Final Exam

- Fri, 24 Apr 2015, 9:00 – 12:00 LAS C
- Closed Book
- Format similar to midterm
- Will cover whole course, with emphasis on material after midterm (maps and hash tables, binary search, loop invariants, binary search trees, sorting, graphs)
- We did not cover breadth-first search so you are **not** responsible for this material.
- I will be away at meetings from Wed Apr 15 – Thurs Apr 23: please see TAs for assistance.

Suggested Study Strategy

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

End of Term Review

Summary of Topics

1. Maps & Hash Tables
2. Binary Search & Loop Invariants
3. Binary Search Trees
4. Sorting
5. Graphs

Summary of Topics

1. **Maps & Hash Tables**
2. Binary Search & Loop Invariants
3. Binary Search Trees
4. Sorting
5. 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

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

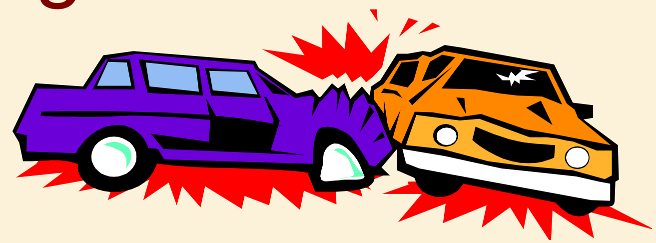
Hash Tables

- 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).

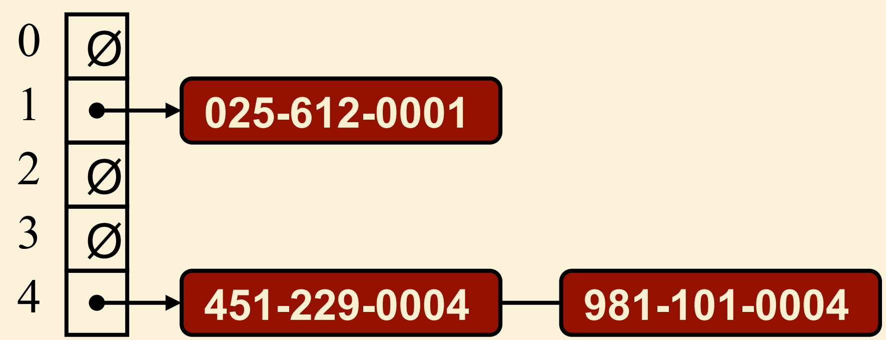
Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime (on the assumption that the differences between hash keys y are less likely to be multiples of primes).
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$, where
    - p is a prime number greater than N
    - a and b are integers chosen at random from the interval [0, p – 1], with a > 0.

