Final Exam

- Tue, 17 Apr 2012 19:00 – 22:00 LAS B
- Closed Book
- Format similar to midterm
- Will cover whole course, with emphasis on material after midterm (maps, hashing, 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 I provide (available early next week).
- Do extra practice problems from the textbook.
- Review the midterm and solutions for practice writing this kind of exam.
- Practice writing clear, succint pseudocode!
- 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 see Ron, Paria or me by appointment

End of Term Review

Summary of Topics

1. Maps
2. Binary Search Trees
3. Sorting
4. Graphs

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

The Map ADT

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M
  - entries(): returns an iterator of the entries in M

Performance of a List-Based Map

- Performance:
  - put, get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for small maps

Hash Tables

- A hash table is a data structure that can be used to make map operations faster.
- While worst-case is still O(n), average case is typically O(1).

Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$

- Example:

$$h(x) = x \bmod N$$

is a hash function for integer keys

- The integer $h(x)$ is called the hash value of key x

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N

- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Polynomial Hash Codes

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0 \, a_1 \, \ldots \, a_{n-1}$$

  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
  - Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
    - The following polynomials are successively computed, each from the previous one in $O(1)$ time

      $$p_0(z) = a_{n-1}$$

      $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

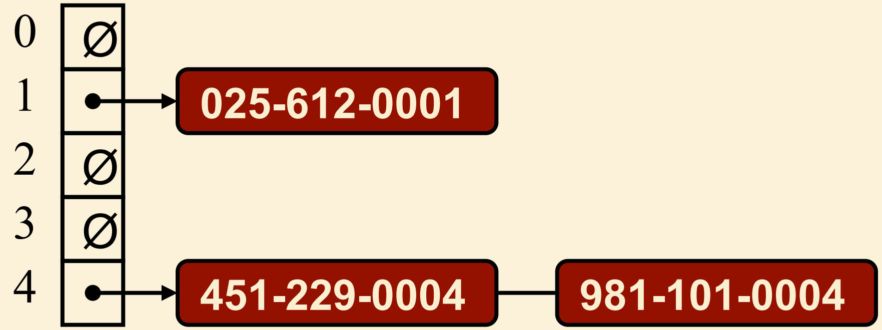
  - We have $p(z) = p_{n-1}(z)$

Compression Functions

- Division:
  - $\boldsymbol{h}_2(\boldsymbol{y}) = \boldsymbol{y} \bmod \boldsymbol{N}$
  - The size $\boldsymbol{N}$ of the hash table is usually chosen to be a prime
- Multiply, Add and Divide (MAD):
  - $\boldsymbol{h}_2(\boldsymbol{y}) = (\boldsymbol{ay} + \boldsymbol{b}) \bmod \boldsymbol{N}$
  - $\boldsymbol{a}$ and $\boldsymbol{b}$ are nonnegative integers such that
    $\boldsymbol{a} \bmod \boldsymbol{N} \neq 0$
  - Otherwise, every integer would map to the same value $\boldsymbol{b}$

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**:
  - Let each cell in the table point to a linked list of entries that map there
  - Separate chaining is simple, but requires additional memory outside the table



Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, so that future collisions cause a longer sequence of probes

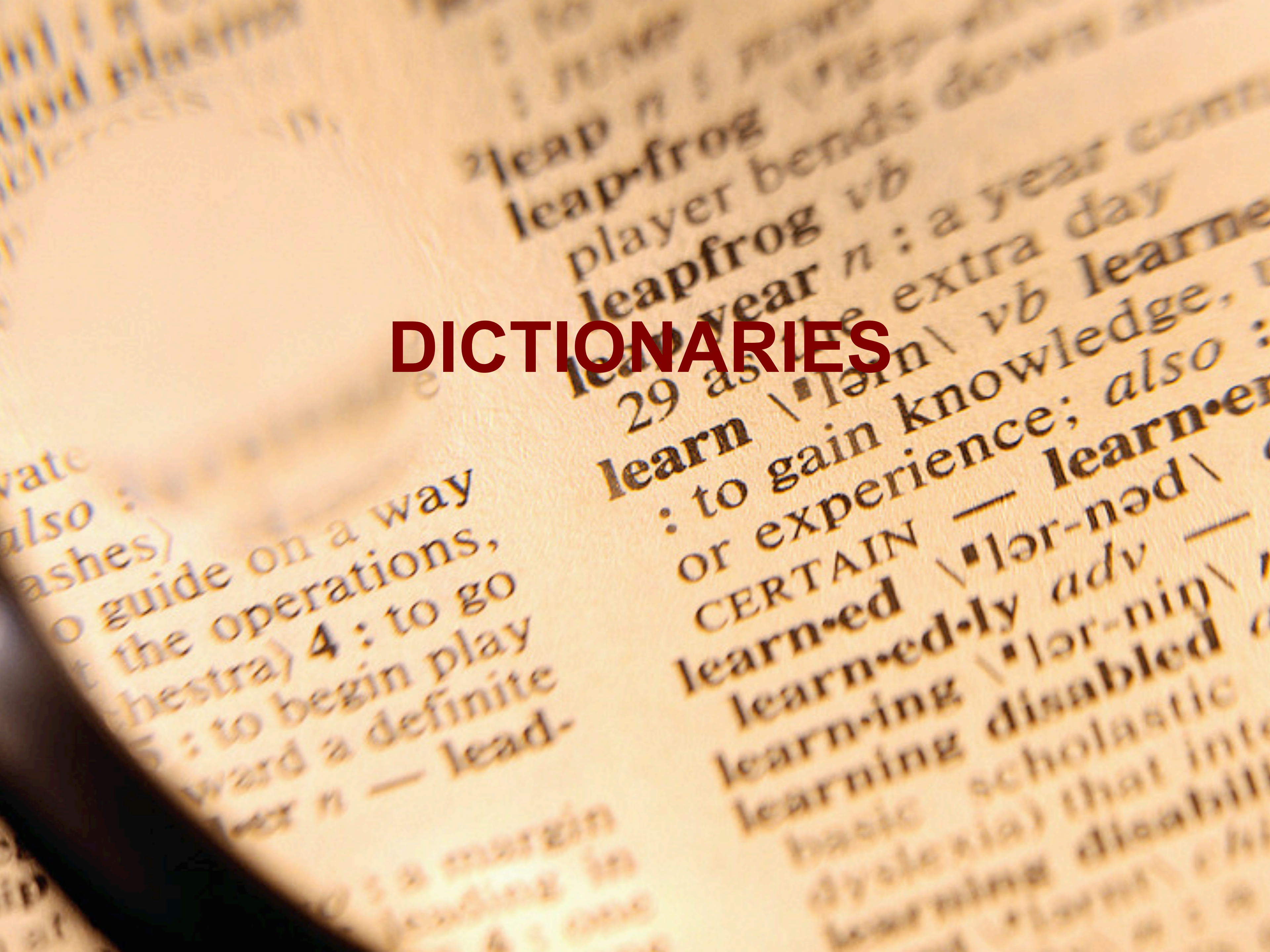
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\lambda = n/N$ affects the performance of a hash table
  - For separate chaining, performance is typically good for $\lambda < 0.9$.
  - For open addressing , performance is typically good for $\lambda < 0.5$.
  - java.util.HashMap maintains $\lambda < 0.75$
- Separate chaining is typically as fast or faster than open addressing.

DICTIONARIES



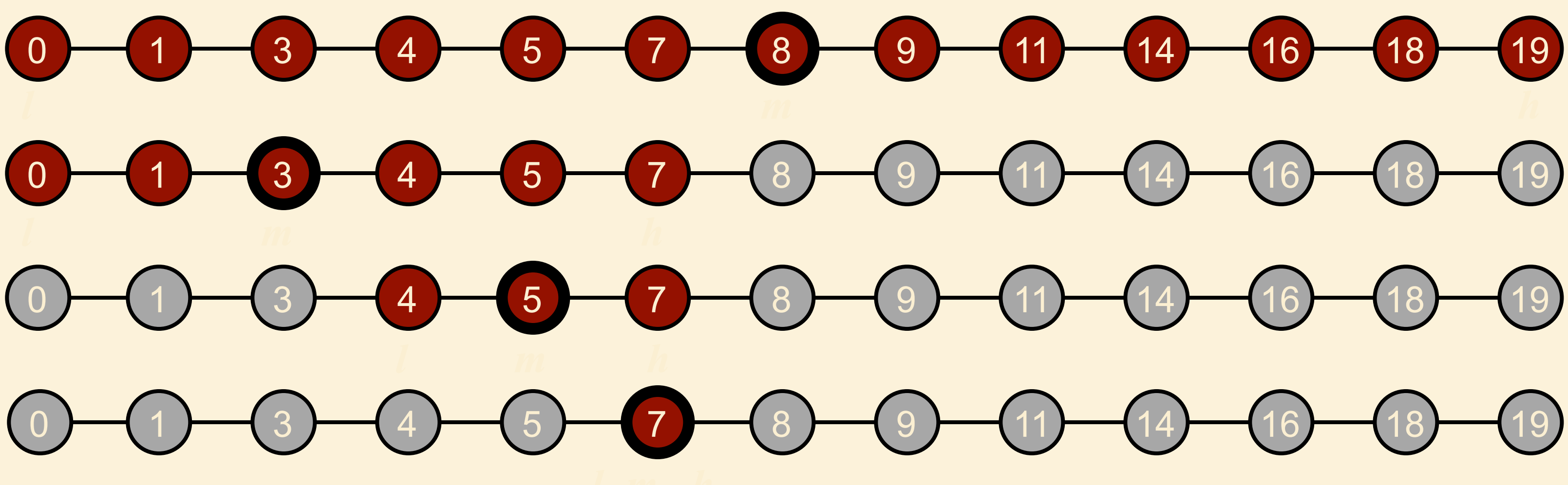
Dictionary ADT

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

- Dictionary ADT methods:
  - get(k): if the dictionary has at least one entry with key k, returns one of them, else, returns null
  - getAll(k): returns an iterable collection of all entries with key k
  - put(k, v): inserts and returns the entry (k, v)
  - remove(e): removes and returns the entry e. Throws an exception if the entry is not in the dictionary.
  - entrySet(): returns an iterable collection of the entries in the dictionary
  - size(), isEmpty()

Dictionaries & Ordered Search Tables

- If keys obey a total order relation, can represent dictionary as an ordered search table stored in an array.
- Can then support a fast find(k) using **binary search.**
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
  - Example: find(7)



```
BinarySearch(A[1..n], key)
<precondition>:  A[1..n] is sorted in non-decreasing order
<postcondition>: If key is in A[1..n], algorithm returns its location
p = 1, q = n
while q > p
    <loop-invariant>: If key is in A[1..n], then key is in A[p..q]
    mid = ⌊(p + q) / 2⌋
    if key ≤ A[mid]
        q = mid
    else
        p = mid + 1
    end
end
if key = A[p]
    return(p)
else
    return("Key not in list")
end
```

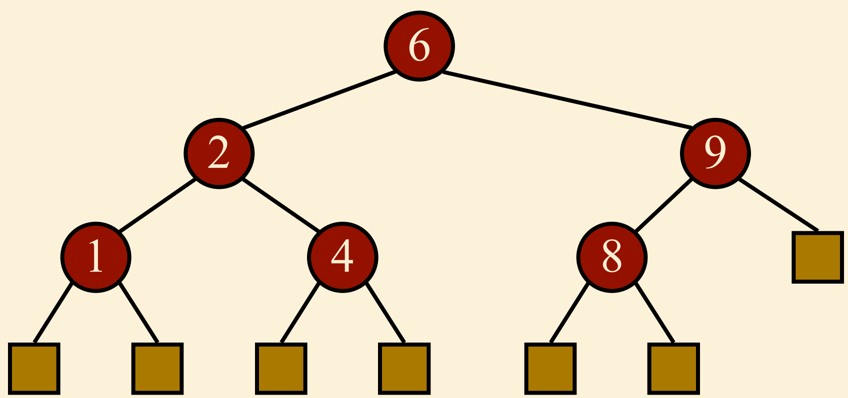
Topic 1. Binary Search Trees

Binary Search Trees

- Insertion
- Deletion
- AVL Trees
- Splay Trees

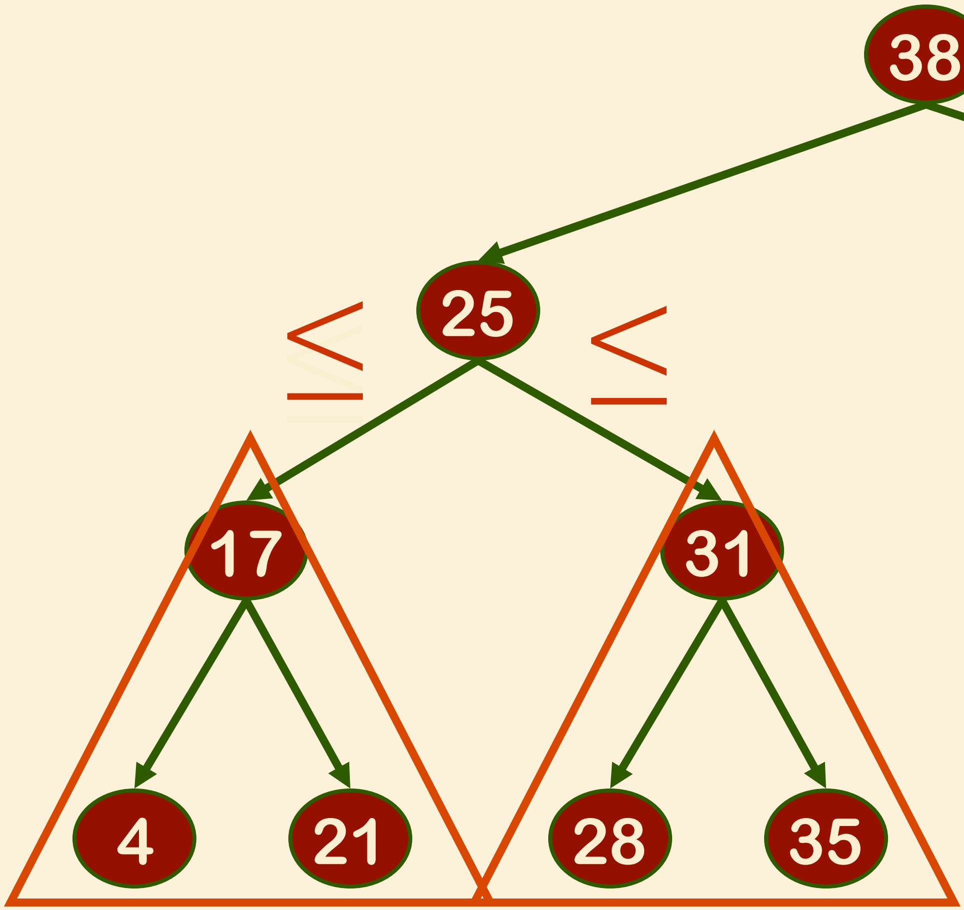
Binary Search Trees

- 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 $key(u) \leq key(v) \leq key(w)$
- 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 trees visits the keys in increasing order
- Binary search trees are ideal for maps or dictionaries with ordered keys.



