

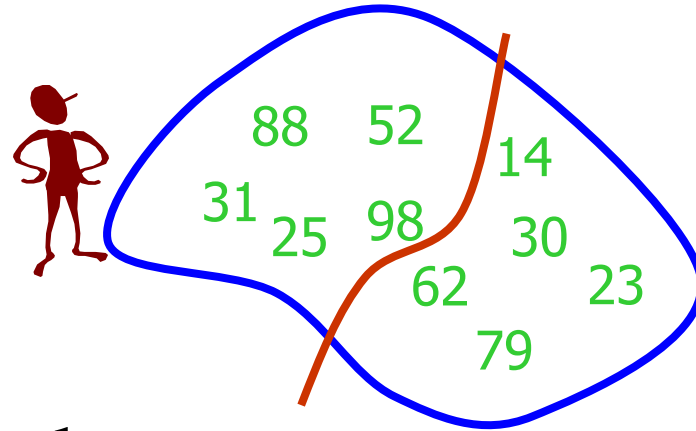
# Bubble Sort

- Bubble Sort operates by successively comparing adjacent elements, swapping them if they are out of order.
- At the end of the first pass, the largest element is in the correct position.
- A total of  $n$  passes are required to sort the entire array.
- Thus bubble sort also has a nested loop structure
- Bubble Sort Example

# Example: Insertion Sort



# Merge Sort



Split Set into Two  
(no real work)

Get one friend to  
sort the first half.



25,31,52,88,98

Get one friend to  
sort the second half.



14,23,30,62,79

# Merge Sort

Merge two sorted lists into one



25,31,52,88,98

14,23,30,62,79

14,23,25,30,31,52,62,79,88,98



# Analysis of Merge-Sort

- The height  $h$  of the merge-sort tree is  $O(\log n)$ 
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth  $i$  is  $O(n)$ 
  - we partition and merge  $2^i$  sequences of size  $n/2^i$
  - we make  $2^{i+1}$  recursive calls
- Thus, the total running time of merge-sort is  $O(n \log n)$

$$T(n) = 2T(n/2) + O(n)$$

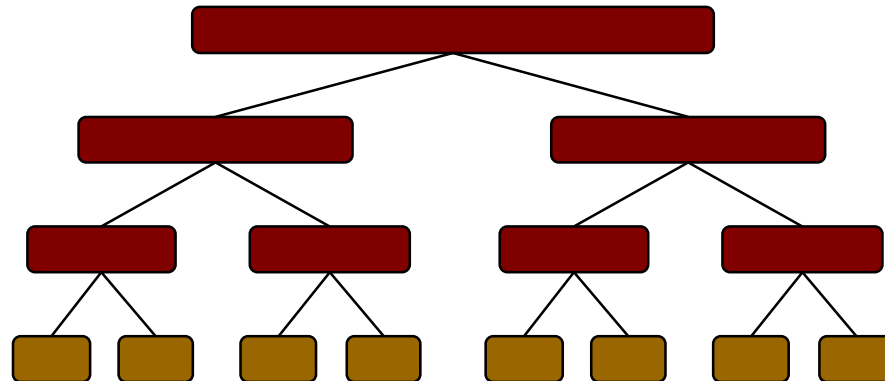
depth	#seqs	size
-------	-------	------

0	1	$n$
---	---	-----

1	2	$n/2$
---	---	-------

$i$	$2^i$	$n/2^i$
-----	-------	---------

...	...	...
-----	-----	-----



# Heap-Sort Algorithm

- Build an array-based (max) heap
- Iteratively call `removeMax()` to extract the keys in descending order
- Store the keys as they are extracted in the unused tail portion of the array

# Heap-Sort Running Time

- The heap can be built bottom-up in  $O(n)$  time
- Extraction of the  $i$ th element takes  $O(\log(n - i + 1))$  time (for downheaping)
- Thus total run time is

$$\begin{aligned} T(n) &= O(n) + \sum_{i=1}^n \log(n - i + 1) \\ &= O(n) + \sum_{i=1}^n \log i \\ &\in O(n) + \sum_{i=1}^n \log n \\ &= O(n \log n) \end{aligned}$$

# Quick-Sort

➤ **Quick-sort** is a divide-and-conquer algorithm:

❑ **Divide**: pick a random element  $x$  (called a **pivot**) and partition  $S$  into

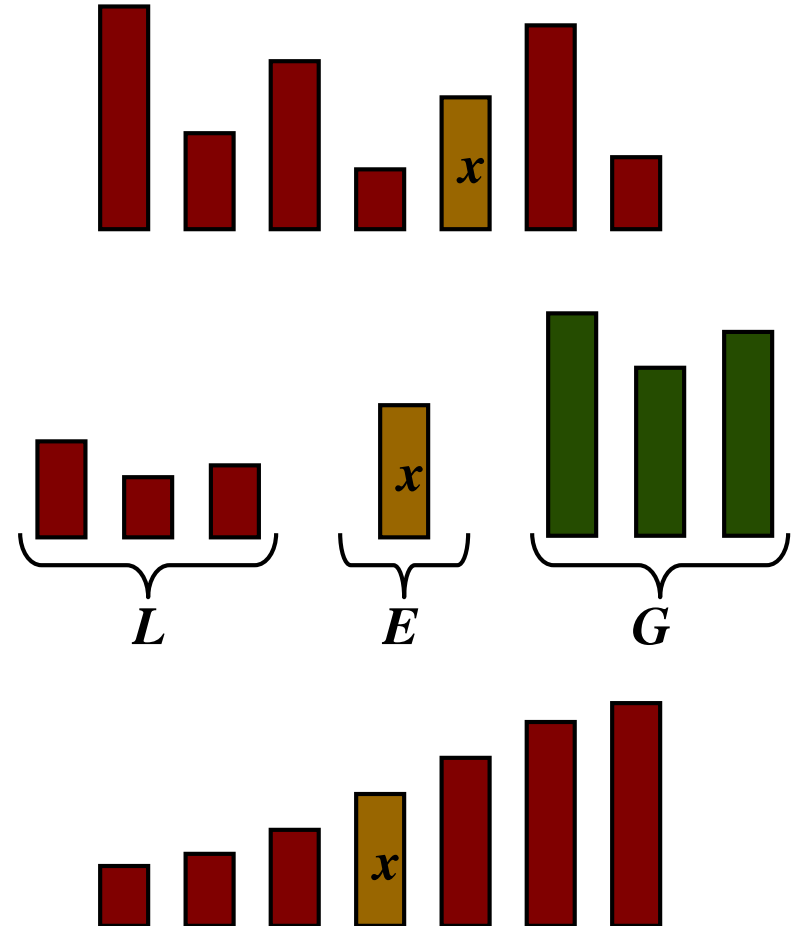
✧  $L$  elements less than  $x$

✧  $E$  elements equal to  $x$

✧  $G$  elements greater than  $x$

❑ **Recur**: Quick-sort  $L$  and  $G$

❑ **Conquer**: join  $L$ ,  $E$  and  $G$



# The Quick-Sort Algorithm

**Algorithm QuickSort(S)**

if S.size() > 1

(L, E, G) = Partition(S)

QuickSort(L)