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



Open Addressing: 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

Open Addressing: Double Hashing

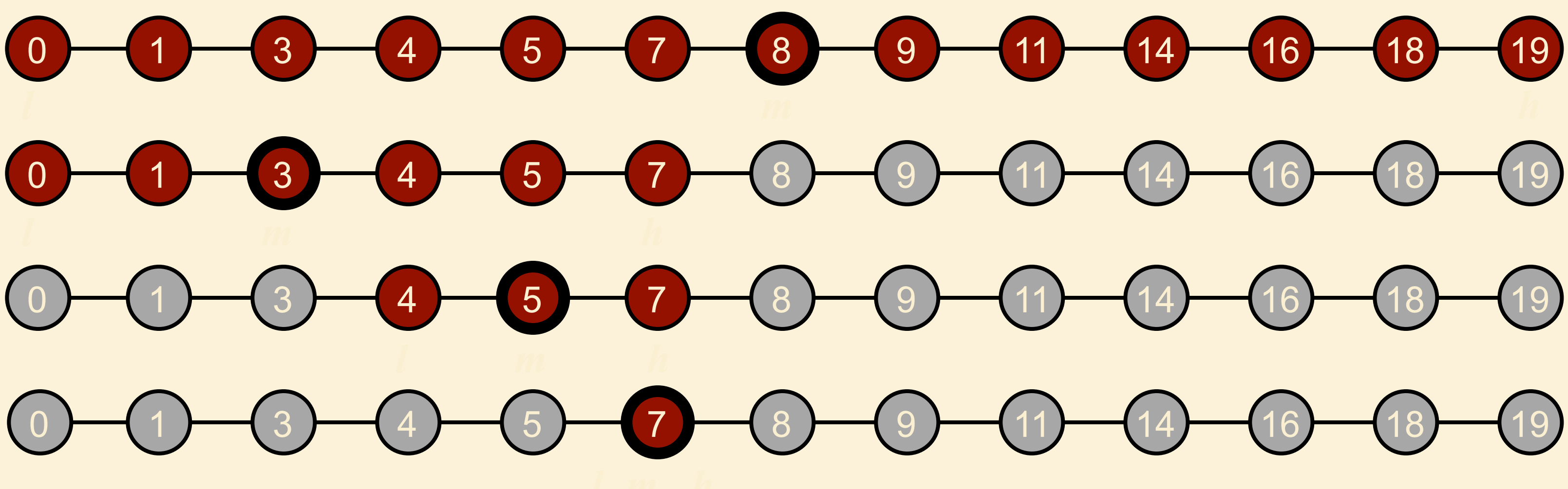
- Double hashing is an alternative open addressing method that uses a **secondary hash function h'(k)** in addition to the primary hash function h(x).
- Suppose that the primary hashing i=h(k) leads to a collision.
- We then iteratively probe the locations
  $(i + jh'(k)) \bmod N$ for $j = 0, 1, \ldots, N - 1$
- The secondary hash function ***h'***(***k***) cannot have zero values
- ***N*** is typically chosen to be prime.
- Common choice of secondary hash function h'(k):
  - $h'(k) = q - k \bmod q$, where
    - $q < N$
    - q is a prime
- The possible values for h'(k) are
  $1, 2, \ldots, \boldsymbol{q}$

Summary of Topics

1. Maps & Hash Tables
2. **Binary Search & Loop Invariants**
3. Binary Search Trees
4. Sorting
5. Graphs

Ordered Maps and Dictionaries

- If keys obey a total order relation, can represent a map or 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)



Loop Invariants

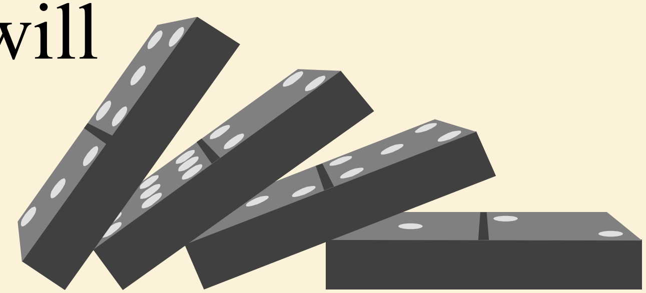
- Binary search can be implemented as an **iterative algorithm** (it could also be done recursively).
- **Loop Invariant:** An **assertion** about the current state useful for designing, analyzing and proving the correctness of iterative algorithms.

Establishing Loop Invariant

From the Pre-Conditions on the input instance we must establish the loop invariant.

Maintain Loop Invariant

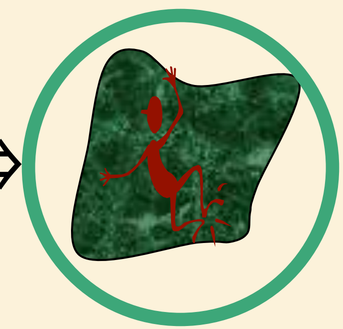
- By Induction the computation will always be in a safe location.