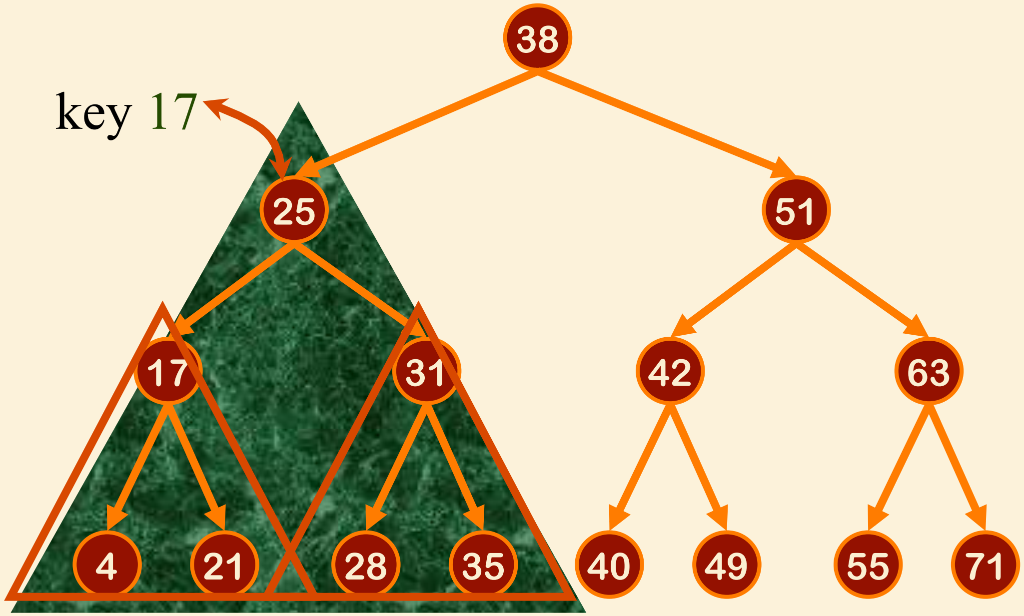
Binary Search Tree

All nodes in left subtree ≤ Any node ≤ All nodes in right subtree



Search: Define Step

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



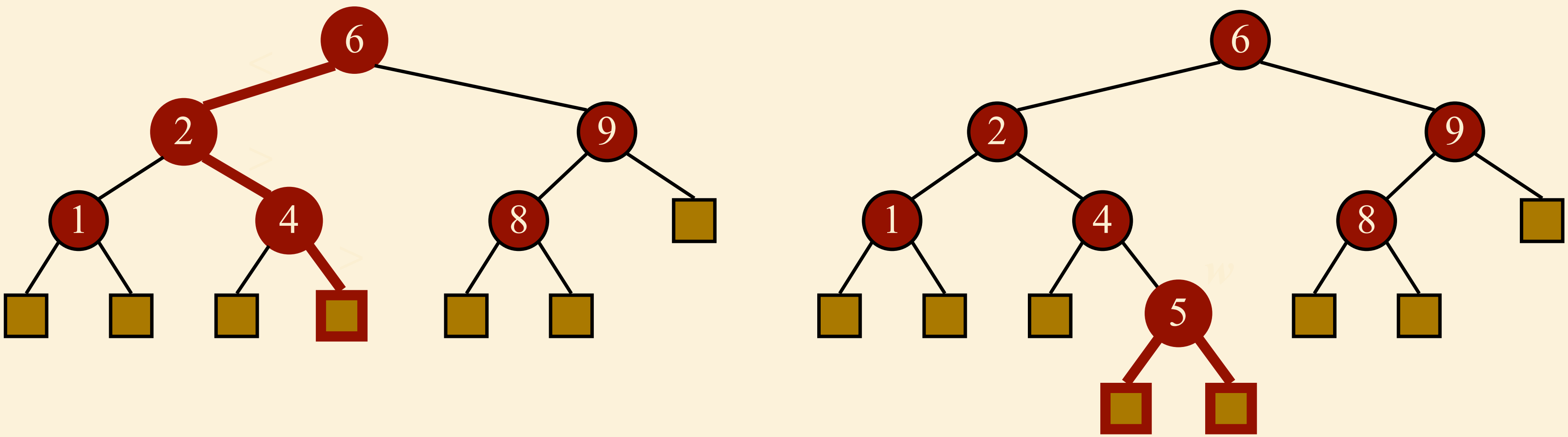
If key < root, then key is in left half.

If key = root, then key is found

If key > root, then key is in right half.

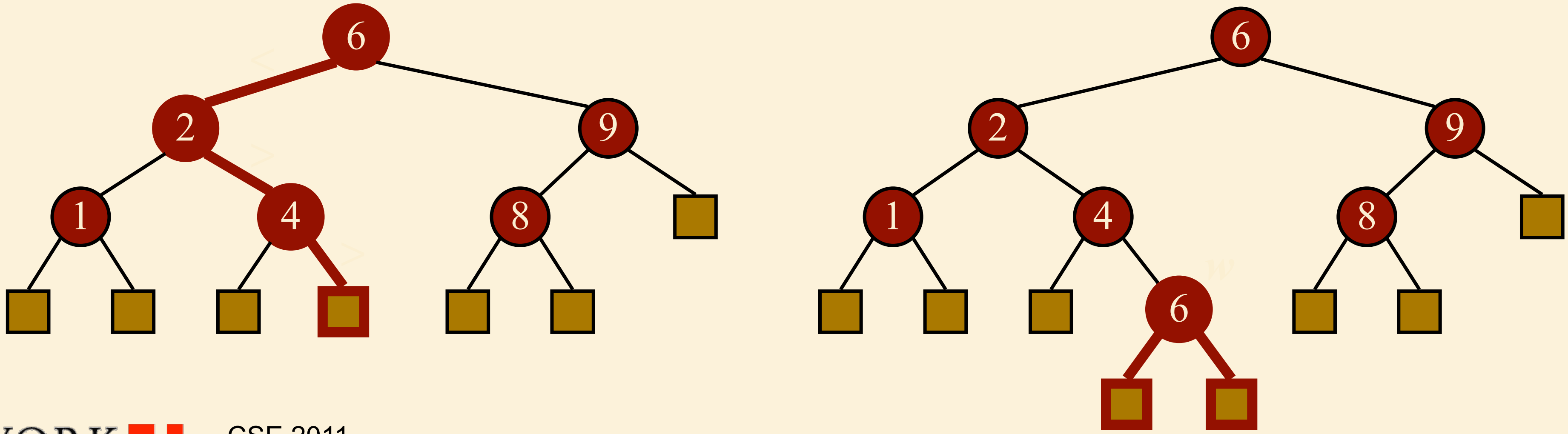
Insertion (For Dictionary)

- 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



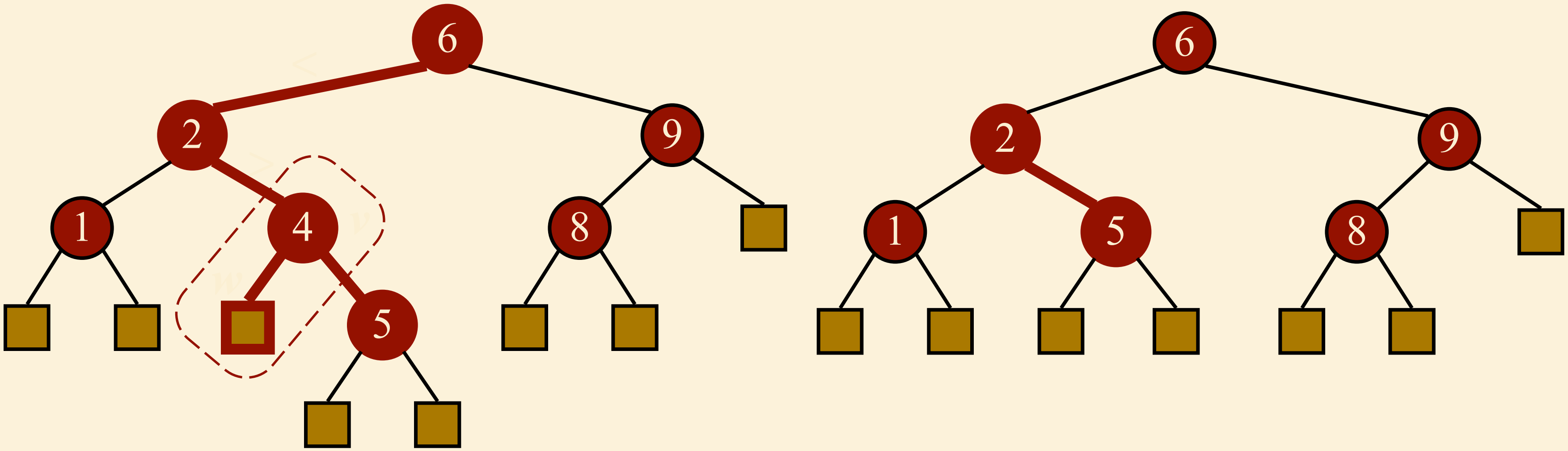
Insertion

- 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



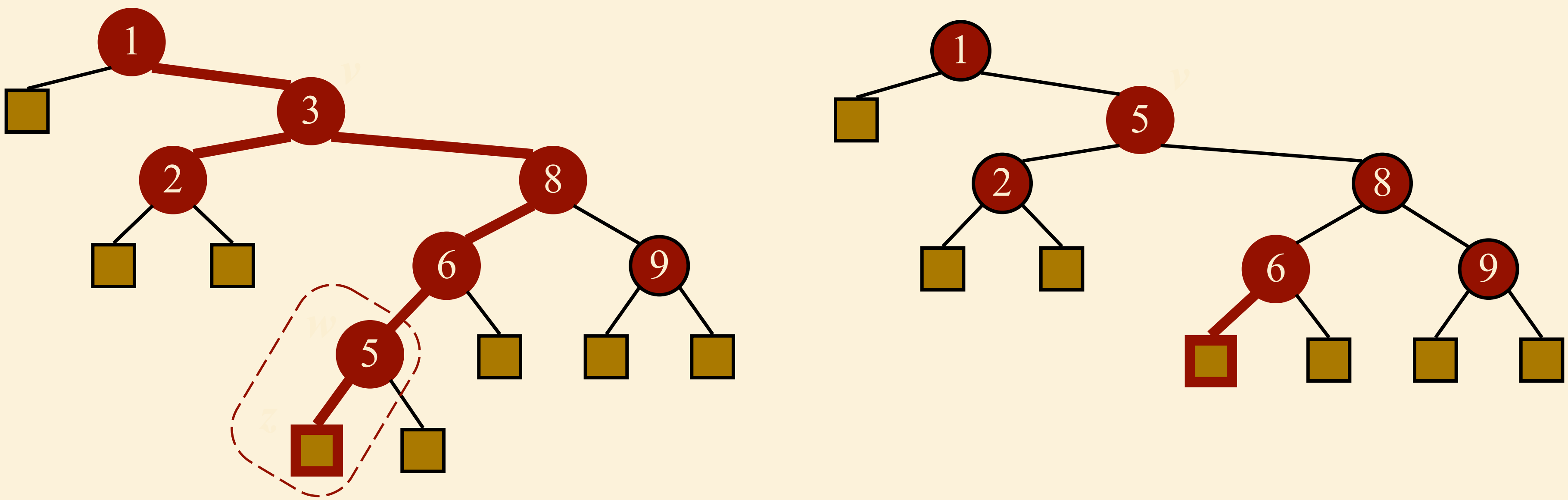
Deletion

- 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



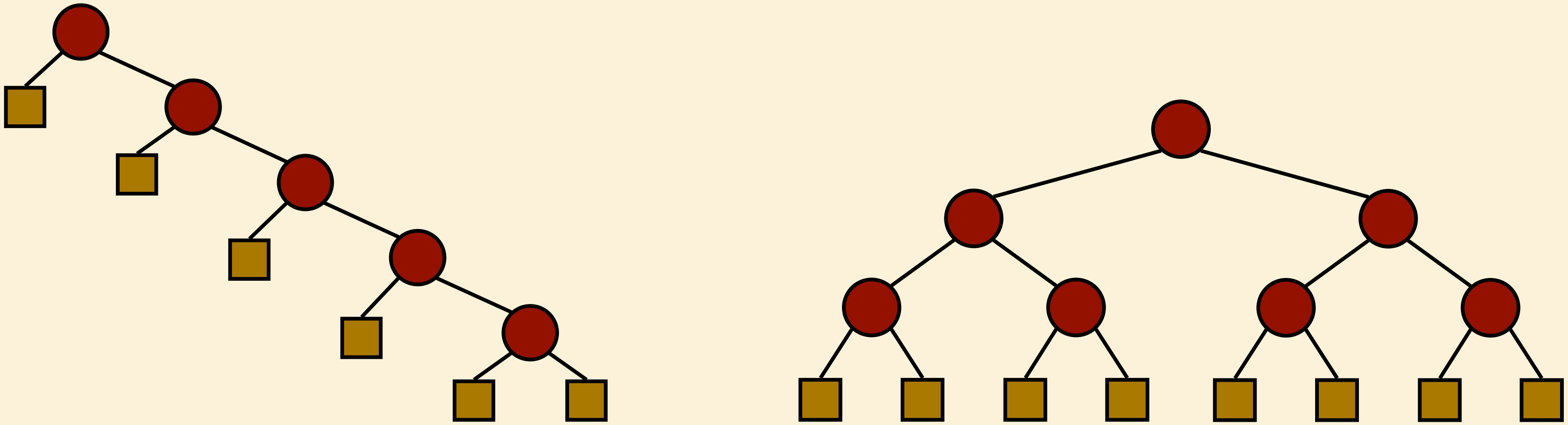
Deletion (cont.)

- 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



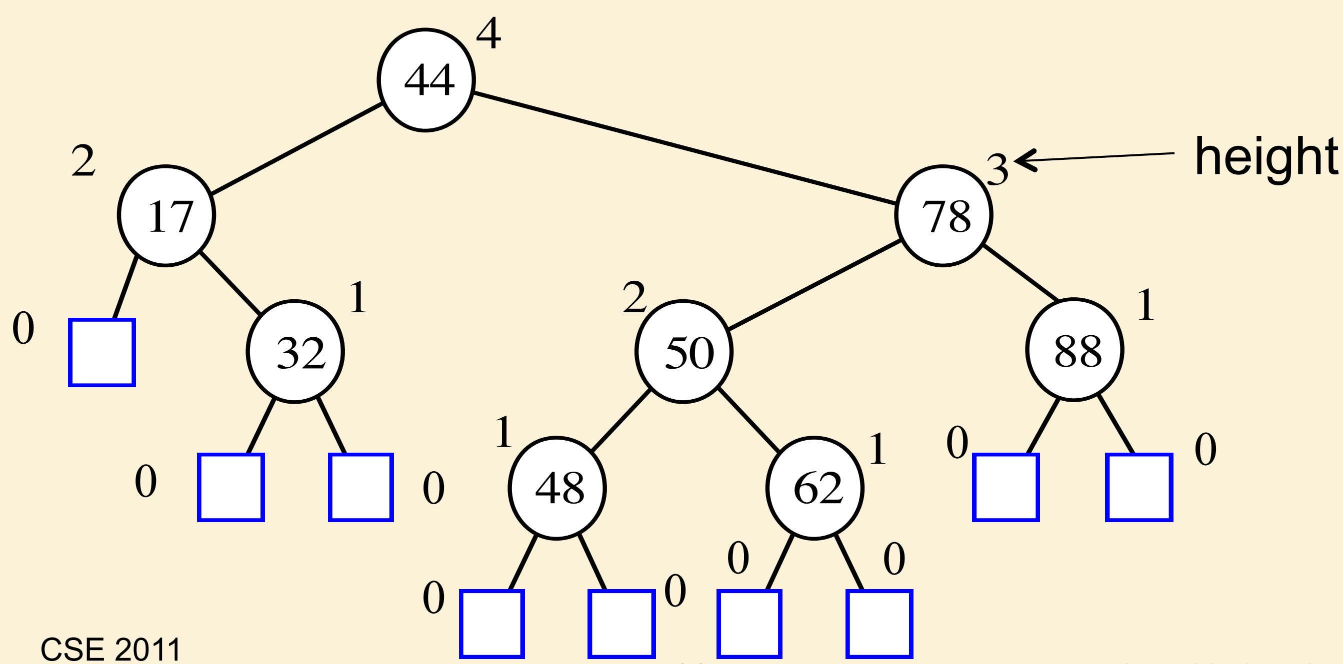
Performance

- 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.