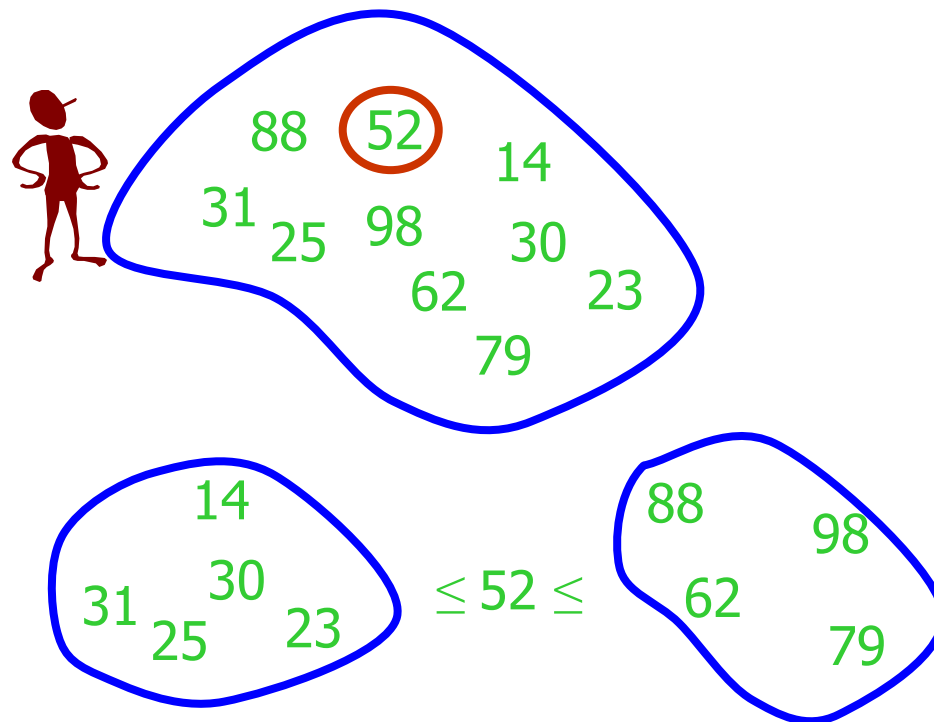
QuickSort(G)

S = (L, E, G)

# In-Place Quick-Sort

- **Note:** Use the lecture slides here instead of the textbook implementation (Section 11.2.2)

Partition set into **two** using randomly chosen pivot



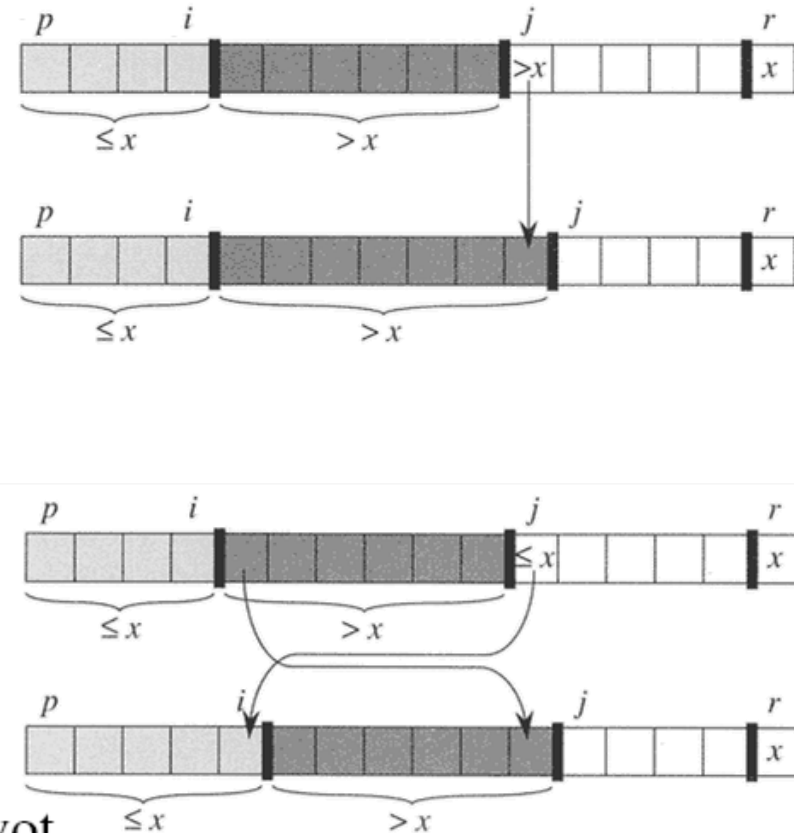
# Maintaining Loop Invariant

PARTITION( $A, p, r$ )

```
1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$ 
4      do if  $A[j] \leq x$ 
5          then  $i \leftarrow i + 1$ 
6              exchange  $A[i] \leftrightarrow A[j]$ 
7  exchange  $A[i + 1] \leftrightarrow A[r]$ 
8  return  $i + 1$ 
```

**Loop invariant:**

1. All entries in  $A[p \dots i]$  are  $\leq$  pivot.
2. All entries in  $A[i + 1 \dots j - 1]$  are  $>$  pivot.
3.  $A[r] =$  pivot.



# The In-Place Quick-Sort Algorithm

**Algorithm QuickSort**(A, p, r)

if  $p < r$

$q = \text{Partition}(A, p, r)$

    QuickSort(A, p,  $q - 1$ )

    QuickSort(A,  $q + 1$ , r)