$\Rightarrow S(0)$

$\Rightarrow \forall i, S(i) \Rightarrow S(i+1)$

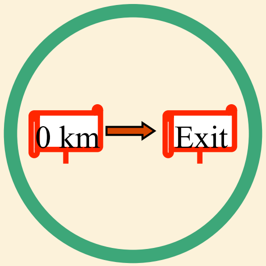
$\Rightarrow \forall i, S(i) \Rightarrow$



Ending The Algorithm

- Define Exit Condition

- Termination: With sufficient progress, the exit condition will be met.



- When we exit, we know
  - exit condition is true
  - loop invariant is true

  from these we must establish the post conditions.

Summary of Topics

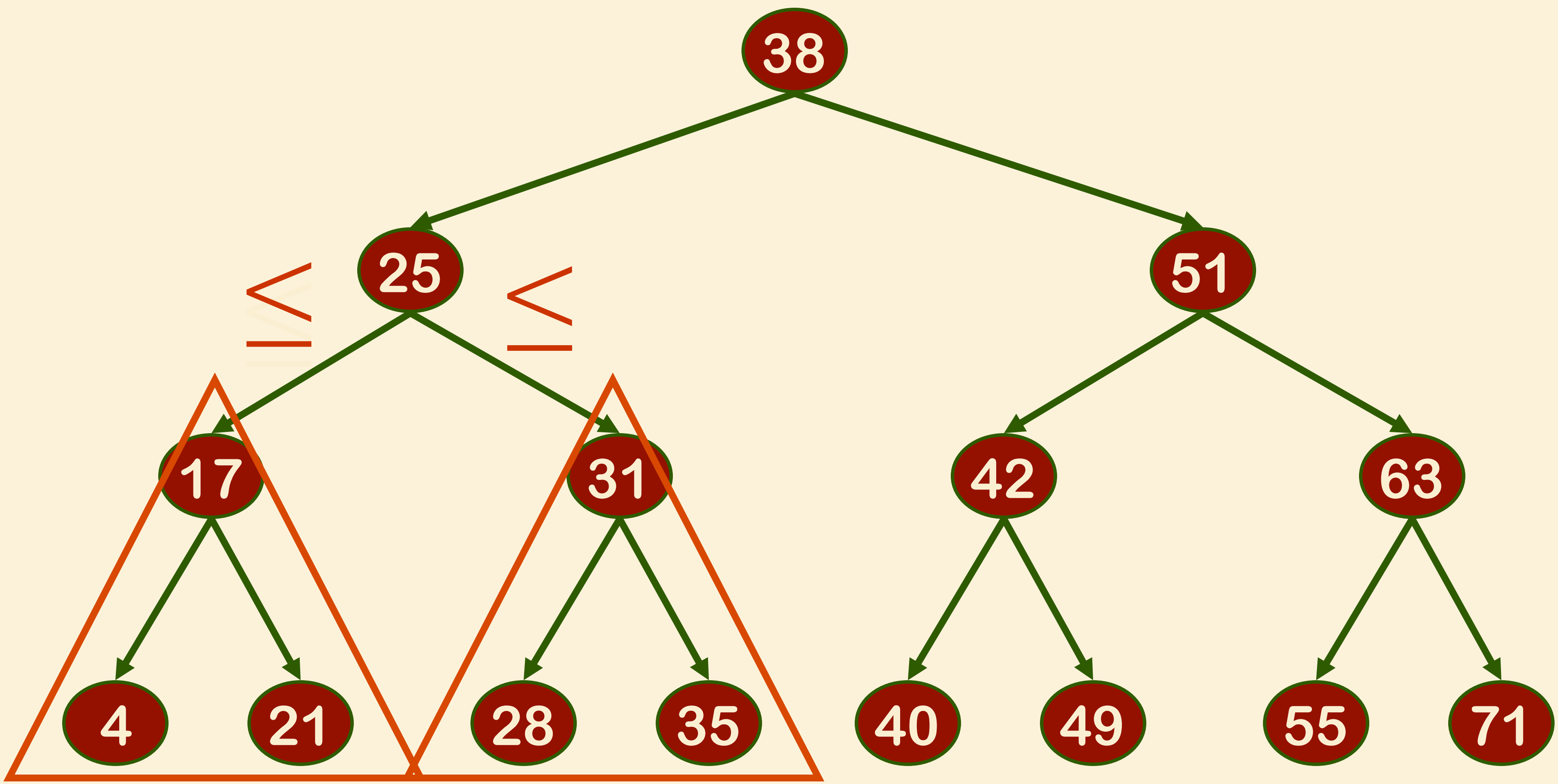
1. Maps & Hash Tables
2. Binary Search & Loop Invariants
3. **Binary Search Trees**
4. Sorting
5. Graphs