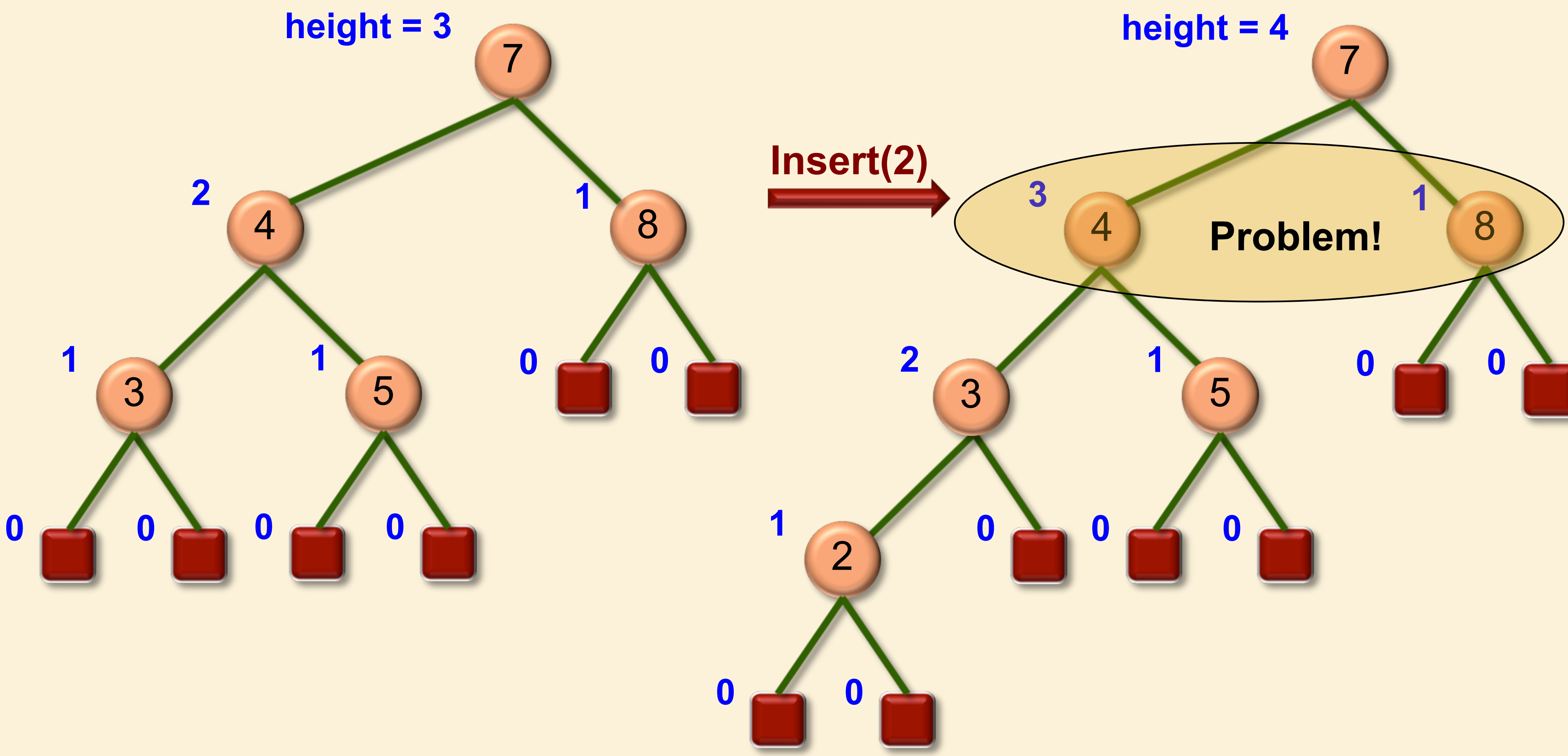


Height of an AVL Tree

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

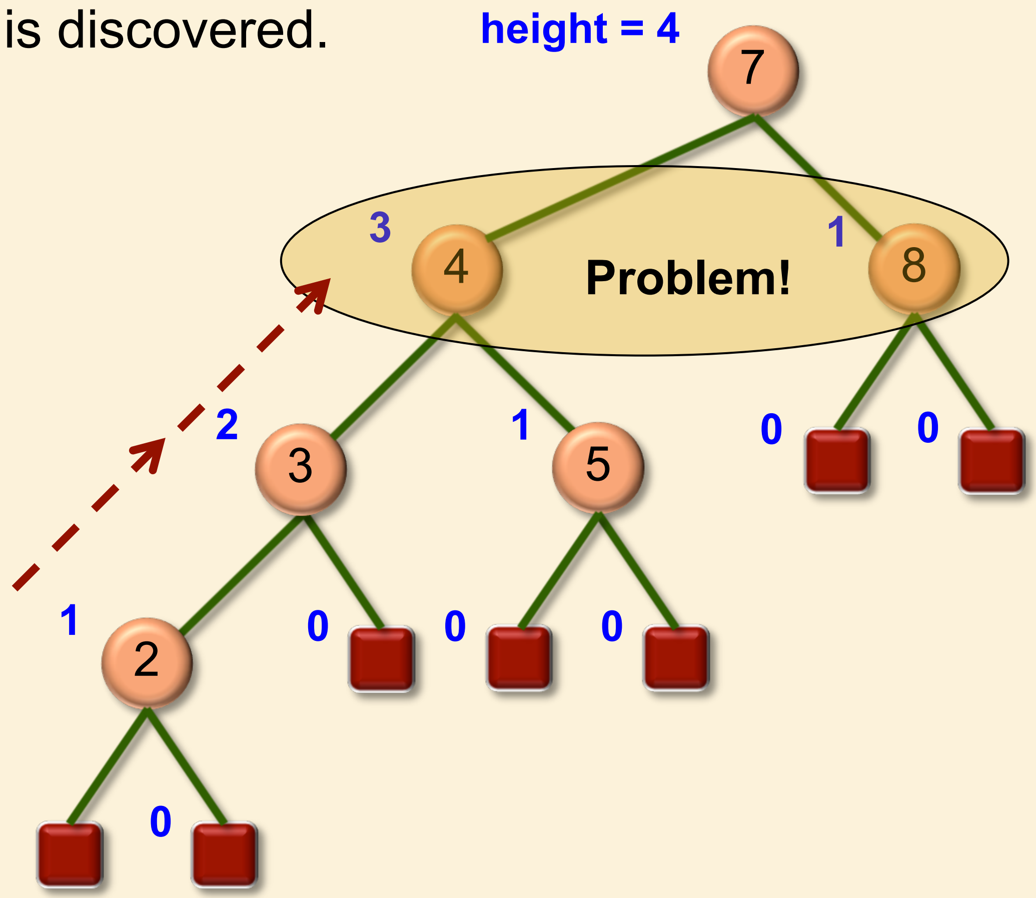
Insertion

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



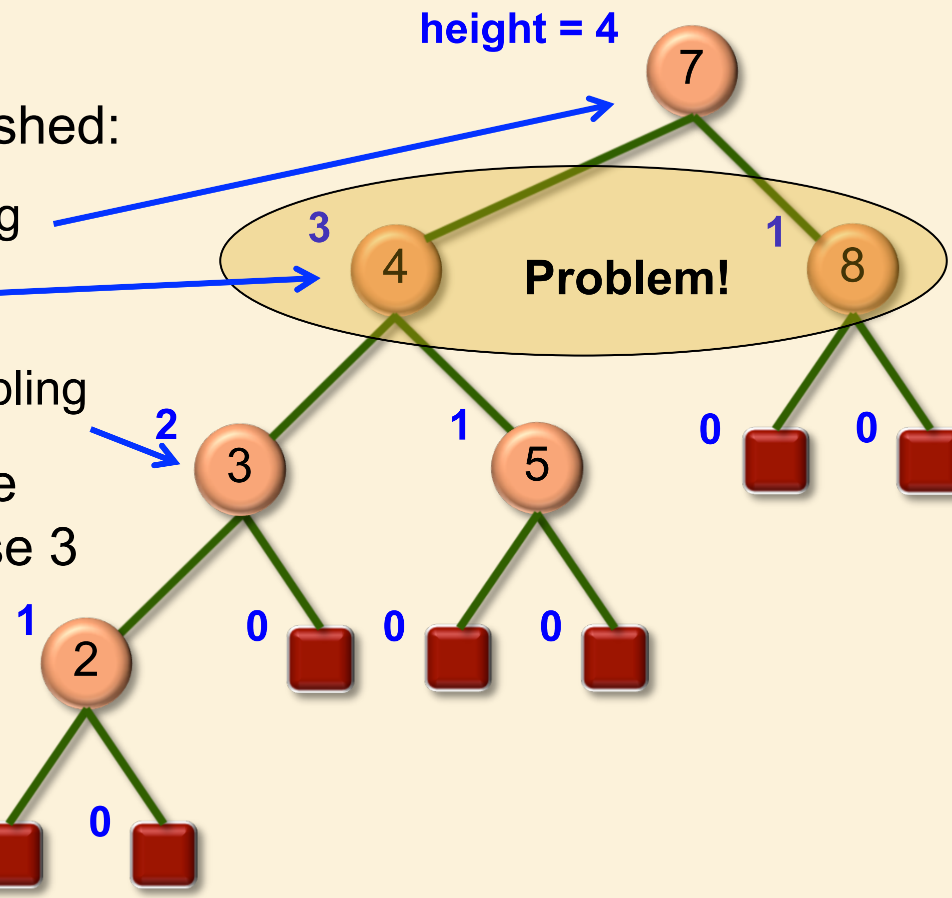
Insertion: Rebalancing Strategy

- Step 1: Search
  - Starting at the inserted node, traverse toward the root until an imbalance is discovered.



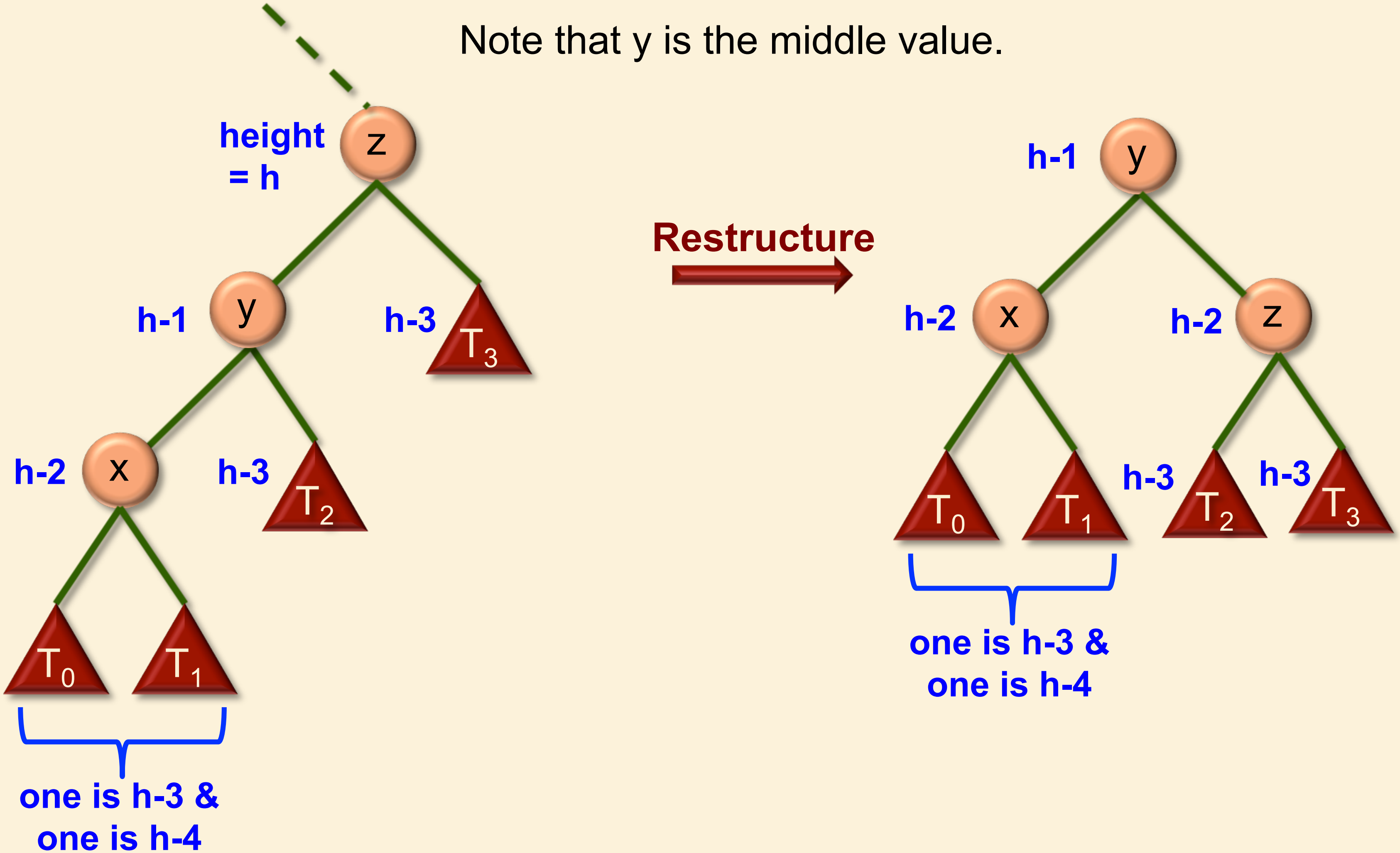
Insertion: Rebalancing Strategy

- 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



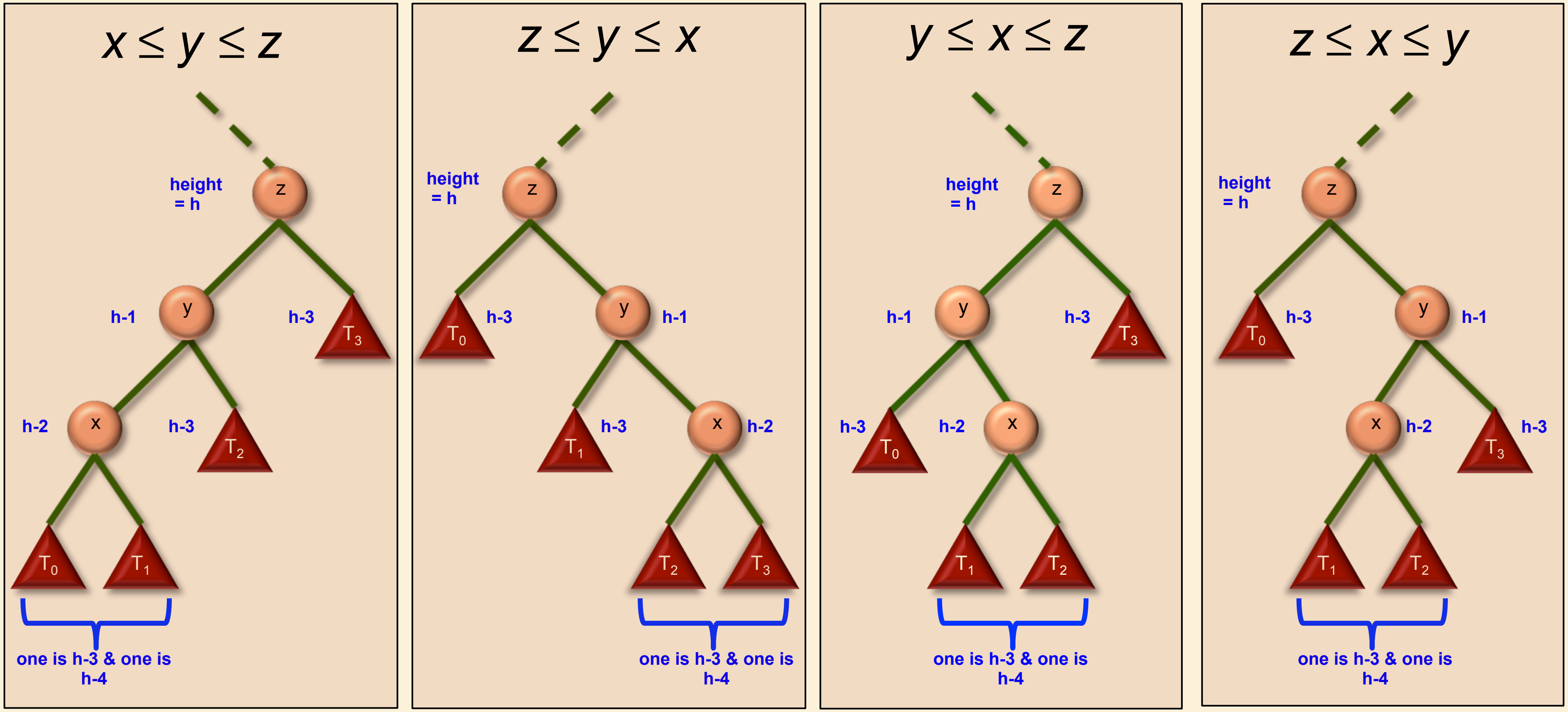
Insertion: Trinode Restructuring Example

Note that y is the middle value.