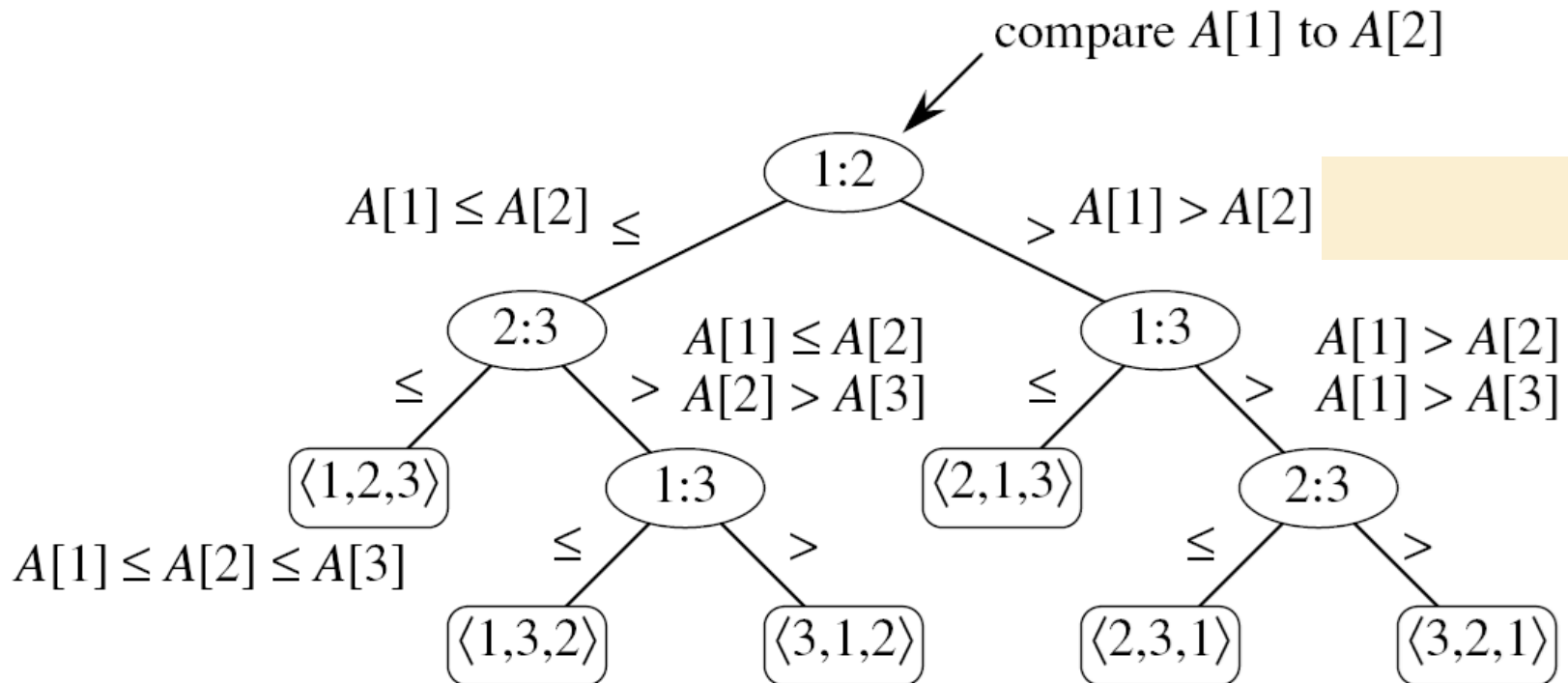


# Summary of Comparison Sorts

Algorithm	Best Case	Worst Case	Average Case	In Place	Stable	Comments
Selection	$n^2$	$n^2$		Yes	Yes	
Bubble	$n$	$n^2$		Yes	Yes	
Insertion	$n$	$n^2$		Yes	Yes	Good if often almost sorted
Merge	$n \log n$	$n \log n$		No	Yes	Good for very large datasets that require swapping to disk
Heap	$n \log n$	$n \log n$		Yes	No	Best if guaranteed $n \log n$ required
Quick	$n \log n$	$n^2$	$n \log n$	Yes	No	Usually fastest in practice

# Comparison Sort: Decision Trees

- For a 3-element array, there are 6 external nodes.
- For an  $n$ -element array, there are  $n!$  external nodes.



# Comparison Sort

- To store  $n!$  external nodes, a decision tree must have a height of at least  $\lceil \log n! \rceil$
- Worst-case time is equal to the height of the binary decision tree.

Thus  $T(n) \in W(\log n!)$

$$\text{where } \log n! = \sum_{i=1}^n \log i \geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log \lfloor n/2 \rfloor \in W(n \log n)$$

Thus  $T(n) \in W(n \log n)$

**Thus MergeSort & HeapSort are asymptotically optimal.**

# Linear Sorts?

Comparison sorts are very general, but are  $\Omega(n \log n)$

Faster sorting may be possible if we can constrain the nature of the input.

# CountingSort

Input:

<del>1</del>	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
					1													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Output:

					1													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

Value v:

0	1	2	3
0	5	14	17

Location of **next** record  
with digit v.

Algorithm: Go through the records in order  
putting them where they go.

# CountingSort

Input:

1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
0					1													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Output:

0					1													
---	--	--	--	--	---	--	--	--	--	--	--	--	--	--	--	--	--	--

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

Value v:

0	1	2	3
0	6	14	17

Location of **next** record  
with digit v.

Algorithm: Go through the records in order  
putting them where they go.

# RadixSort

344		333		2 24
125		143		1 25
333	Sort wrt which	243	Sort wrt which	2 25
134	digit first?	344	digit Second?	3 25
224		134		3 33
334	The least	224	The next least	1 34
143	significant.	334	significant.	3 34
225		125		1 43
325		225		2 43
243		325		3 44



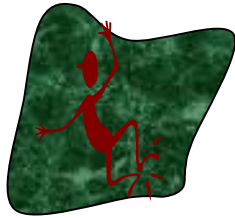
Is sorted wrt least sig. 2 digits.



# RadixSort

2	24
1	25
2	25
3	25
3	33
1	34
3	34
1	43
2	43
3	44

$i+1$



Is sorted wrt  
first  $i$  digits.



Sort wrt  $i+1$ st  
digit.

1	25
1	34
1	43
<hr/>	
2	24
2	25
2	43
<hr/>	
3	25
3	33
3	34
3	44



Is sorted wrt  
first  $i+1$  digits.

These are in the  
correct order  
because sorted  
wrt high order digit



# RadixSort

2 24

1 25

2 25

3 25

3 33

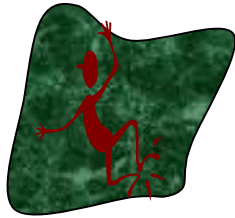
1 34

3 34

1 43

2 43

3 44  
i+1



Is sorted wrt  
first  $i$  digits.



Sort wrt  $i+1$ st  
digit.

1 25

1 34

1 43

2 24

2 25

2 43

3 25

3 33

3 34

3 44



Is sorted wrt  
first  $i+1$  digits.

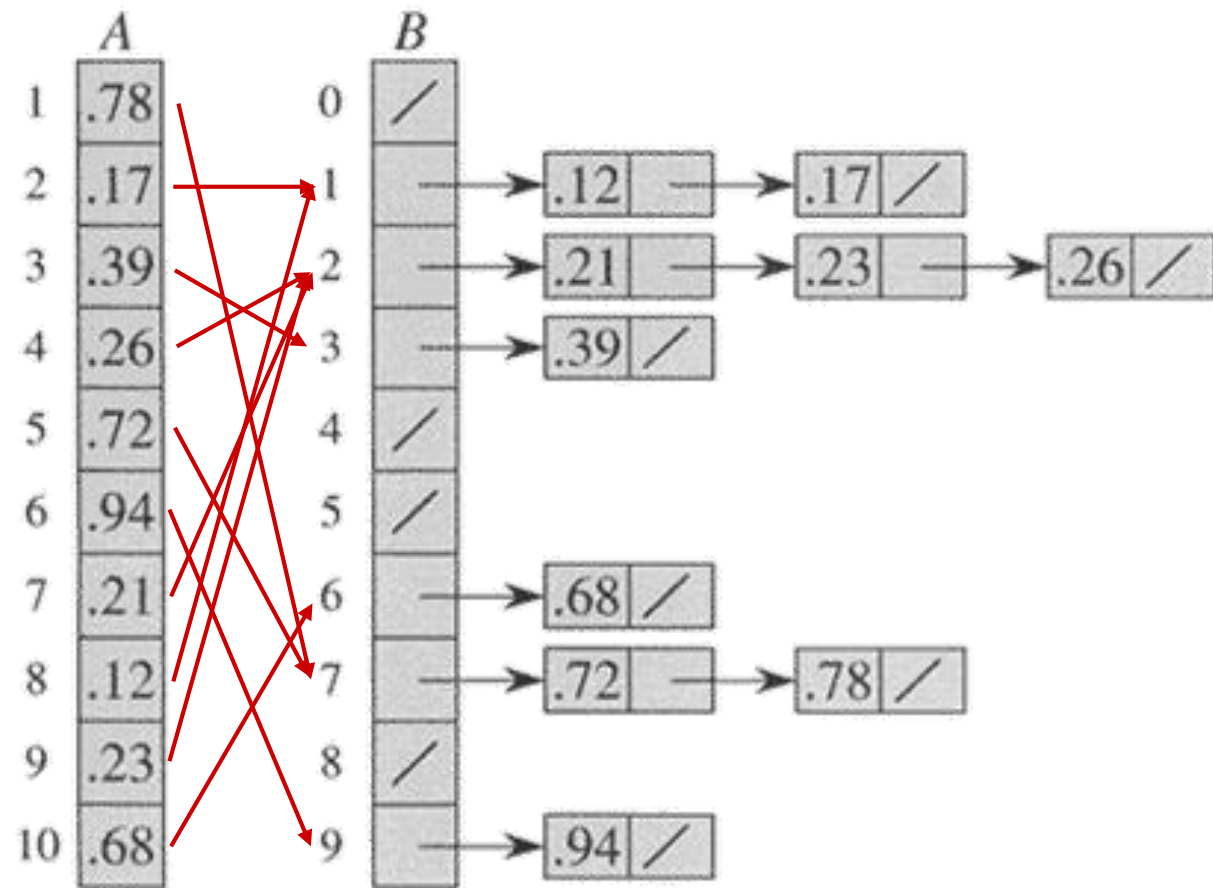
These are in the  
correct order  
because was sorted &  
stable sort left sorted