Binary Search Trees

- Insertion
- Deletion
- AVL Trees
- Splay Trees

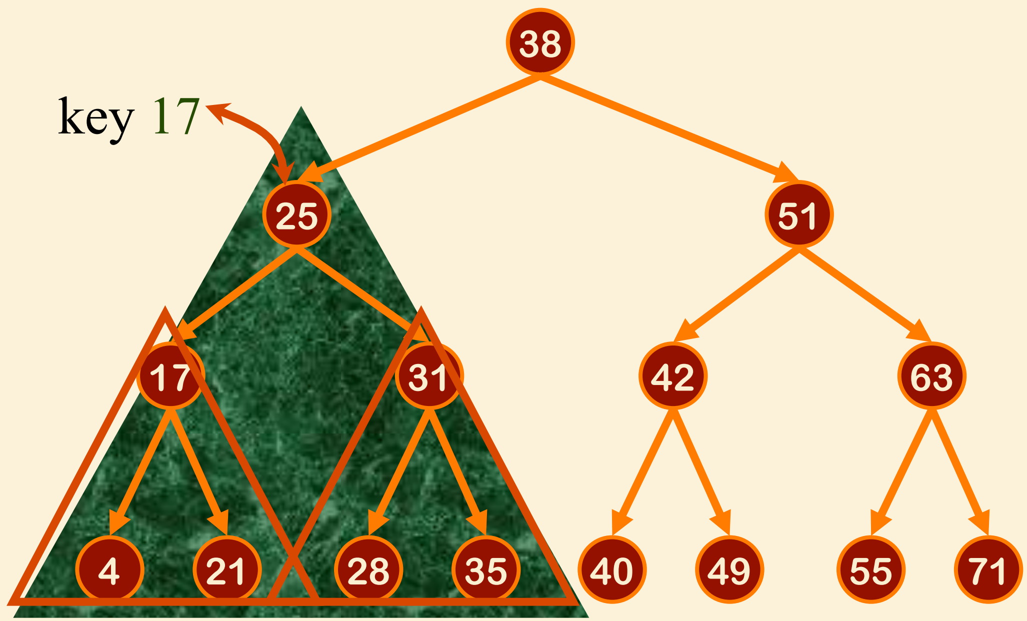
Binary Search Tree

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.