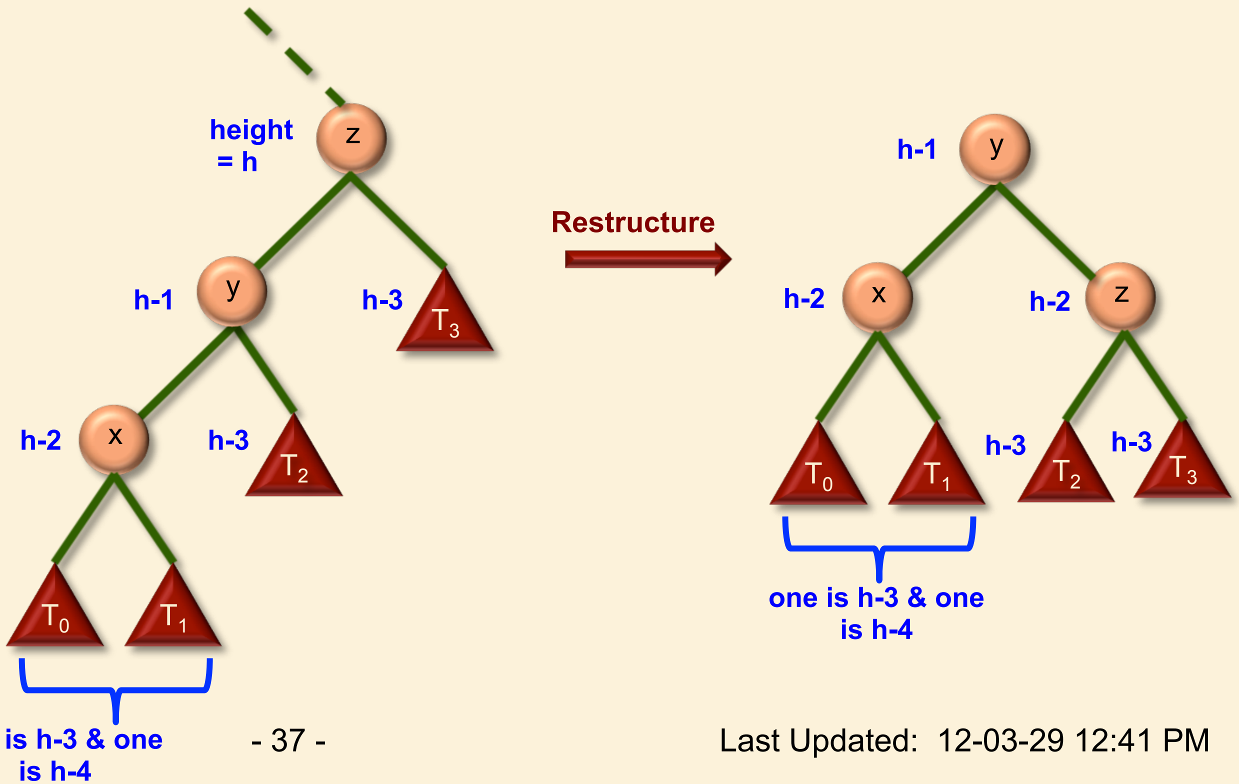
Insertion: Trinode Restructuring - 4 Cases

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



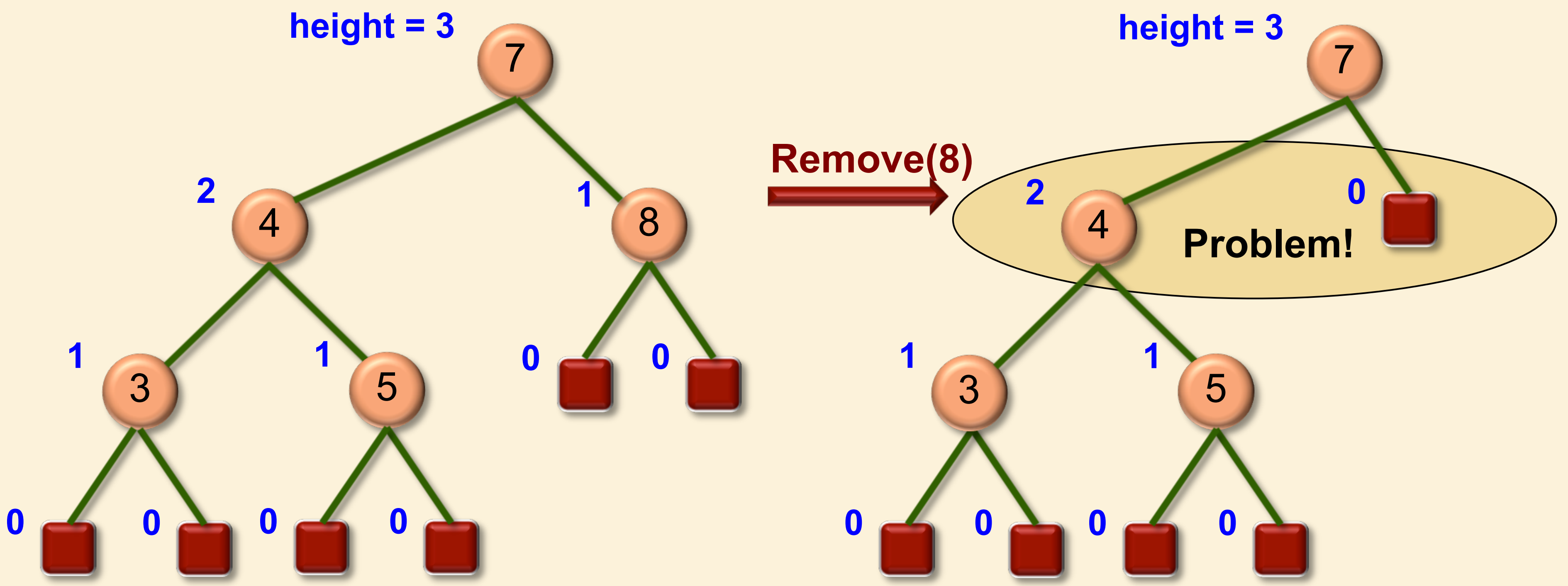
Insertion: Trinode Restructuring - The Whole Tree

- 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.



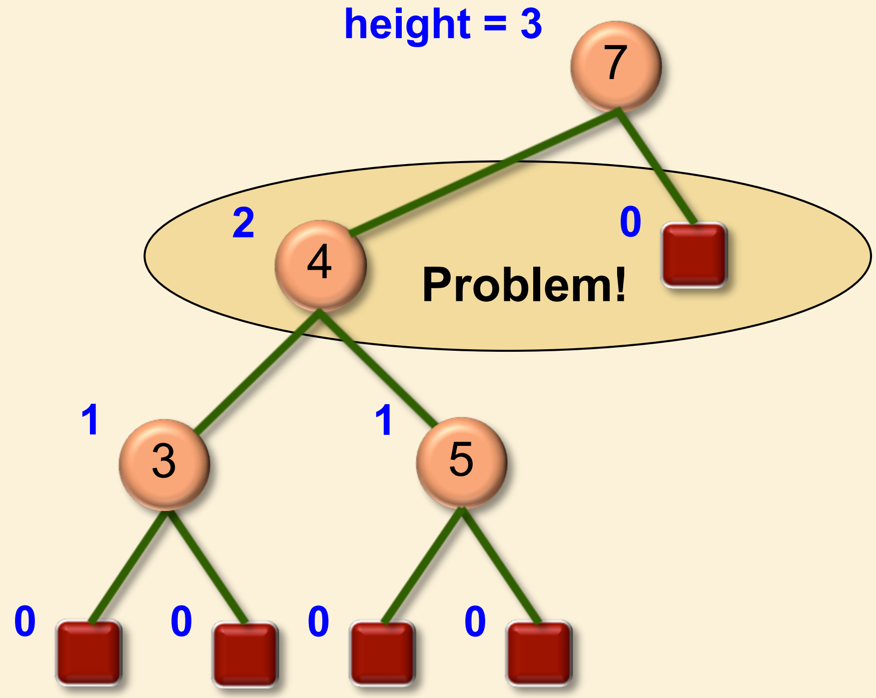
Removal

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



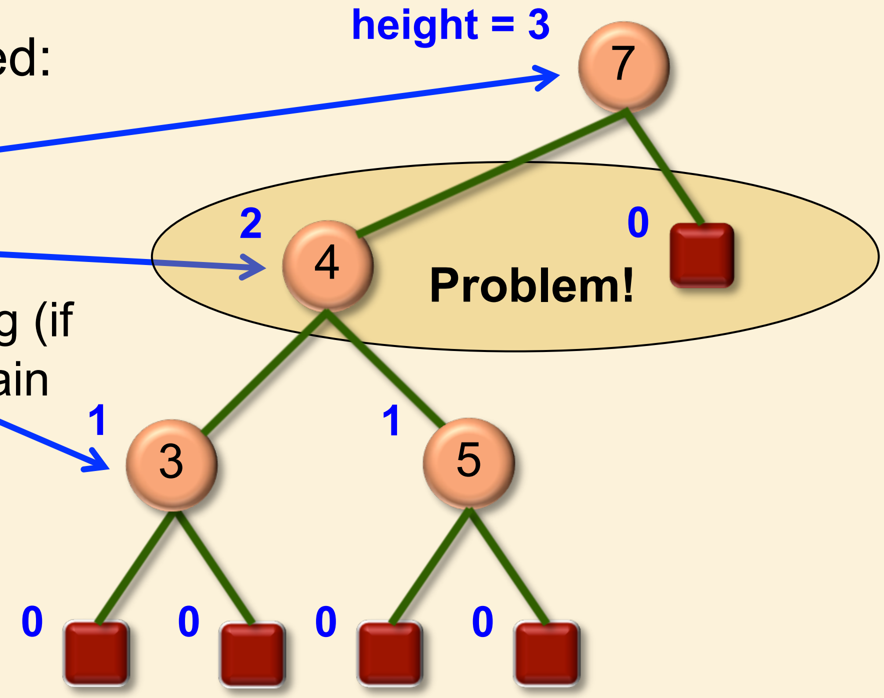
Removal: Rebalancing Strategy

- Step 1: Search
  - Starting at the location of the removed node, traverse toward the root until an imbalance is discovered.



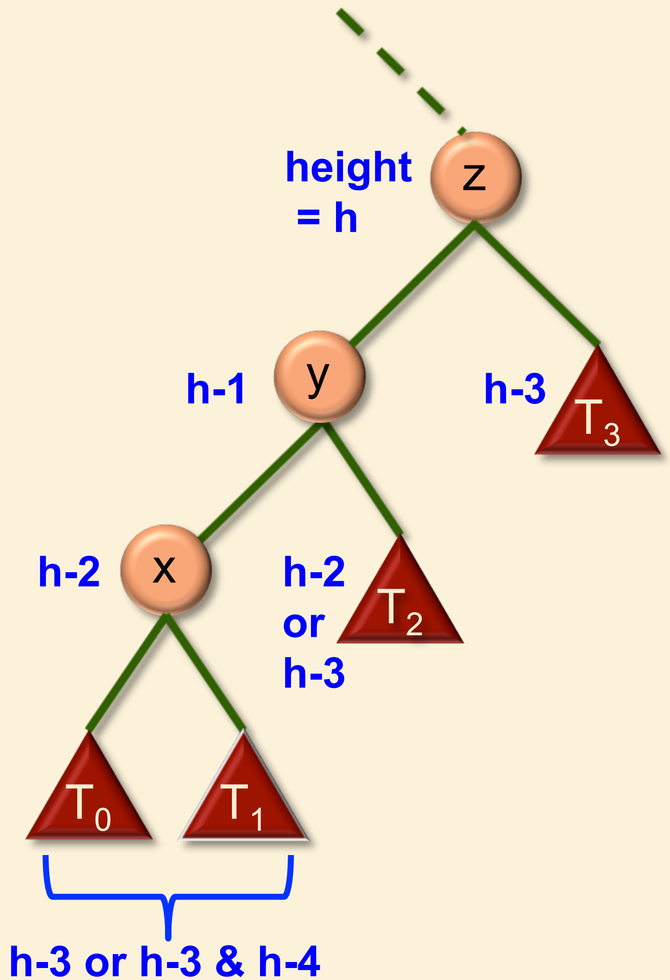
Removal: Rebalancing Strategy

- 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)



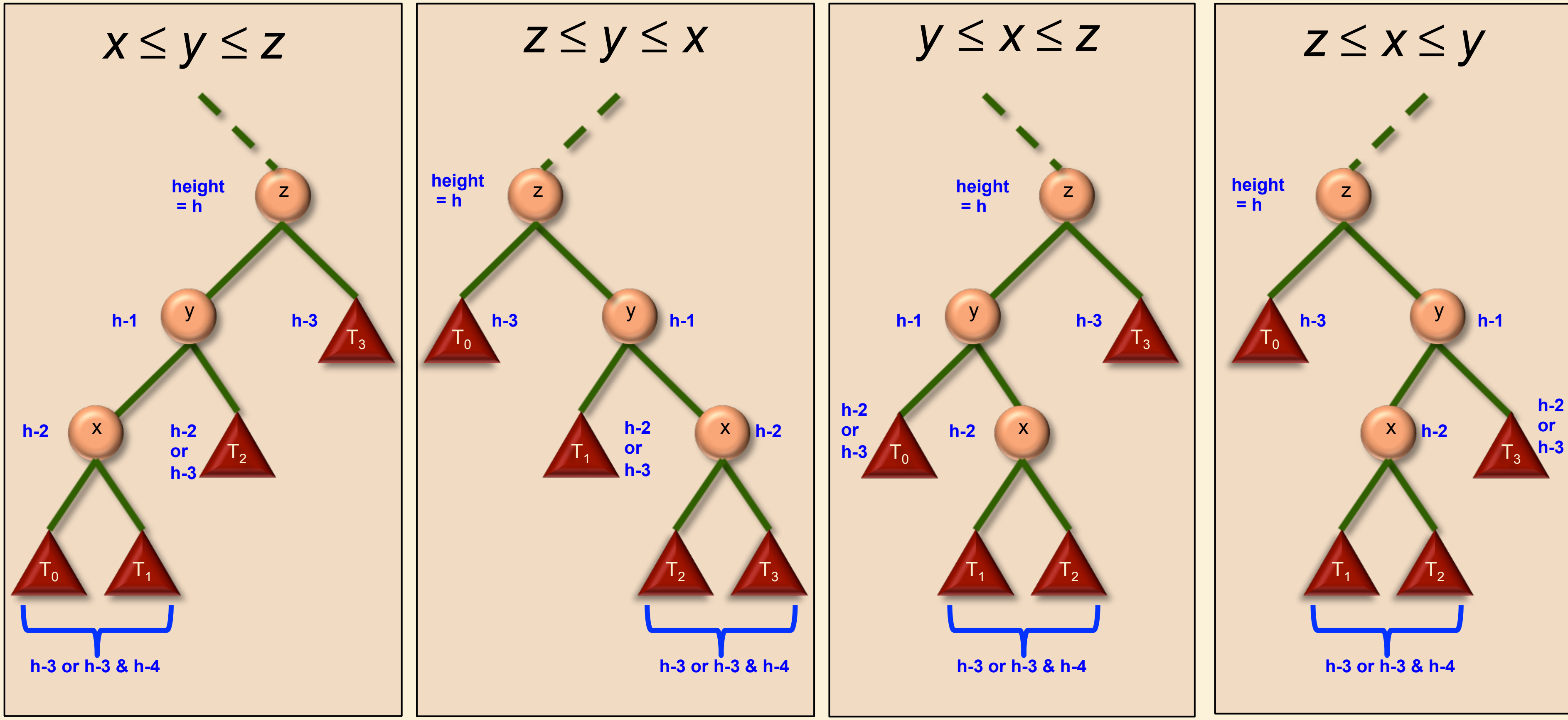
Removal: Rebalancing Strategy

- 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 $T_0 - T_3$ are attached at the appropriate places.
  - Although the subtrees $T_0 - T_3$ can differ in height by up to 2, after restructuring, sibling subtrees will differ by at most 1.



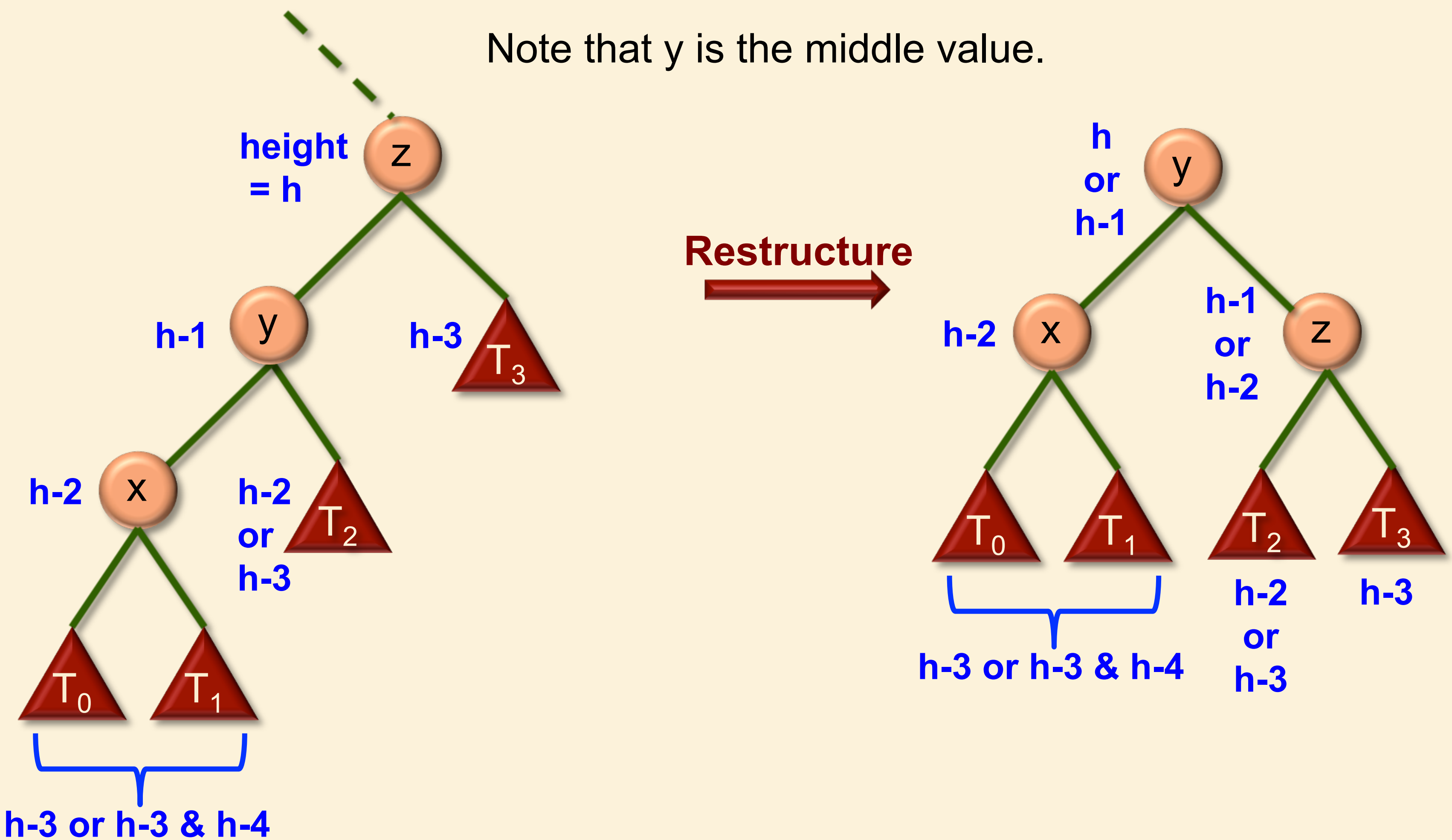
Removal: Trinode Restructuring - 4 Cases

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



Removal: Trinode Restructuring - Case 1

Note that y is the middle value.