## Example 3. Bucket Sort

- Applicable if input is constrained to finite interval, e.g.,  $[0 \dots 1)$ .
- If input is random and uniformly distributed, **expected** run time is  $\Theta(n)$ .

# Bucket Sort

insert  $A[i]$  into list  $B[\lfloor n \cdot A[i] \rfloor]$



# Topic 3. Graphs

# Graphs

- Definitions & Properties
- Implementations
- Depth-First Search
- Topological Sort
- Breadth-First Search

# Properties

## Property 1

$$\sum_v \deg(v) = 2|E|$$

Proof: each edge is counted twice

## Notation

$|V|$  number of vertices

$|E|$  number of edges

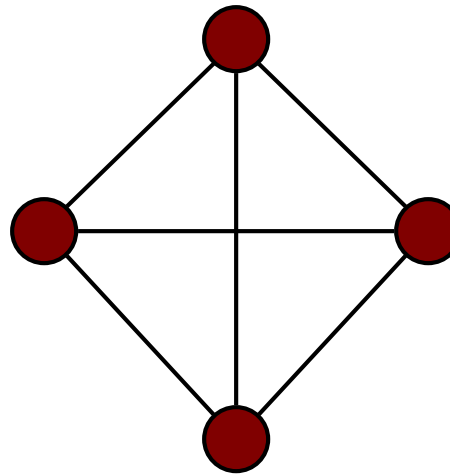
$\deg(v)$  degree of vertex  $v$

## Property 2

In an undirected graph with no self-loops and no multiple edges

$$|E| \leq |V|(|V| - 1)/2$$

Proof: each vertex has degree at most  $(|V| - 1)$



## Example

- $|V| = 4$
- $|E| = 6$
- $\deg(v) = 3$

Q: What is the bound for a digraph?

A:  $|E| \leq |V|(|V| - 1)$

# Main Methods of the (Undirected) Graph ADT

## ➤ Vertices and edges

- ❑ are positions
- ❑ store elements

## ➤ Accessor methods

- ❑ **endVertices**(e): an array of the two endvertices of e
- ❑ **opposite**(v, e): the vertex opposite to v on e
- ❑ **areAdjacent**(v, w): true iff v and w are adjacent
- ❑ **replace**(v, x): replace element at vertex v with x
- ❑ **replace**(e, x): replace element at edge e with x

## ➤ Update methods

- ❑ **insertVertex**(o): insert a vertex storing element o
- ❑ **insertEdge**(v, w, o): insert an edge (v,w) storing element o
- ❑ **removeVertex**(v): remove vertex v (and its incident edges)
- ❑ **removeEdge**(e): remove edge e

## ➤ Iterator methods

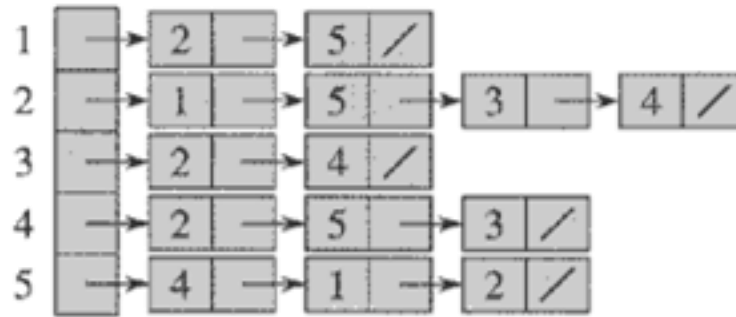
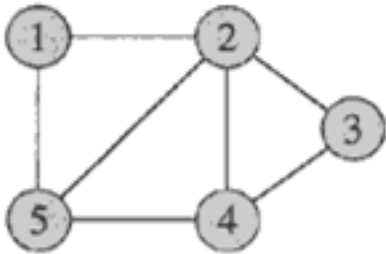
- ❑ **incidentEdges**(v): edges incident to v
- ❑ **vertices**(): all vertices in the graph
- ❑ **edges**(): all edges in the graph

# Running Time of Graph Algorithms

- Running time often a function of both  $|V|$  and  $|E|$ .
- For convenience, we sometimes drop the  $| \cdot |$  in asymptotic notation, e.g.  $O(V+E)$ .



# Implementing a Graph (Simplified)



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency List

Adjacency Matrix

Space complexity:

$$\theta(V + E)$$

$$\theta(V^2)$$

Time to find all neighbours of vertex  $u$ :  $\theta(\text{degree}(u))$

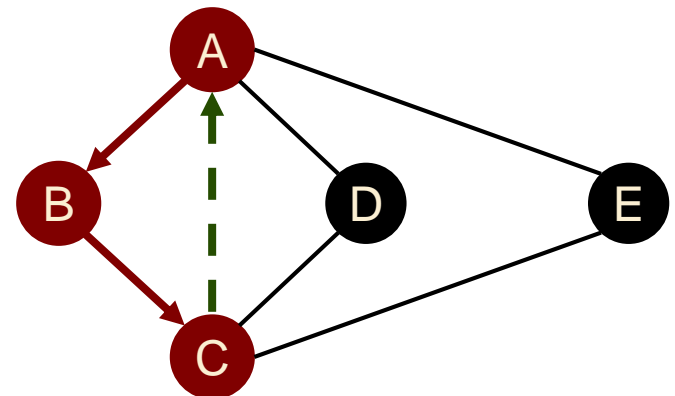
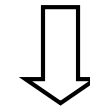
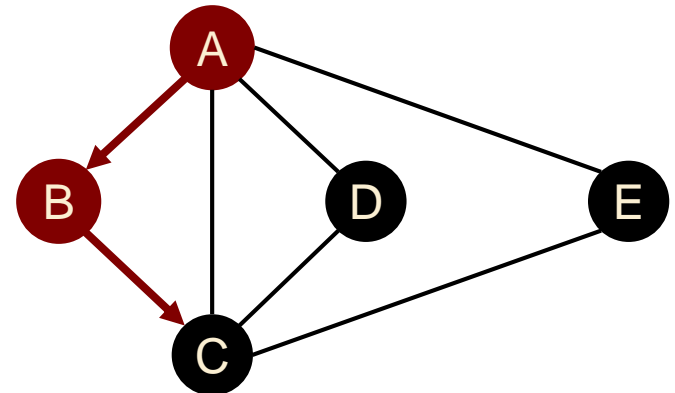
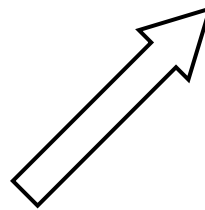
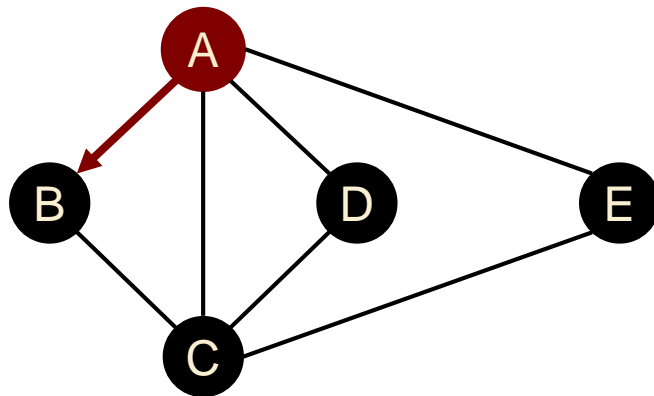
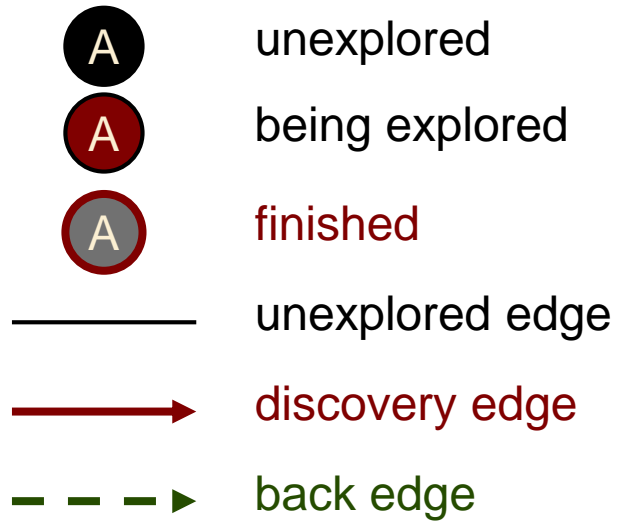
$$\theta(V)$$

Time to determine if  $(u, v) \in E$ :

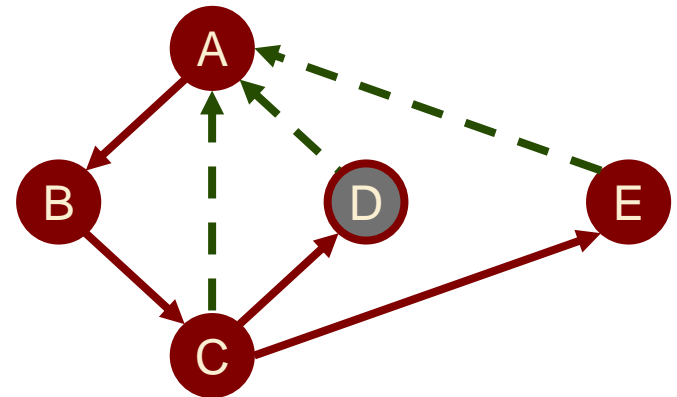
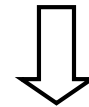
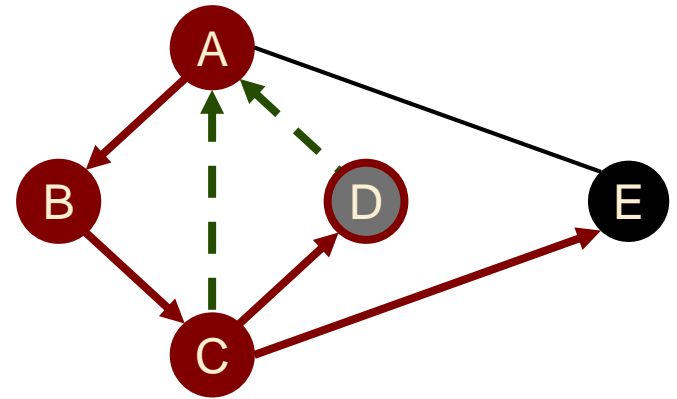
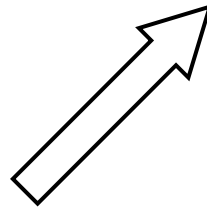
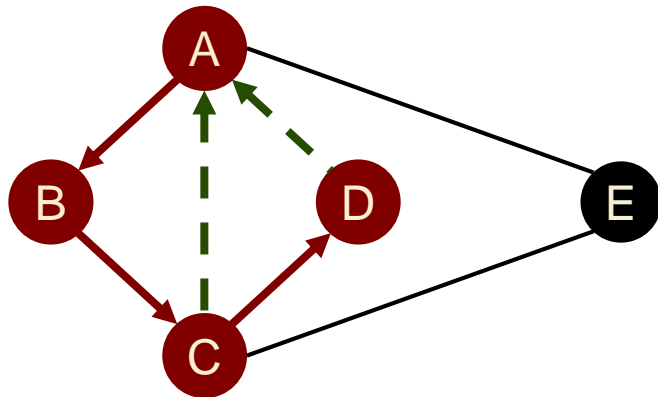
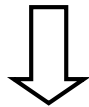
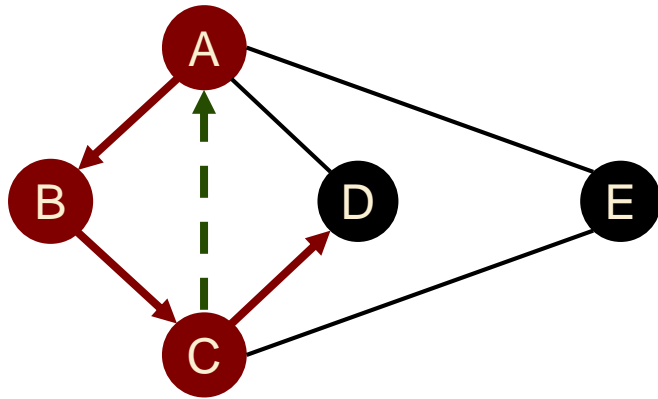
$$\theta(\text{degree}(u))$$

$$\theta(1)$$

# DFS Example on Undirected Graph



## Example (cont.)



# DFS Algorithm Pattern

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

for each vertex  $u \in V[G]$

    color[u] = BLACK //initialize vertex

for each vertex  $u \in V[G]$

    if color[u] = BLACK //as yet unexplored

        DFS-Visit( $u$ )

} total work  
=  $O(V)$



# DFS Algorithm Pattern

DFS-Visit ( $u$ )