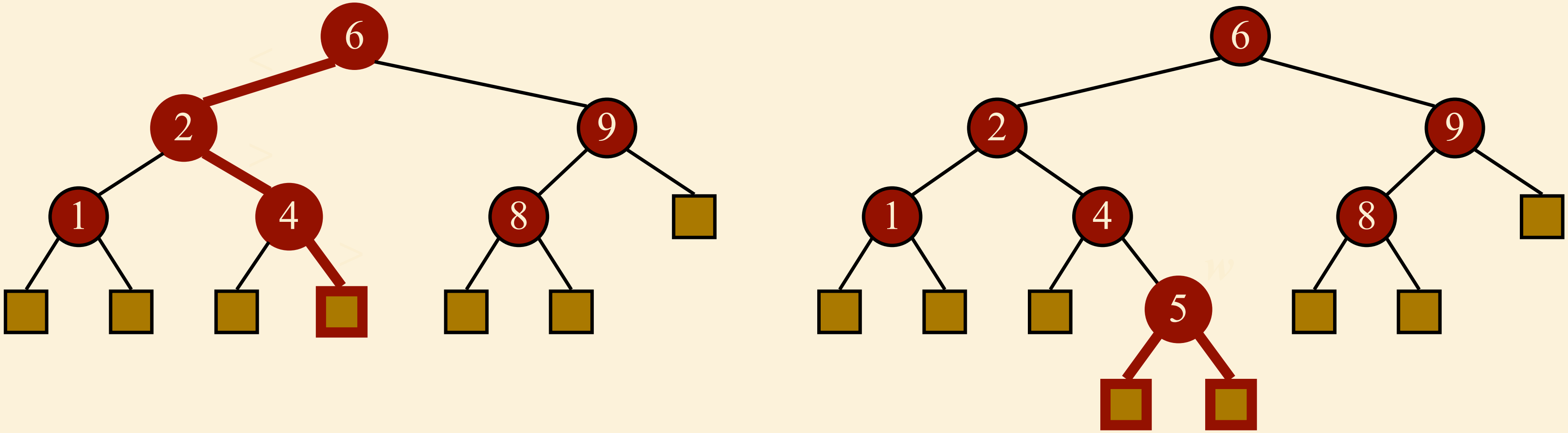
If key < root, then key is in left half.

If key = root, then key is found

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

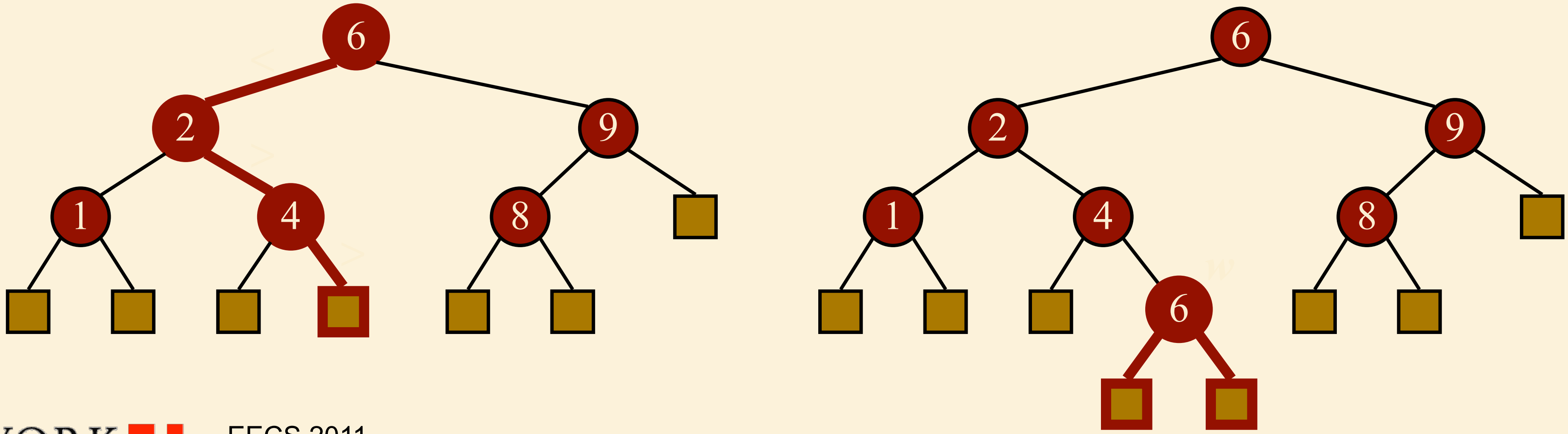
Insertion (For Dictionary)