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.

Splay Trees

- Self-balancing BST
- Invented by Daniel Sleator and Bob Tarjan
- Allows quick access to recently accessed elements
- Bad: worst-case O(n)
- Good: average (amortized) case O(log n)
- Often perform better than other BSTs in practice

D. Sleator

R. Tarjan

Splaying

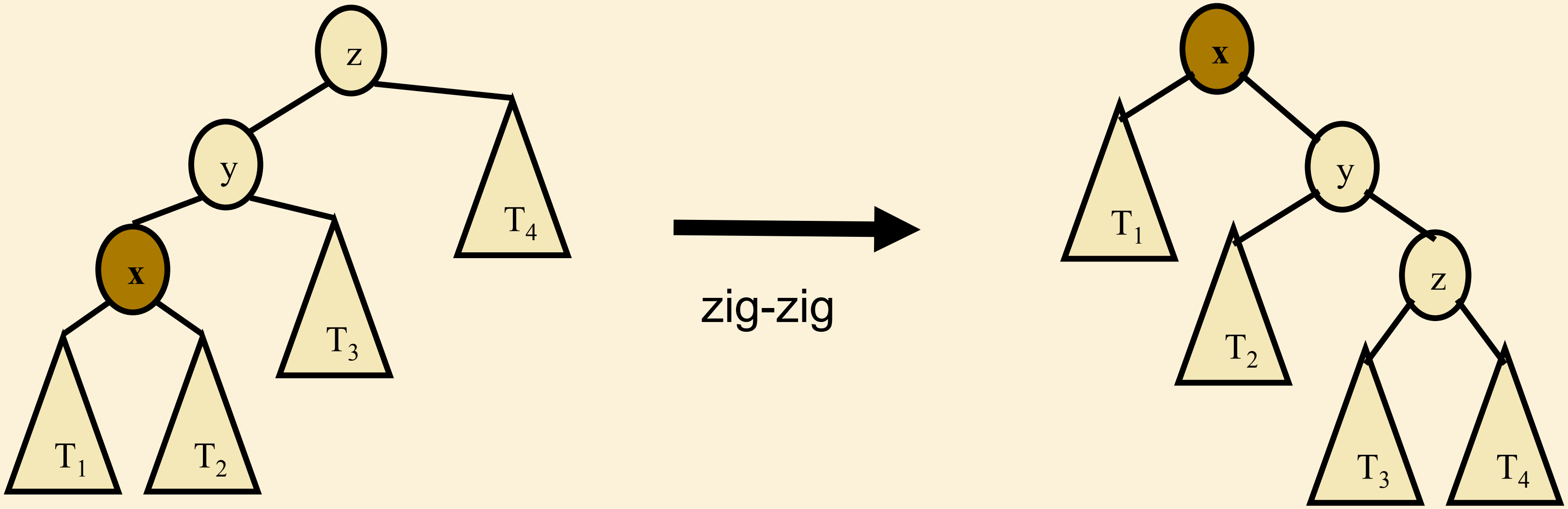
- Splaying is an operation performed on a node that iteratively moves the node to the root of the tree.
- In splay trees, each BST operation (find, insert, remove) is augmented with a splay operation.
- In this way, recently searched and inserted elements are near the top of the tree, for quick access.

3 Types of Splay Steps

- Each splay operation on a node consists of a sequence of splay steps.
- Each splay step moves the node up toward the root by 1 or 2 levels.
- There are 2 types of step:
  - Zig-Zig
  - Zig-Zag
  - Zig

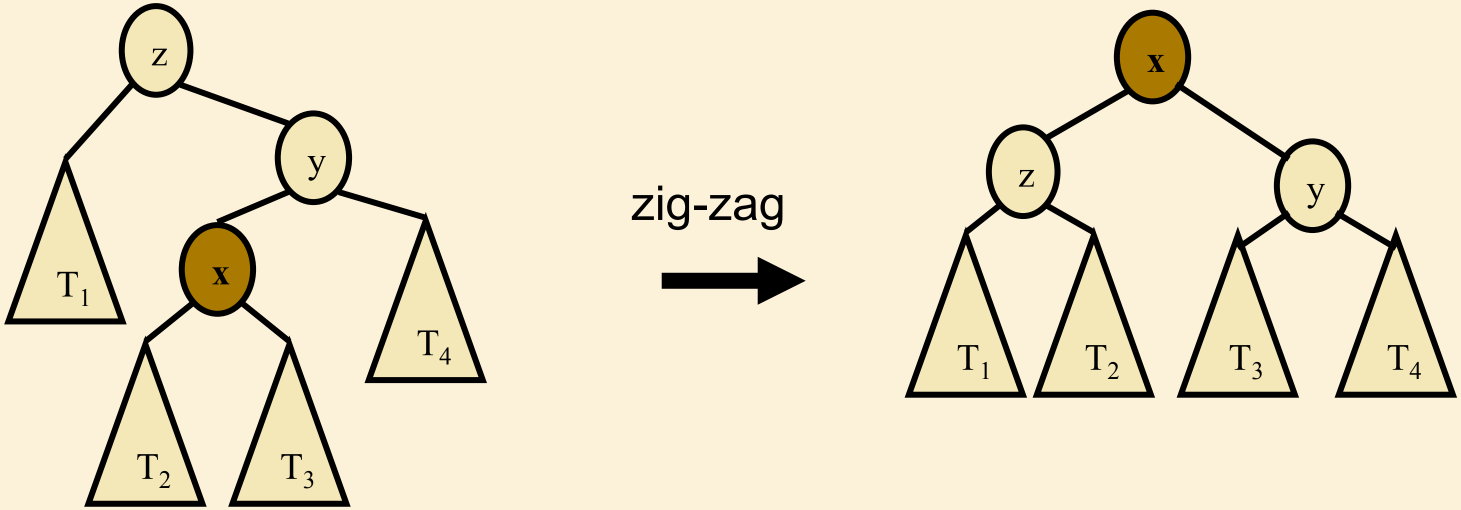
Zig-Zig

- Performed when the node x forms a linear chain with its parent and grandparent.
  - i.e., right-right or left-left



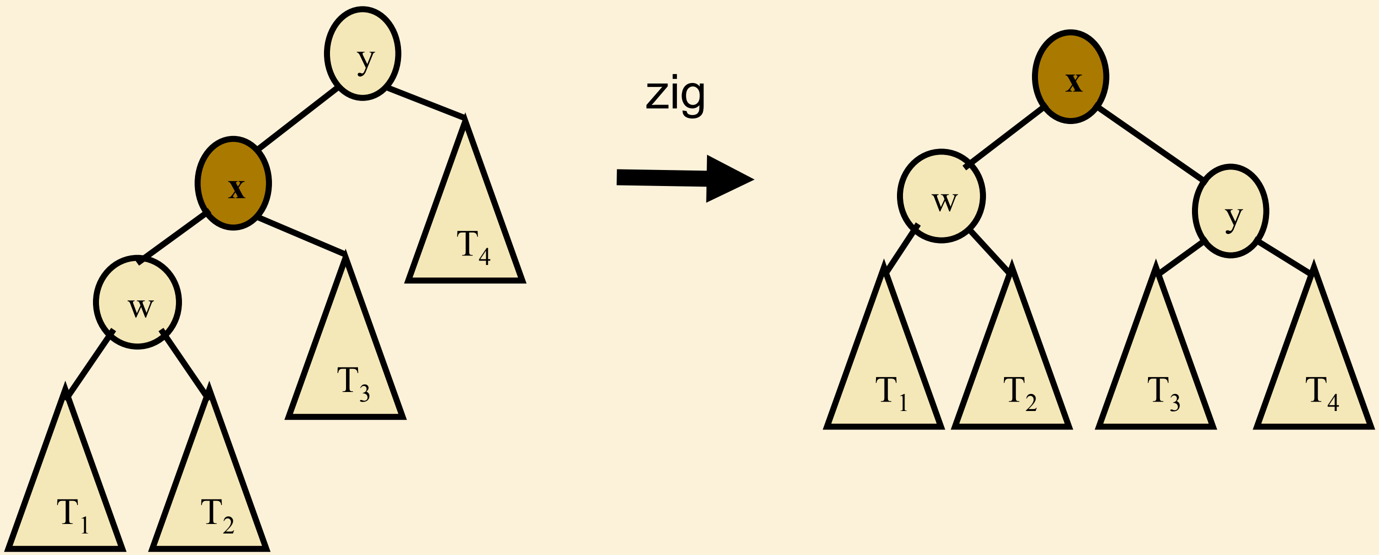
Zig-Zag

- Performed when the node x forms a non-linear chain with its parent and grandparent
  - i.e., right-left or left-right



Zig

- Performed when the node x has no grandparent
  - i.e., its parent is the root



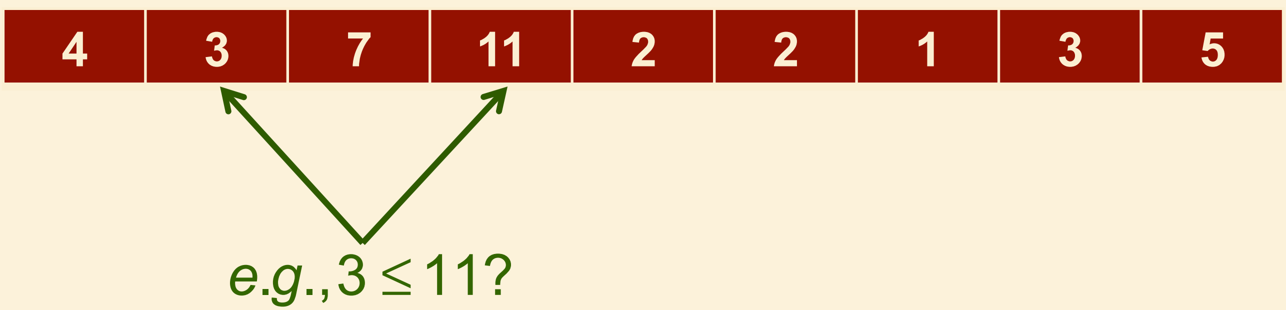
Topic 2. Sorting

Sorting Algorithms

- Comparison Sorting
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Heap Sort
  - Quick Sort
- Linear Sorting
  - Counting Sort
  - Radix Sort
  - Bucket Sort

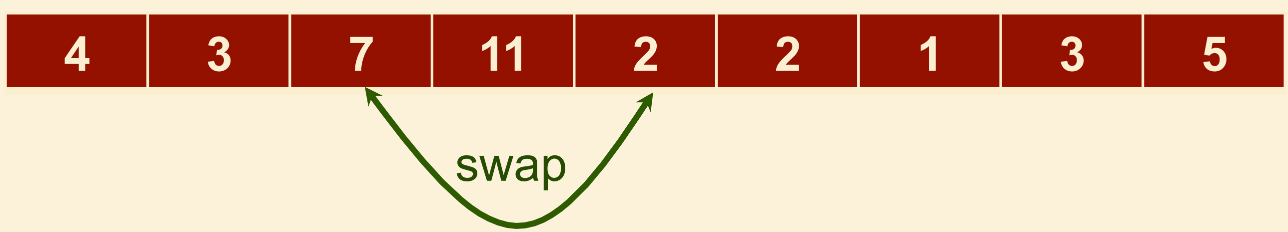
Comparison Sorts

- 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.



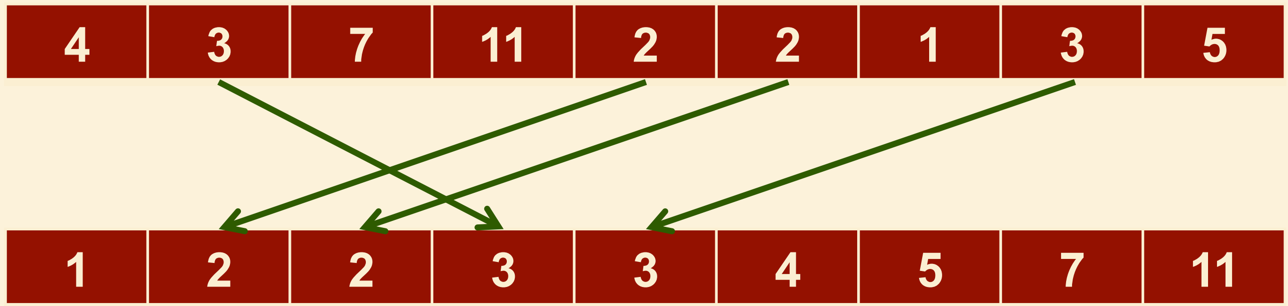
Sorting Algorithms and Memory

- 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.)



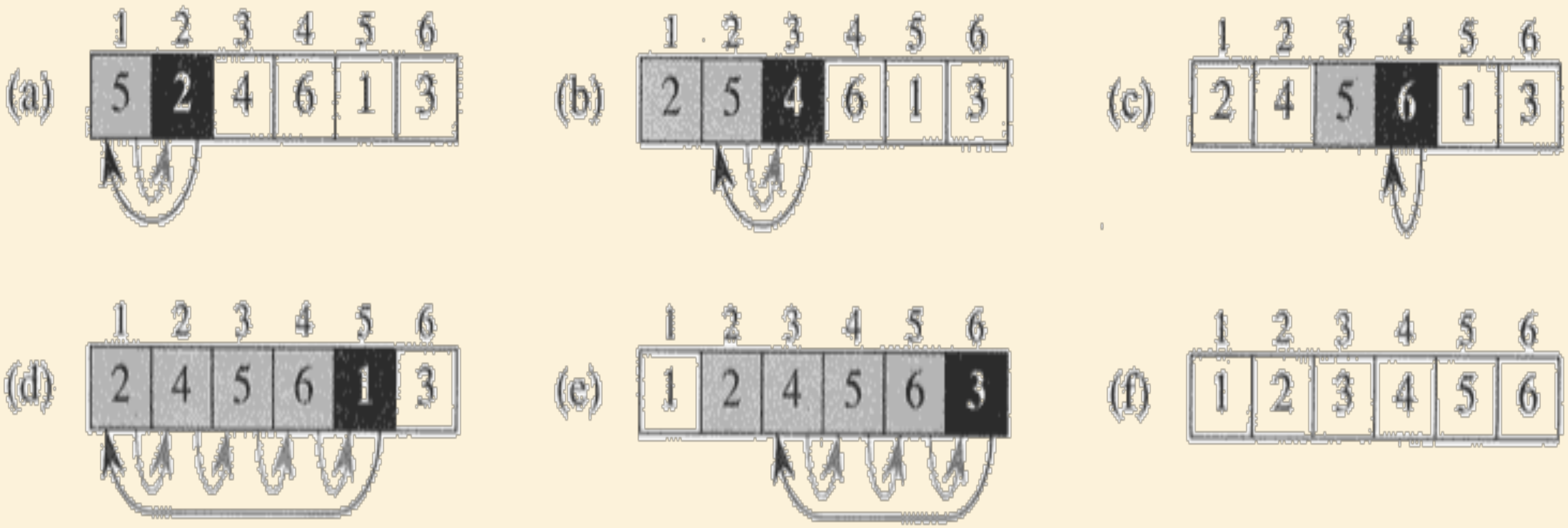
Selection Sort

- 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

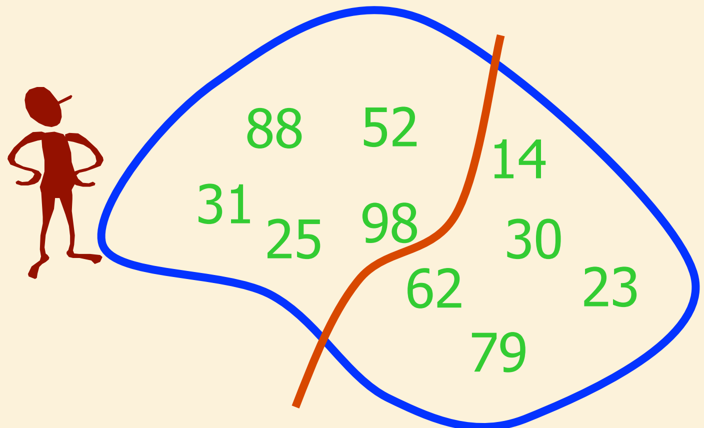
Bubble Sort

- 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.