Precondition: vertex  $u$  is undiscovered

Postcondition: all vertices reachable from  $u$  have been processed

$\text{colour}[u] \leftarrow \text{RED}$

for each  $v \in \text{Adj}[u]$  //explore edge  $(u,v)$

if  $\text{color}[v] = \text{BLACK}$

DFS-Visit( $v$ )

$\text{colour}[u] \leftarrow \text{GRAY}$

} total work  
=  $\sum_{v \in V} |\text{Adj}[v]| = q(E)$



Thus running time =  $q(V + E)$

(assuming adjacency list structure)

# Other Variants of Depth-First Search

- The DFS Pattern can also be used to
  - ❑ Compute a forest of spanning trees (one for each call to DFS-visit) encoded in a predecessor list  $\pi[u]$
  - ❑ Label edges in the graph according to their role in the search (see textbook)
    - ✧ **Tree edges**, traversed to an undiscovered vertex
    - ✧ **Forward edges**, traversed to a descendent vertex on the current spanning tree
    - ✧ **Back edges**, traversed to an ancestor vertex on the current spanning tree
    - ✧ **Cross edges**, traversed to a vertex that has already been discovered, but is not an ancestor or a descendent

# DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

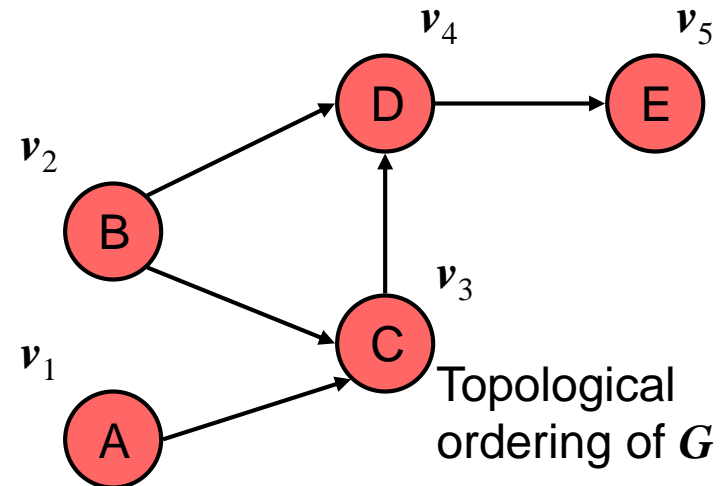
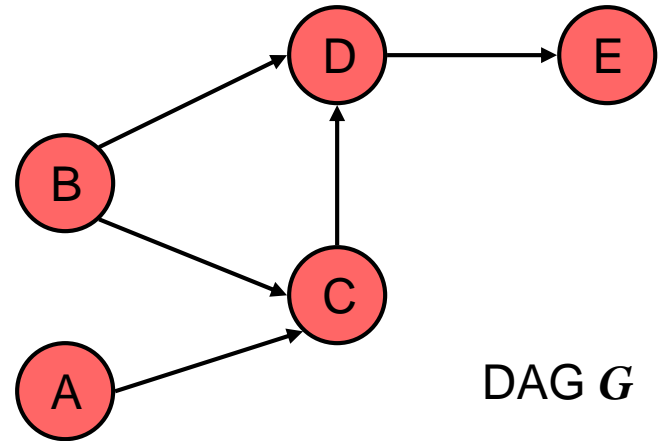
$$v_1, \dots, v_n$$