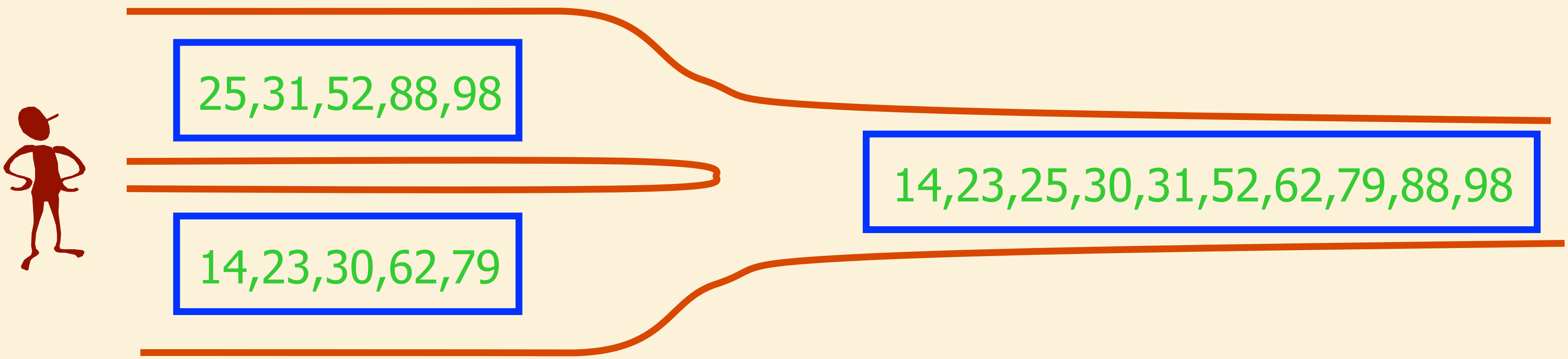
25,31,52,88,98

Get one friend to
sort the second half.

14,23,30,62,79

Merge Sort

Merge two sorted lists into one

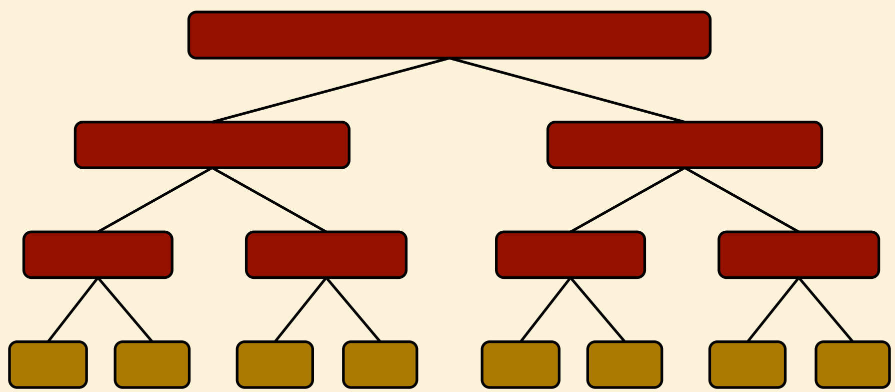


Analysis of Merge-Sort

- 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 or 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

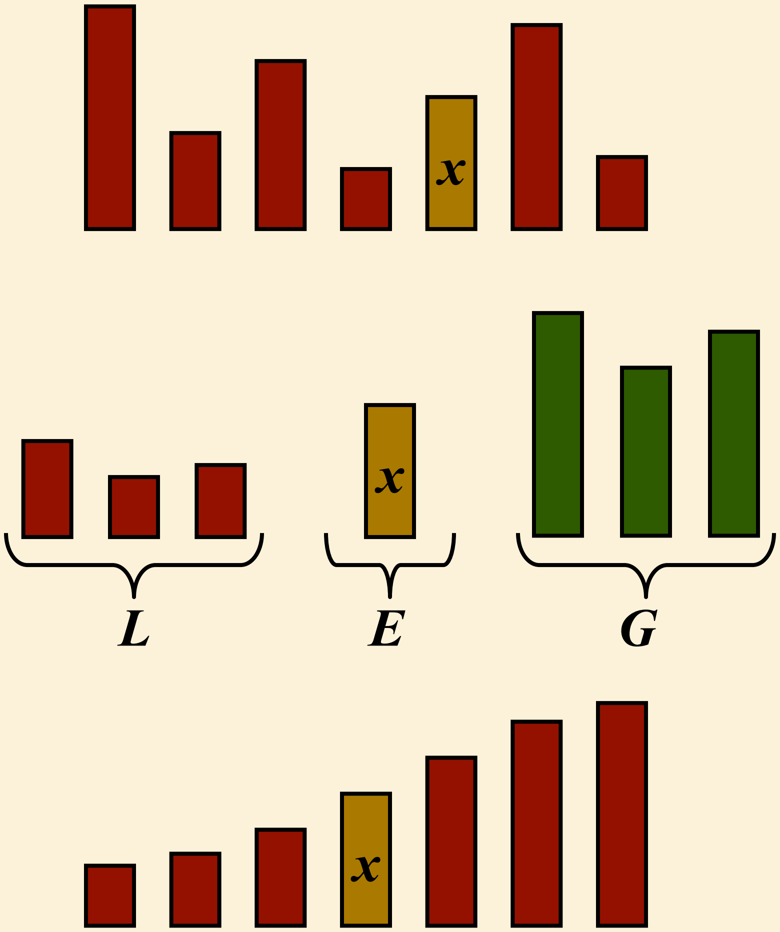
Heap-Sort Running Time

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

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

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)

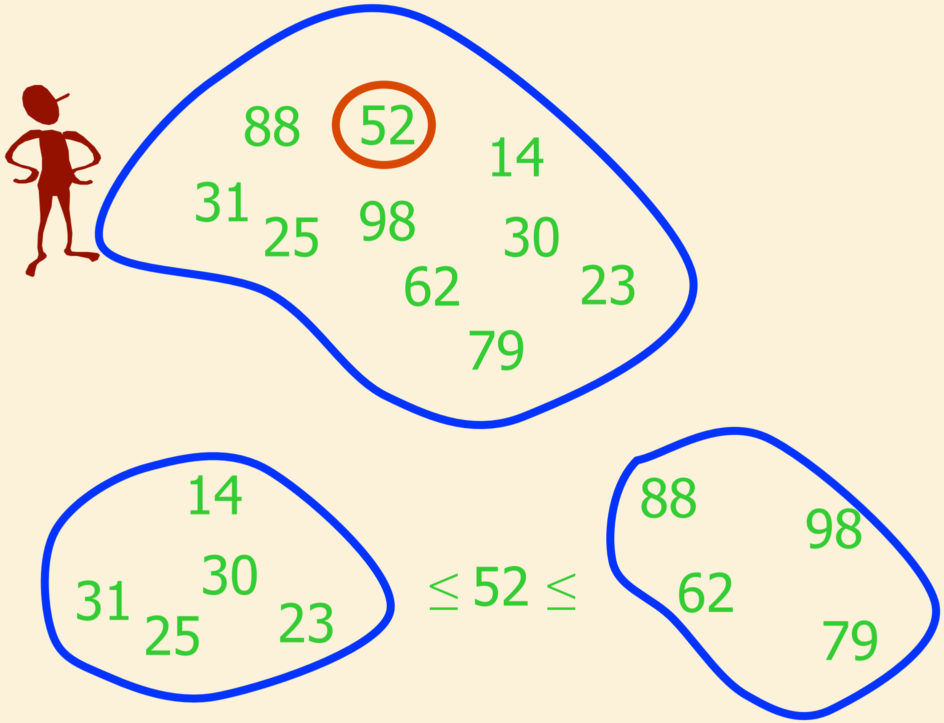
QuickSort(G)

S = (L, E, G)

In-Place Quick-Sort

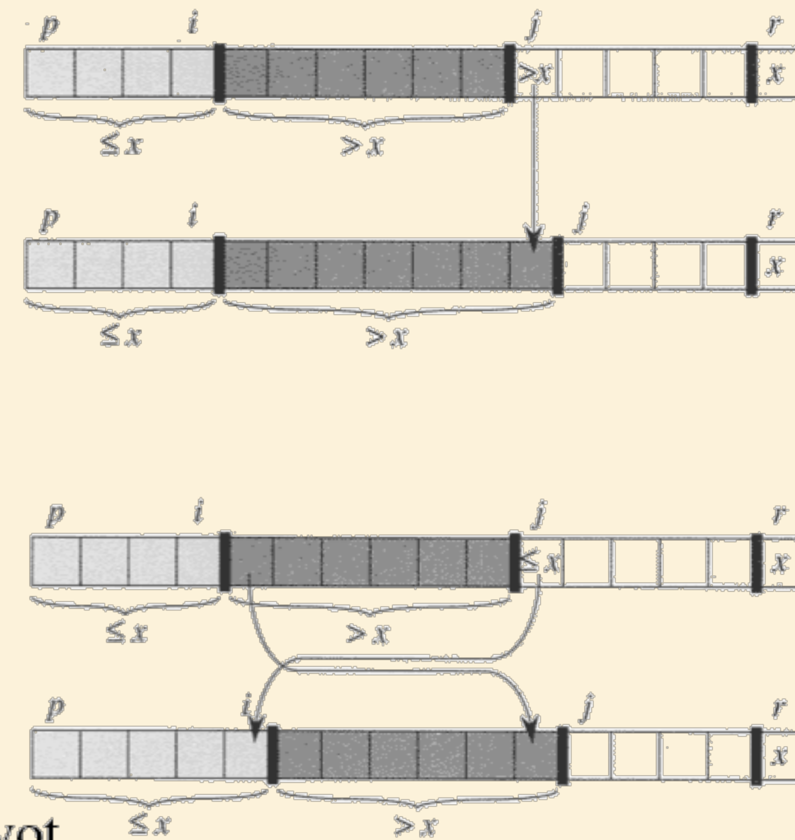
- **Note**: Use the lecture slides here instead of the textbook implementation (Section 11.2.2)

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



Maintaining Loop Invariant

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
        do if A[j] ≤ x
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```



Loop invariant:

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

The In-Place Quick-Sort Algorithm

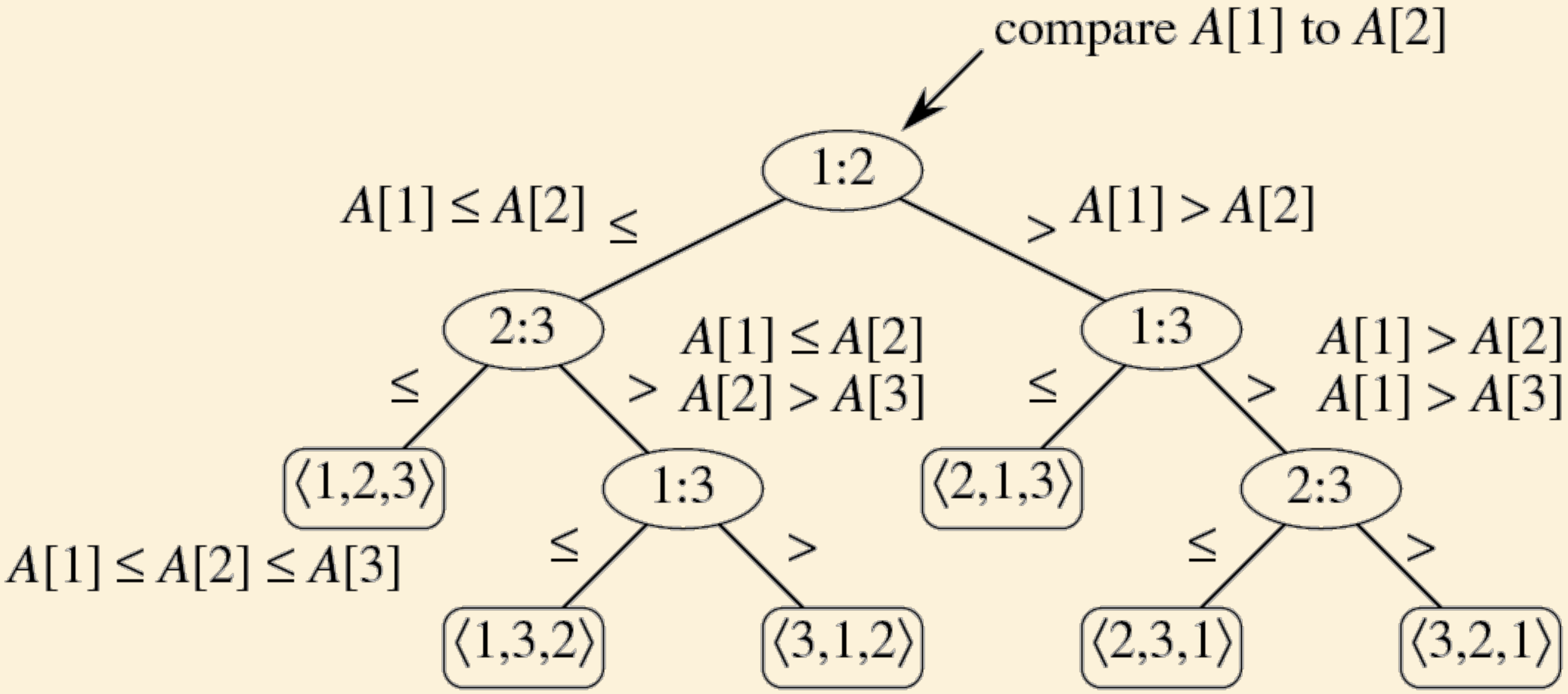
```
Algorithm QuickSort(A, p, r)
    if p < r
        q = Partition(A, p, r)
        QuickSort(A, p, q - 1)
        QuickSort(A, q + 1, r)
```

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 $T(n) \in \Omega(\log n!)$

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

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

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:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:						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
Location of next record with digit v.	0	5	14	17

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
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
Location of next record with digit v.	0	6	14	17

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

RadixSort

344	333	2 24
125	143	1 25
333	243	2 25
134	344	3 25
224	134	3 33
334	224	1 34
143	334	3 34
225	125	1 43
325	225	2 43
243	325	3 44

Sort wrt which digit first?

The least significant.

Sort wrt which digit Second?

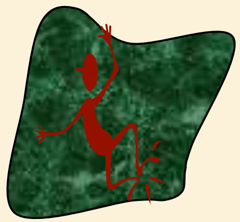
The next least significant.

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+1st 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 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+1st 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.

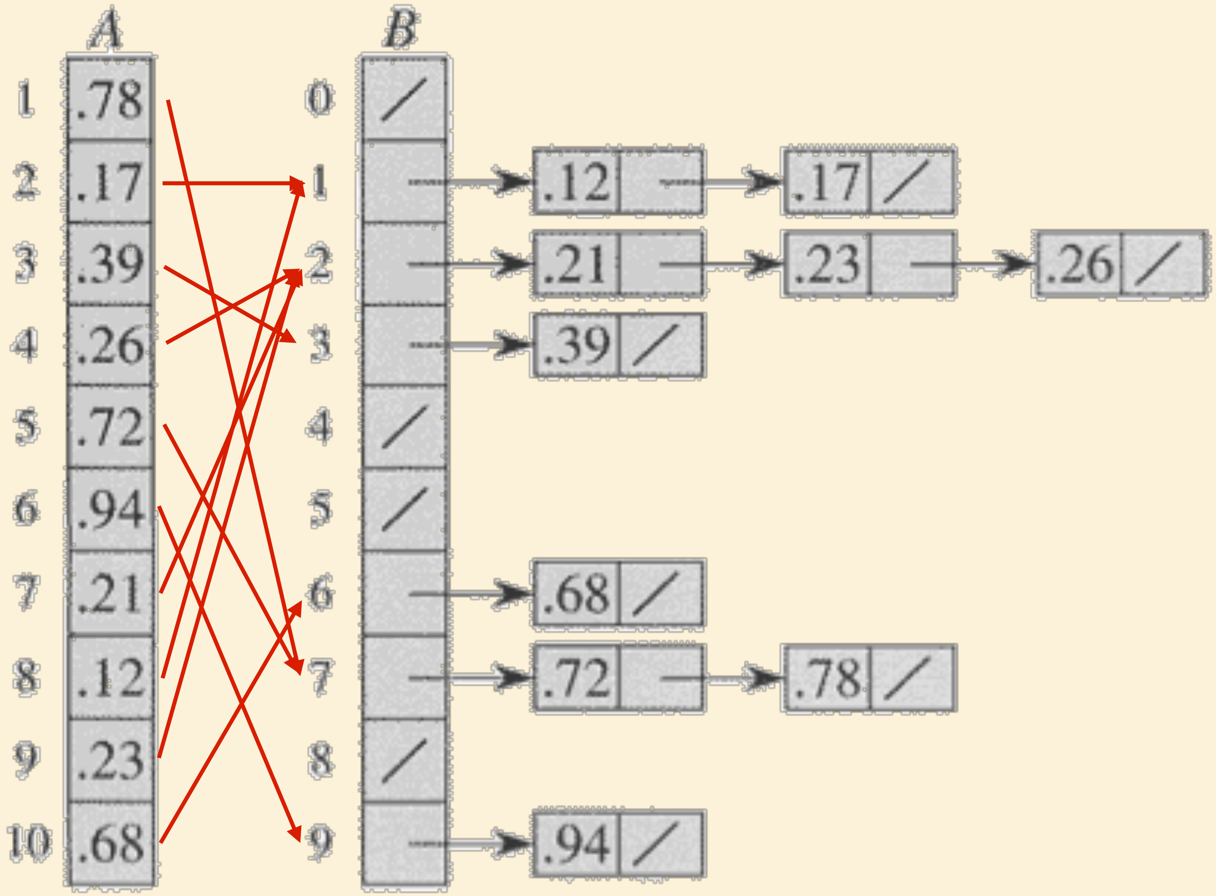
These are in the correct order because was sorted & stable sort left sorted

Example 3. Bucket Sort

- Applicable if input is constrained to finite interval, e.g., [0…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

Property 2

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

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

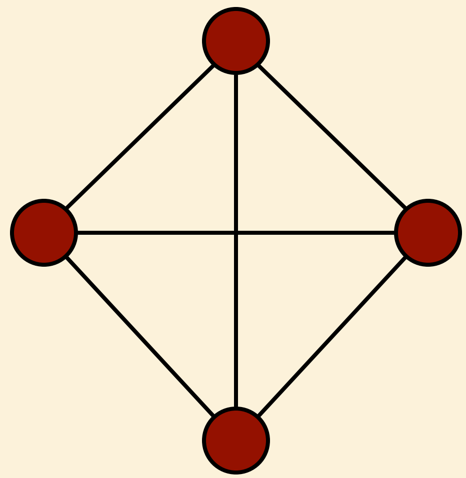
Proof: each vertex has degree at most $(|V| - 1)$

Q: What is the bound for a digraph?

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

Notation

$|V|$	number of vertices
$|E|$	number of edges
$\deg(v)$	degree of vertex v



Example

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

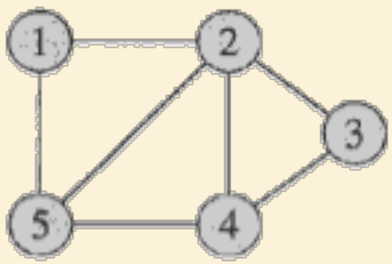
Main Methods of the (Undirected) Graph ADT

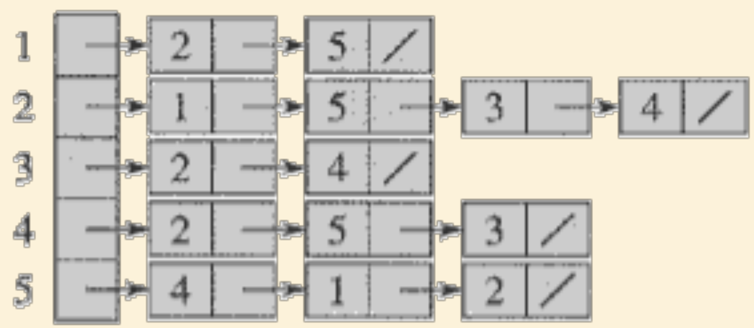
- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - endVertices(e): an array of the two endvertices of e
  - opposite(v, e): the vertex opposite to v on e
  - areAdjacent(v, w): true iff v and w are adjacent
  - replace(v, x): replace element at vertex v with x
  - replace(e, 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 | . | in asymptotic notation, e.g. *O(V+E).*

Implementing a Graph (Simplified)

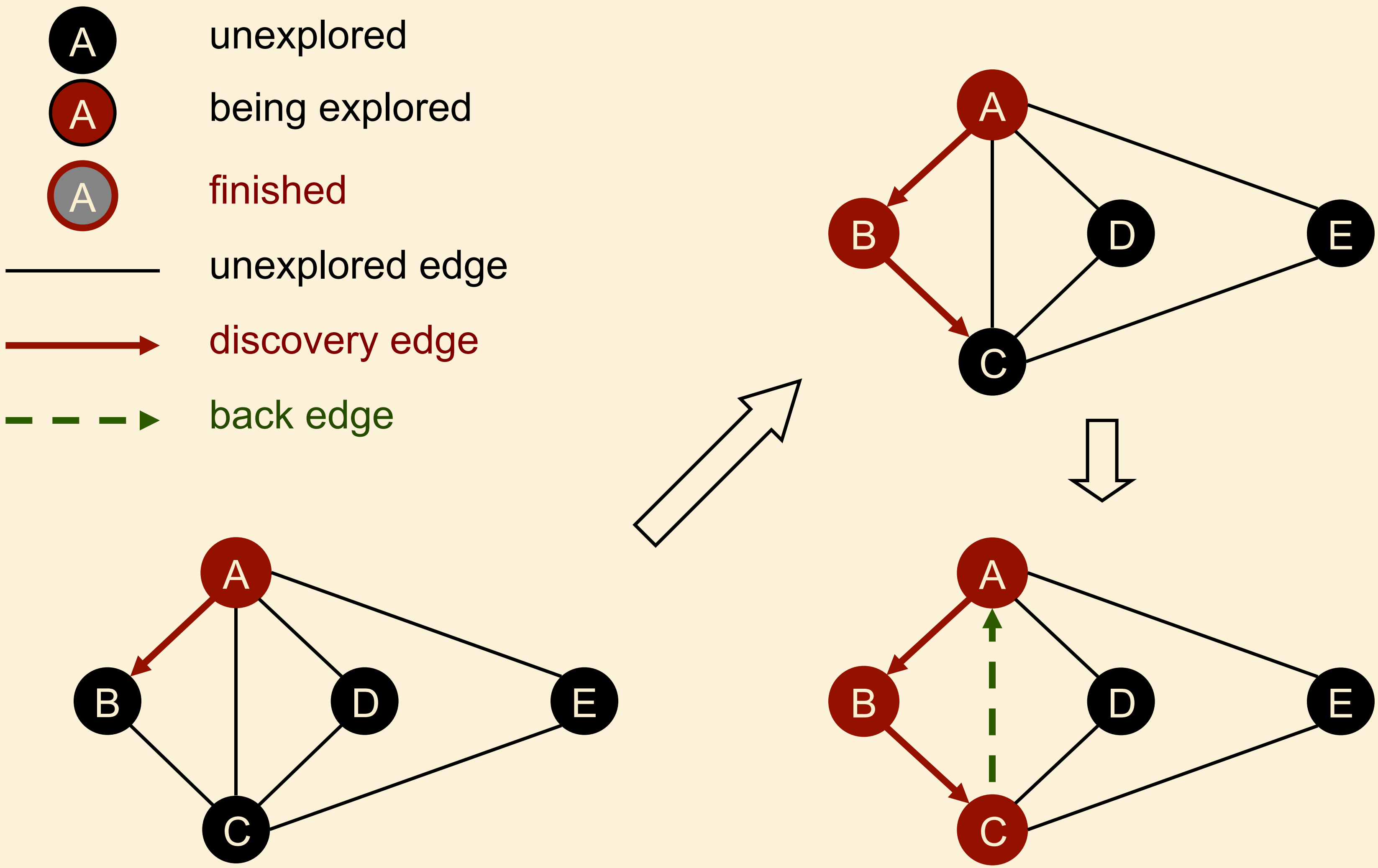




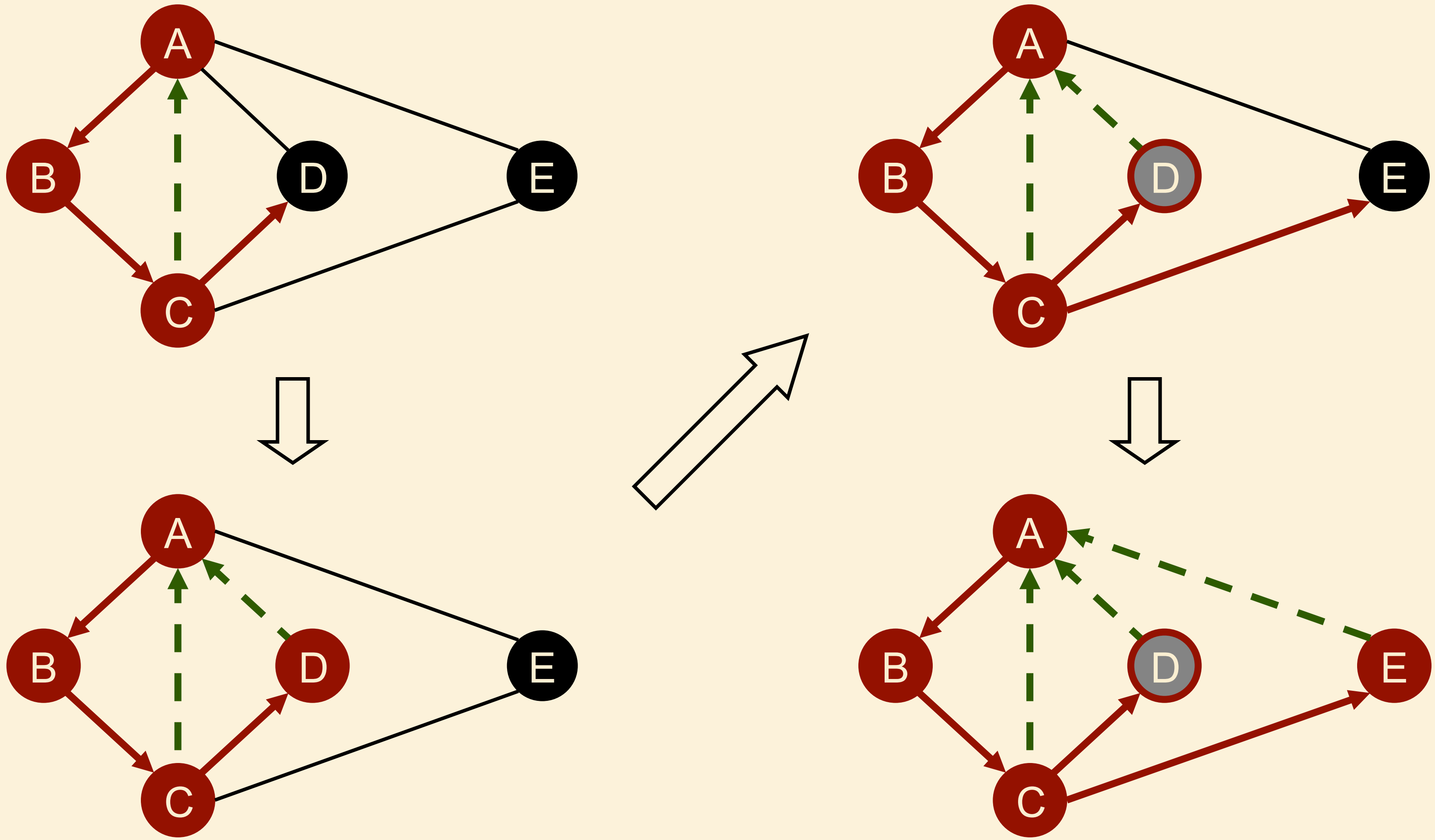
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	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

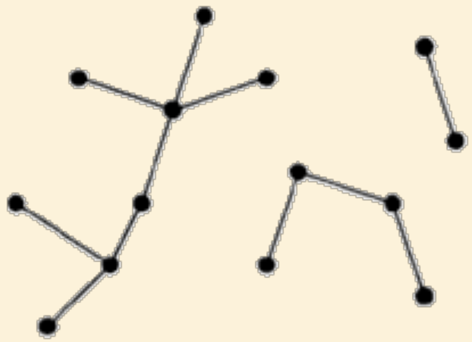
DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

```
for each vertex u ∈ V[G]
        color[u] = BLACK //initialize vertex
for each vertex u ∈ V[G]
        if color[u] = BLACK //as yet unexplored
                DFS-Visit(u)
```