of the vertices such that for every edge  $(v_i, v_j)$ , we have  $i < j$

- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints

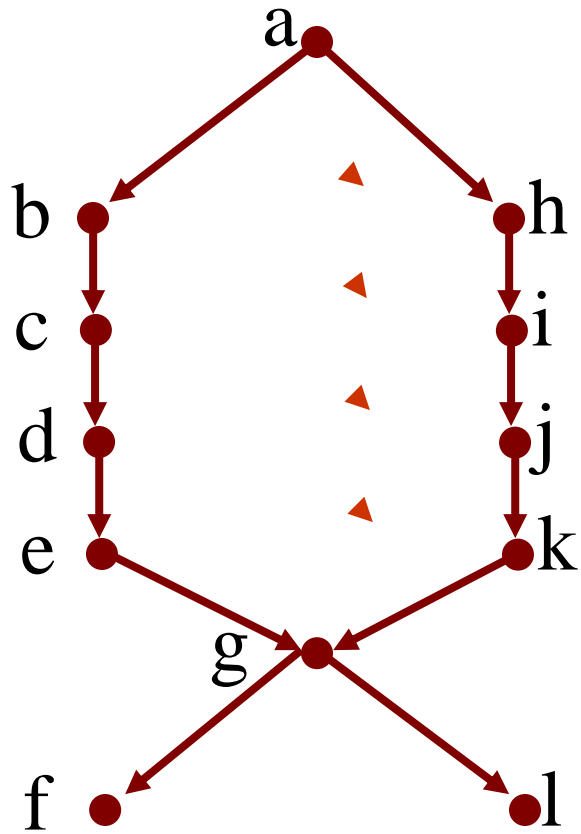
## Theorem

A digraph admits a topological ordering if and only if it is a DAG



# Linear Order

Alg: DFS



Found  
Not Handled  
Stack

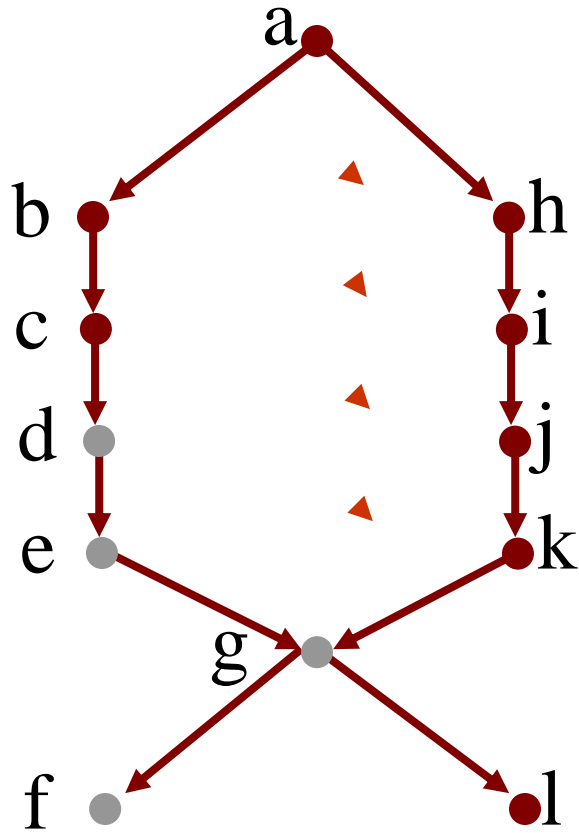


..... f



# Linear Order

Alg: DFS



Found  
Not Handled  
Stack



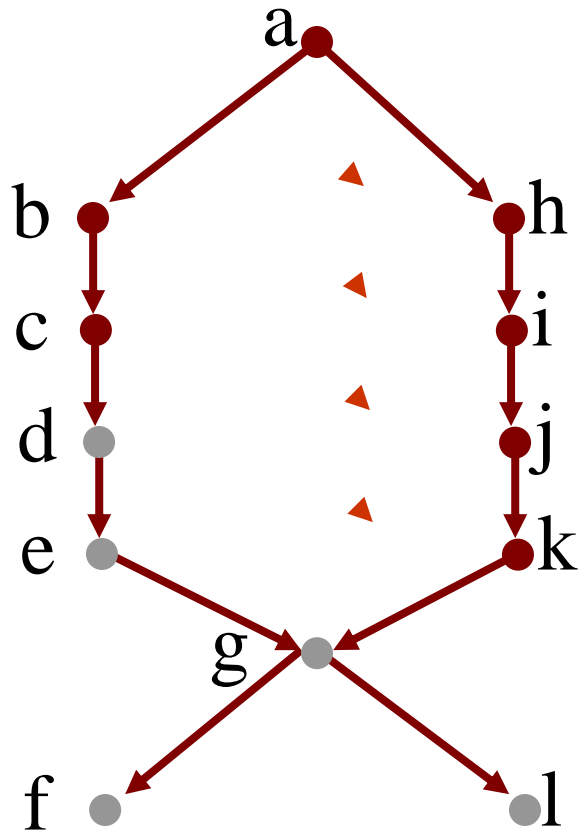
When node is popped off stack, insert at front of linearly-ordered “to do” list.

Linear Order:

..... f

# Linear Order

Alg: DFS



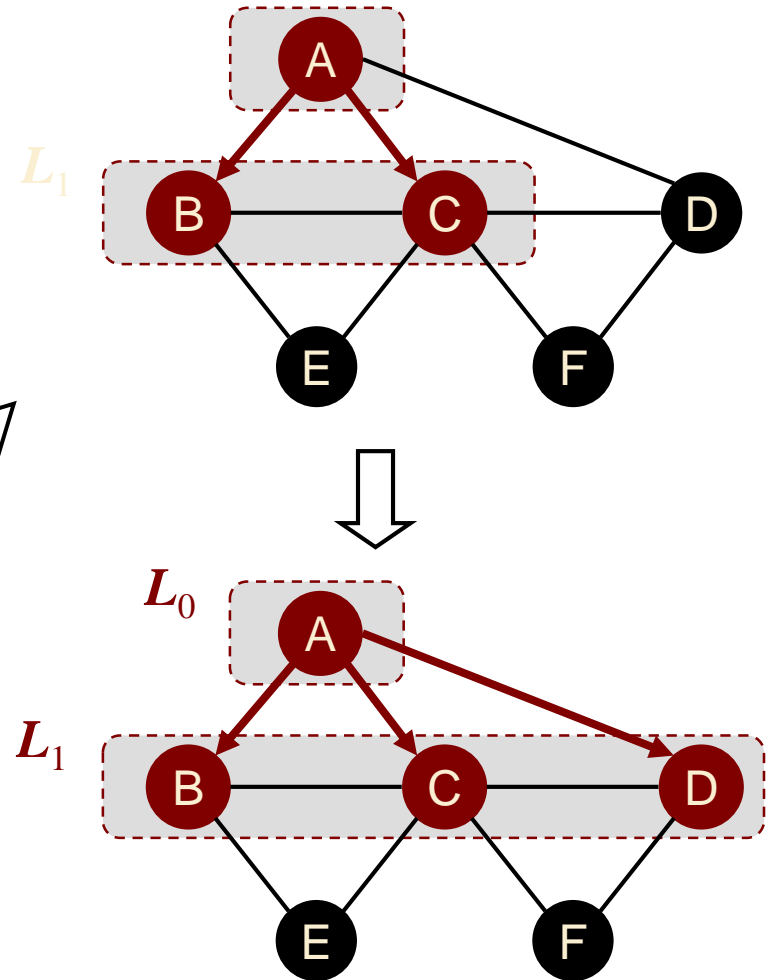
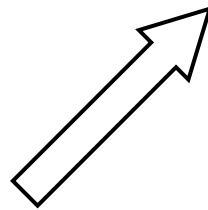
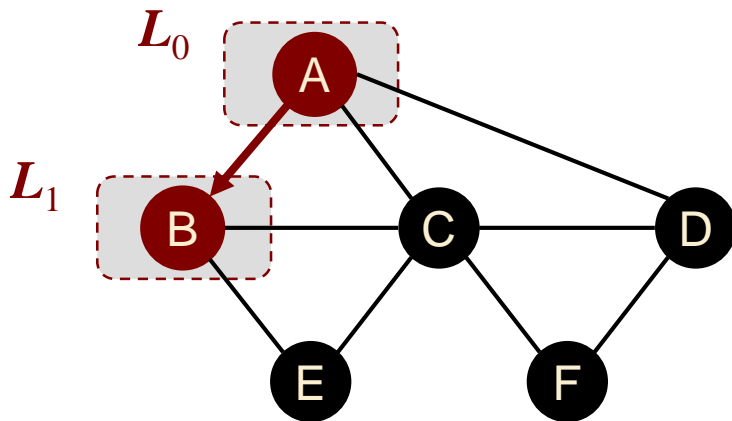
Found  
Not Handled  
Stack



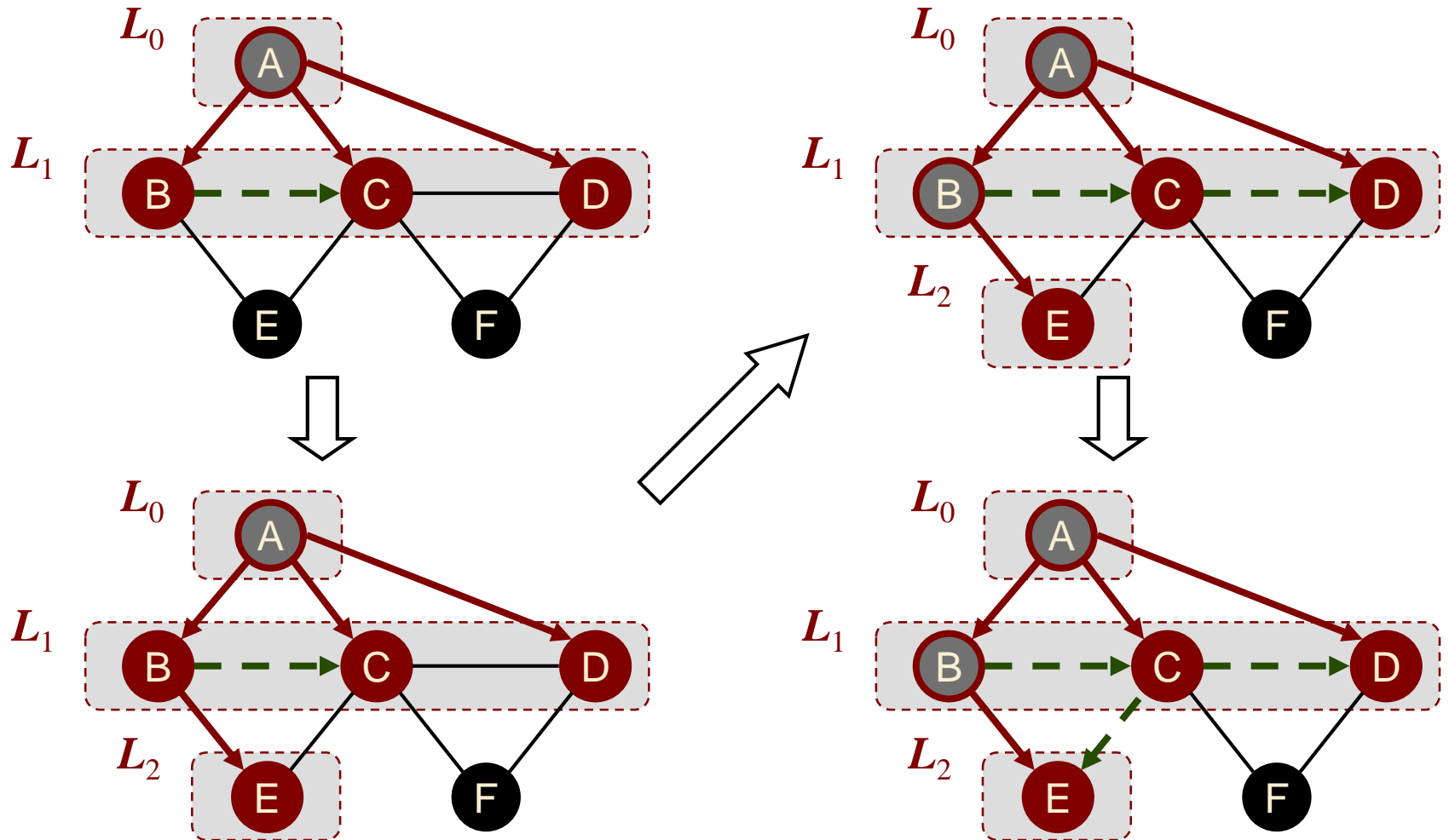
Linear Order:

l,f

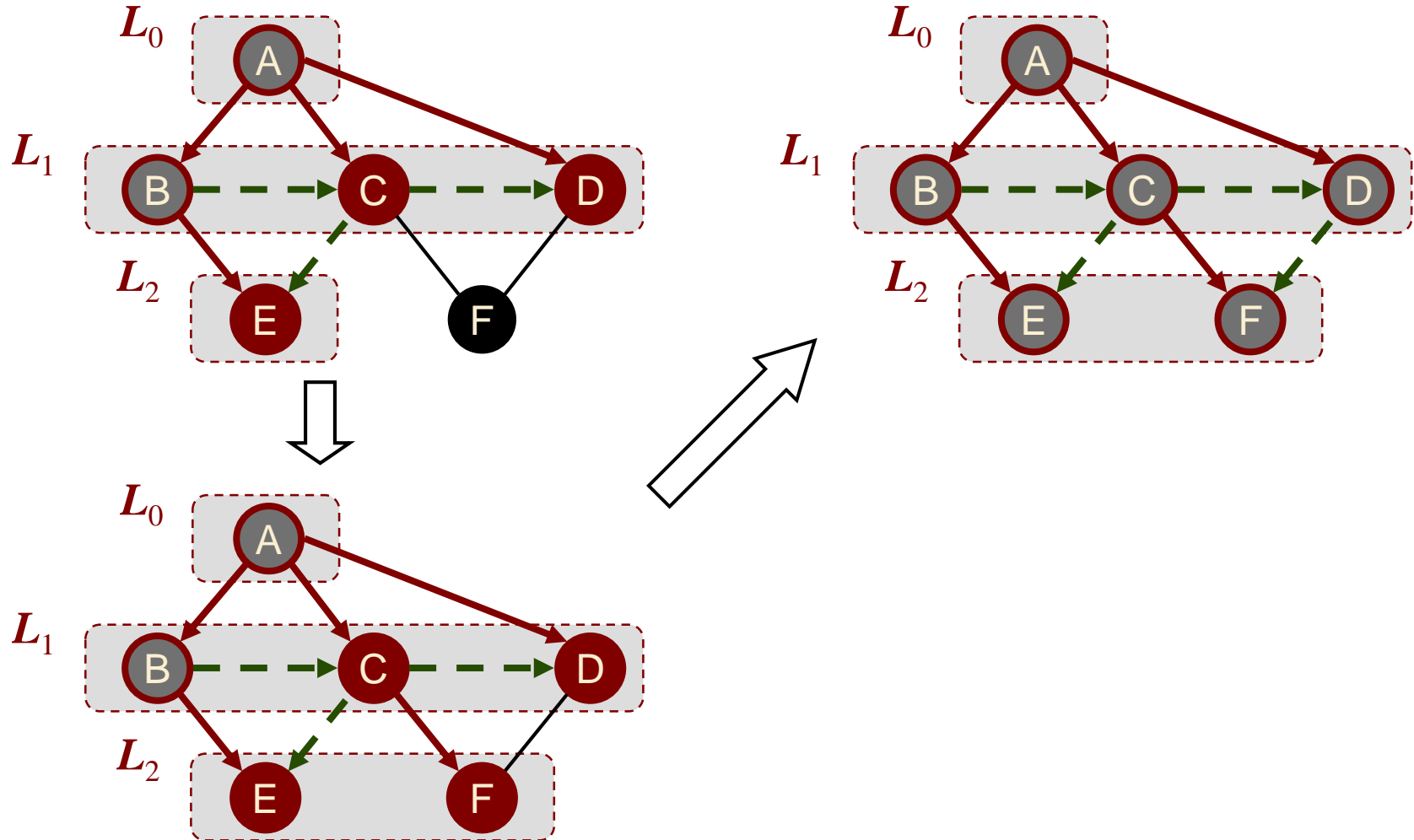
# BFS Example



# BFS Example (cont.)



# BFS Example (cont.)



# Analysis

- Setting/getting a vertex/edge label takes  $O(1)$  time
- Each vertex is labeled three times
  - ❑ once as BLACK (undiscovered)
  - ❑ once as RED (discovered, on queue)
  - ❑ once as GRAY (finished)
- Each edge is considered twice (for an undirected graph)
- Thus BFS runs in  $O(|V| + |E|)$  time provided the graph is represented by an adjacency list structure

# BFS Algorithm with Distances and Predecessors

BFS( $G, s$ )

Precondition:  $G$  is a graph,  $s$  is a vertex in  $G$

Postcondition:  $d[u]$  = shortest distance  $d[u]$  and

$p[u]$  = predecessor of  $u$  on shortest paths from  $s$  to each vertex  $u$  in  $G$

for each vertex  $u \in V[G]$

$d[u] \leftarrow \infty$

$p[u] \leftarrow \text{null}$

$\text{color}[u] = \text{BLACK}$  //initialize vertex

$\text{colour}[s] \leftarrow \text{RED}$

$d[s] \leftarrow 0$

$Q.\text{enqueue}(s)$

while  $Q \neq \emptyset$

$u \leftarrow Q.\text{dequeue}()$

for each  $v \in \text{Adj}[u]$  //explore edge  $(u, v)$

if  $\text{color}[v] = \text{BLACK}$

$\text{colour}[v] \leftarrow \text{RED}$

$d[v] \leftarrow d[u] + 1$

$p[v] \leftarrow u$

$Q.\text{enqueue}(v)$

$\text{colour}[u] \leftarrow \text{GRAY}$

# Summary of Topics

1. Binary Search Trees
2. Sorting
3. Graphs