total work $= \theta(V)$



DFS Algorithm Pattern

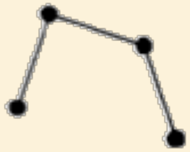
DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
colour[u] ← RED
for each v ∈ Adj[u] //explore edge (u,v)
    if color[v] = BLACK
        DFS-Visit(v)
colour[u] ← GRAY
```

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



Thus running time = $\theta(V+E)$

(assuming adjacency list structure)

Other Variants of Depth-First Search

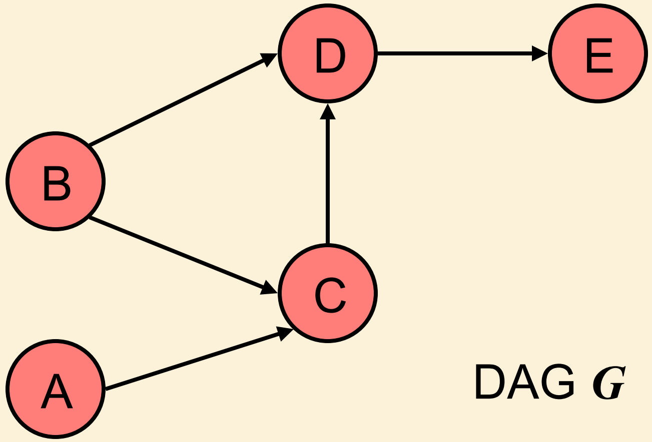
- The DFS Pattern can also be used to
  - Compute a forest of spanning trees (one for each call to DFS-visit) encoded in a predecessor list π[u]
  - Label edges in the graph according to their role in the search (see textbook)
    - **Tree edges**, traversed to an undiscovered vertex
    - **Forward 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

  $$v_1, \ldots, v_n$$

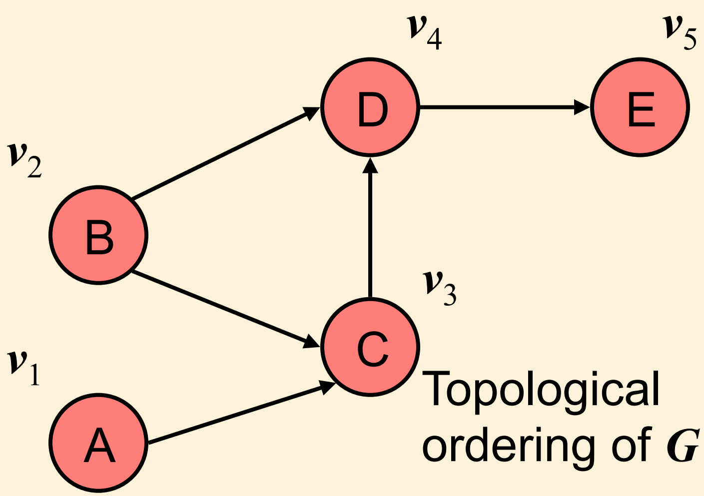
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints



DAG G

Theorem

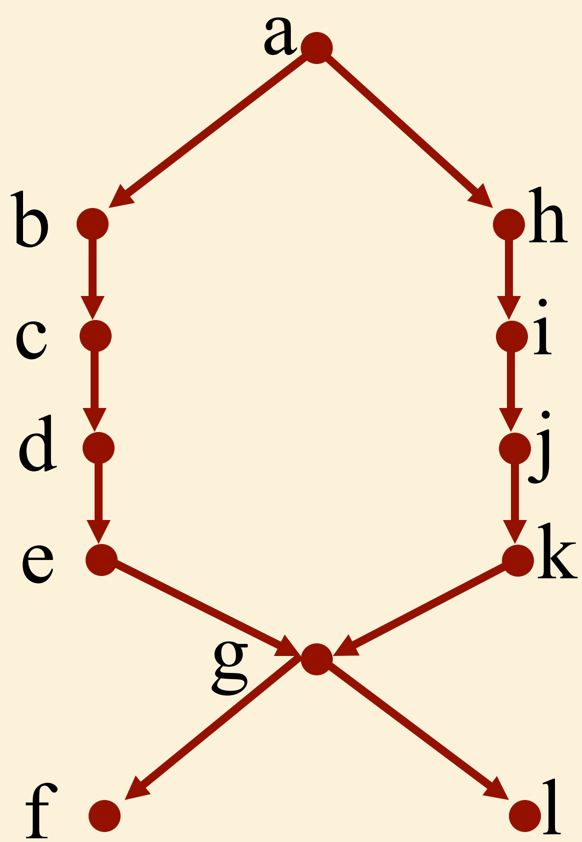
A digraph admits a topological ordering if and only if it is a DAG



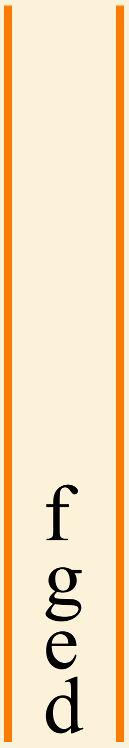
Topological ordering of G

Linear Order

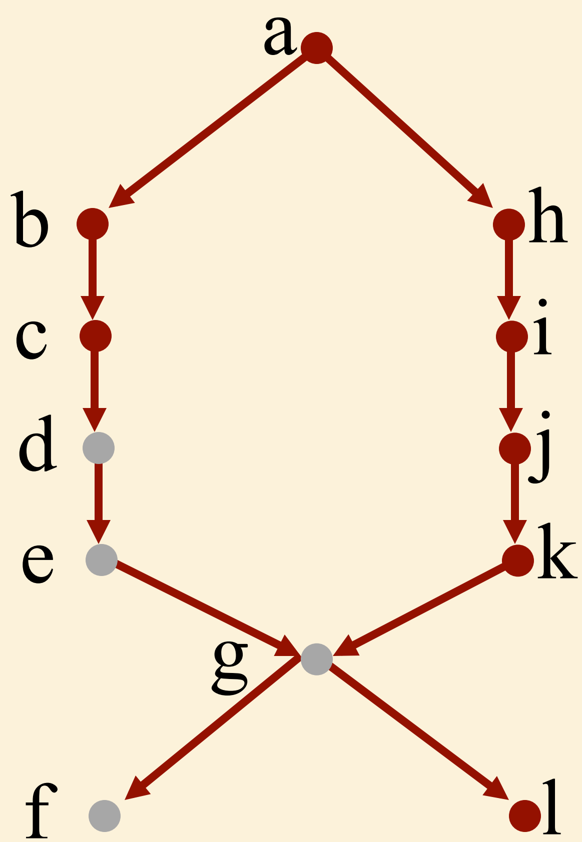
Alg: DFS

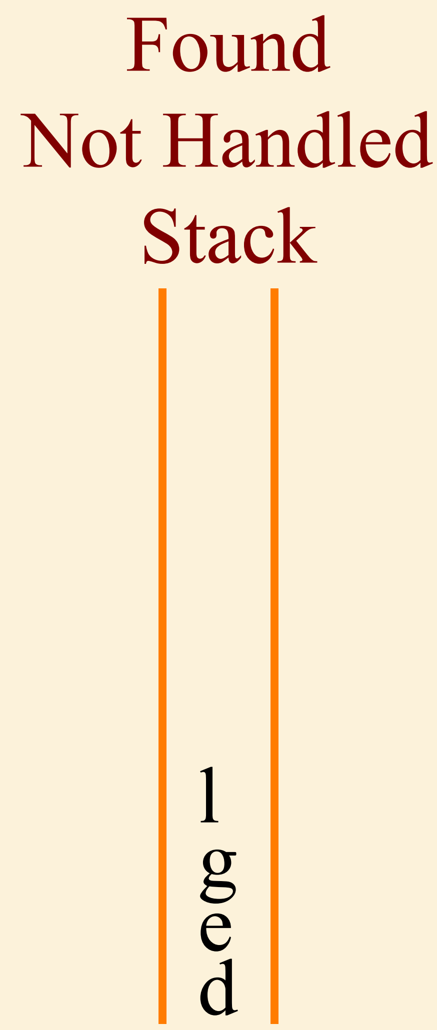


Found
Not Handled
Stack



Linear Order
Alg: DFS



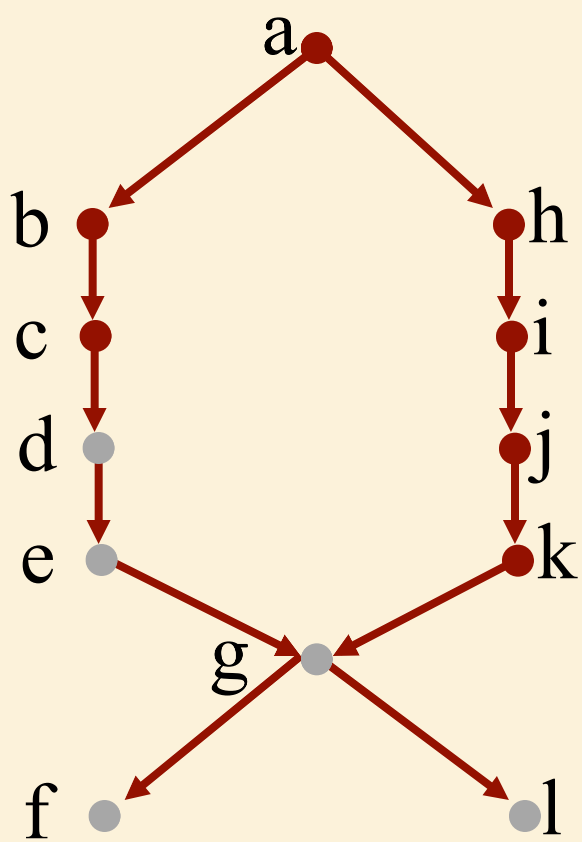


When node is popped off stack, insert at front of linearly-ordered “to do” list.

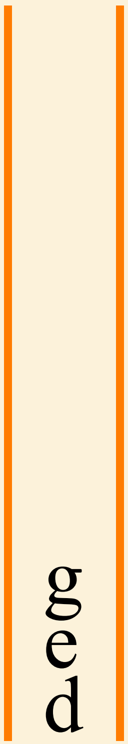
Linear Order:

….. f

Linear Order
Alg: DFS



Found
Not Handled
Stack

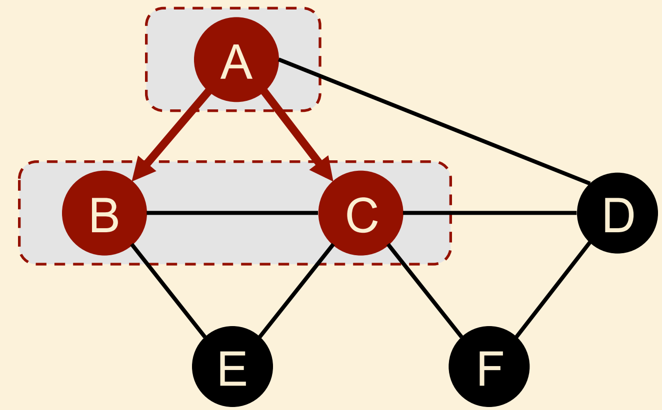


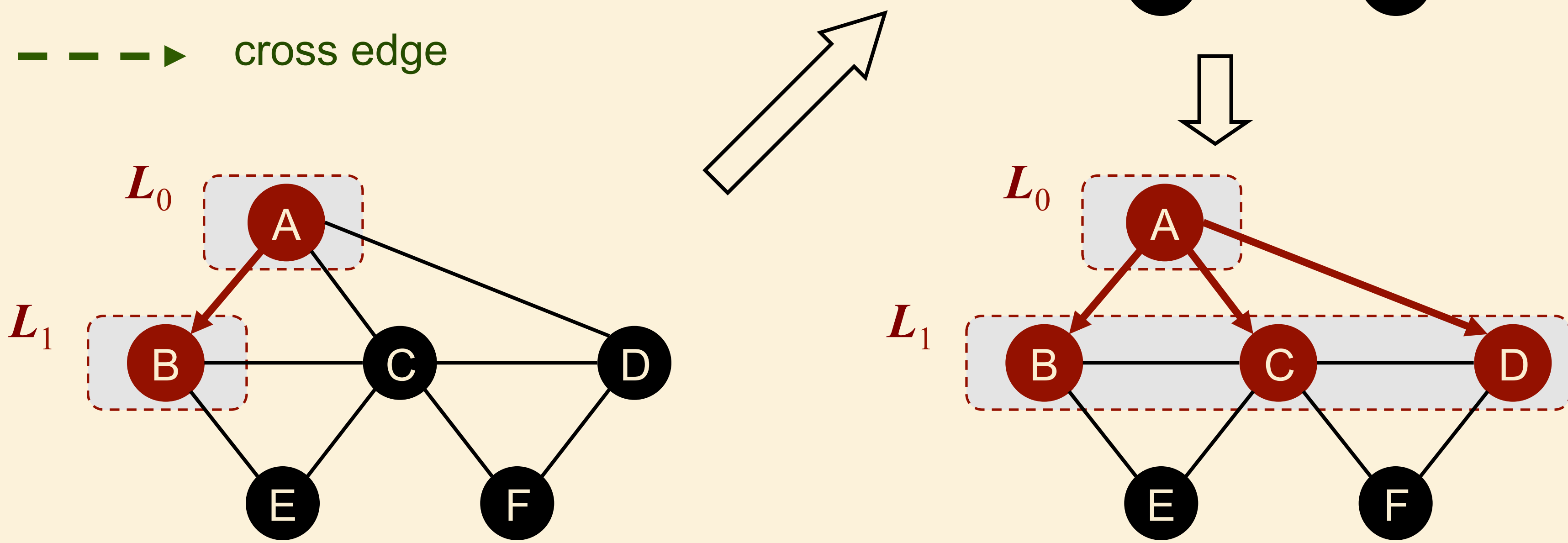
Linear Order:

l,f

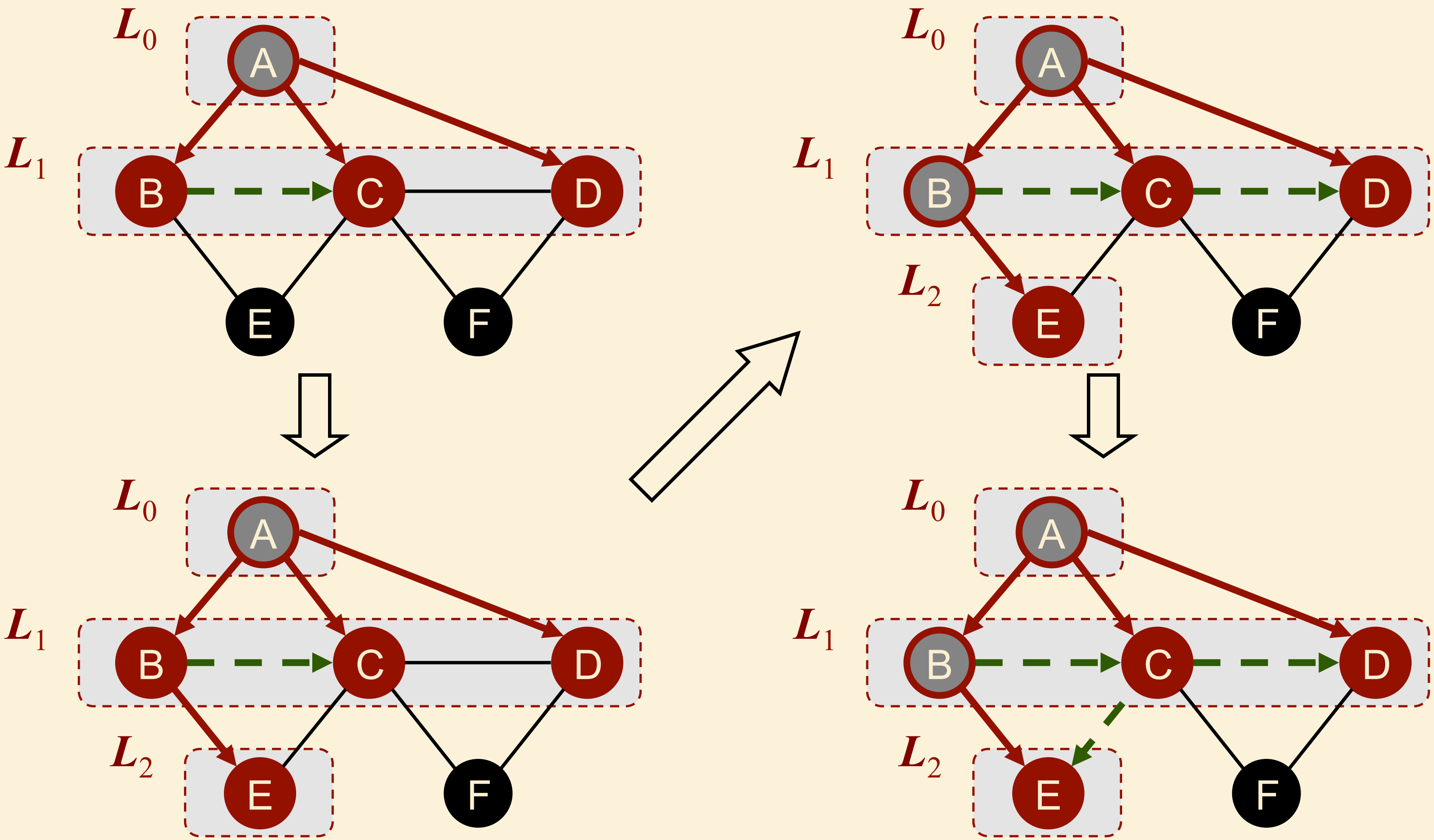
BFS Example

- A — undiscovered
- A — discovered (on Queue)
- A — finished
- unexplored edge
- discovery edge
- cross edge

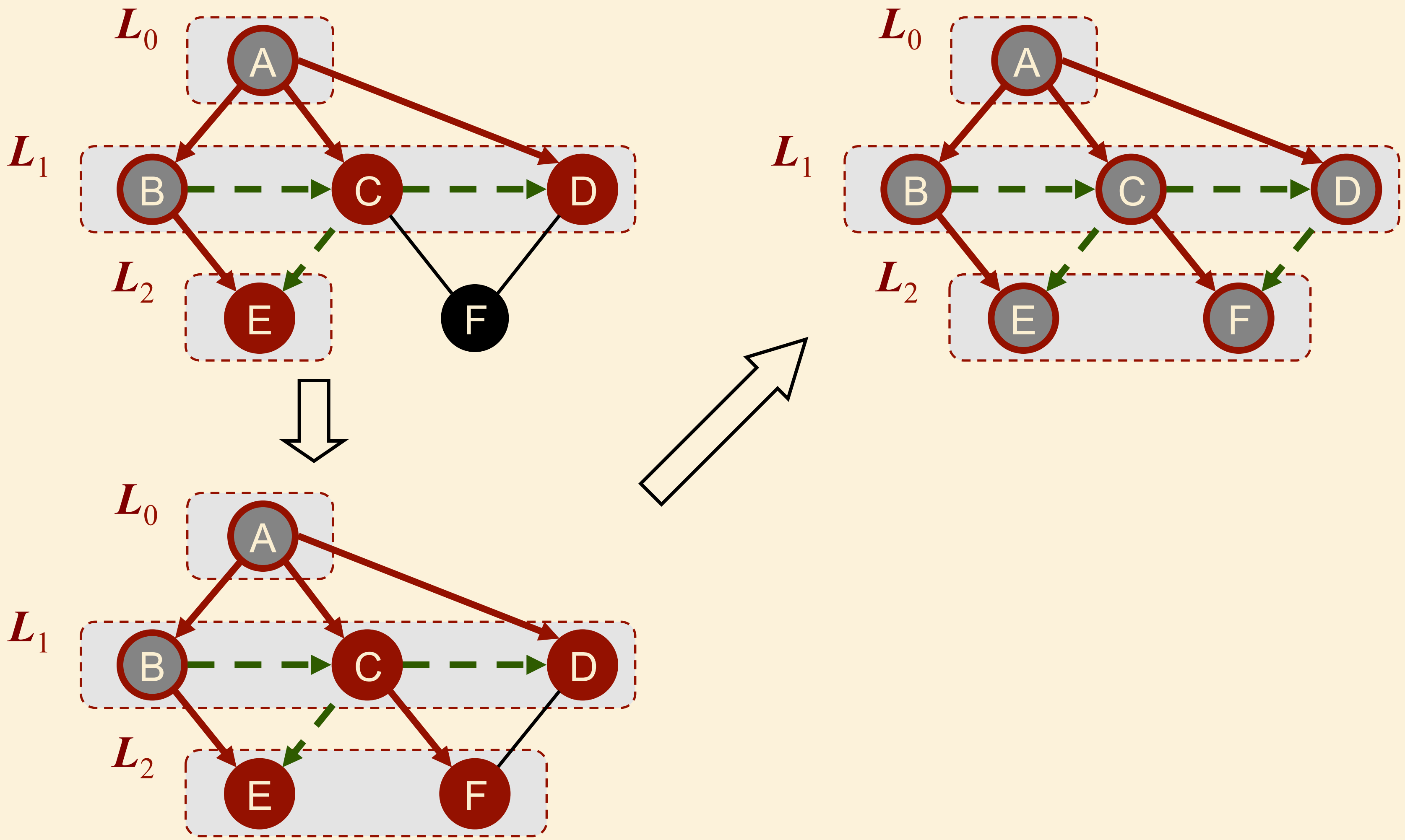




BFS Example (cont.)



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 δ[u] and
π[u] = predecessor of u on shortest paths from s to each vertex u in G
    for each vertex u ∈ V[G]
        d[u] ← ∞
        π[u] ← null
        color[u] = BLACK //initialize vertex
    colour[s] ← RED
    d[s] ← 0
    Q.enqueue(s)
    while Q ≠ ∅
        u ← Q.dequeue()
        for each v ∈ Adj[u] //explore edge (u,v)
            if color[v] = BLACK
                colour[v] ← RED
                d[v] ← d[u] + 1
                π[v] ← u
                Q.enqueue(v)
        colour[u] ← GRAY
```