- 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



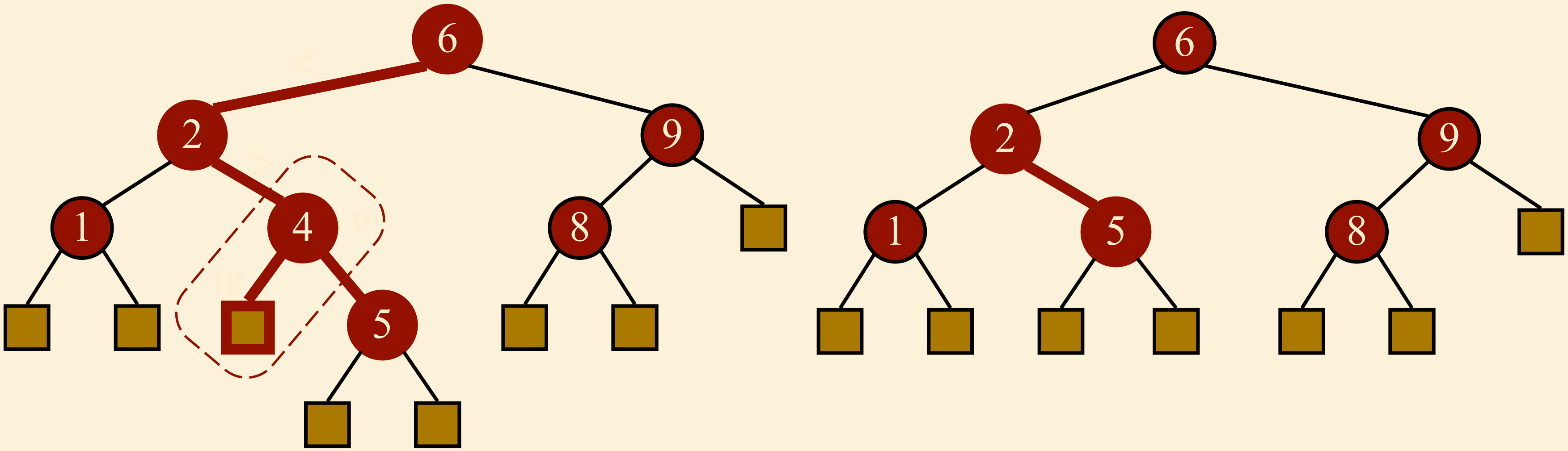
Insertion

- 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



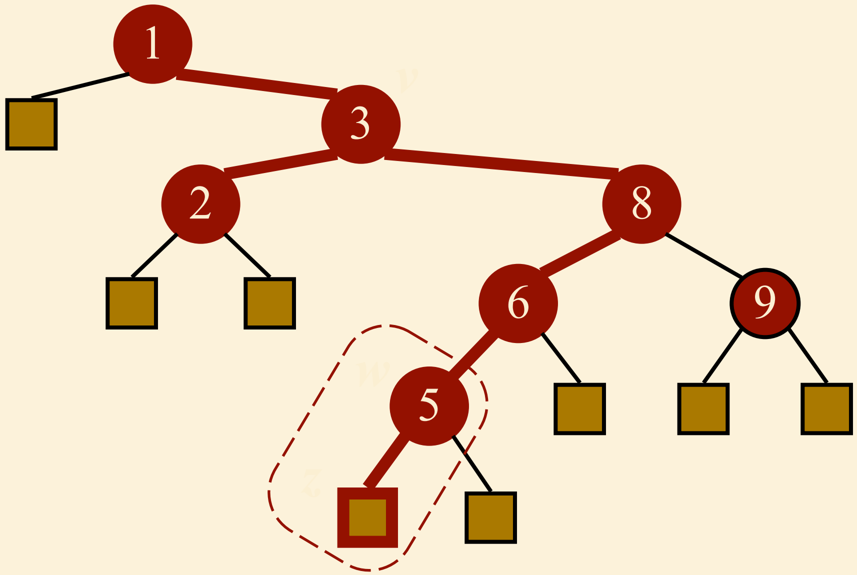
Deletion

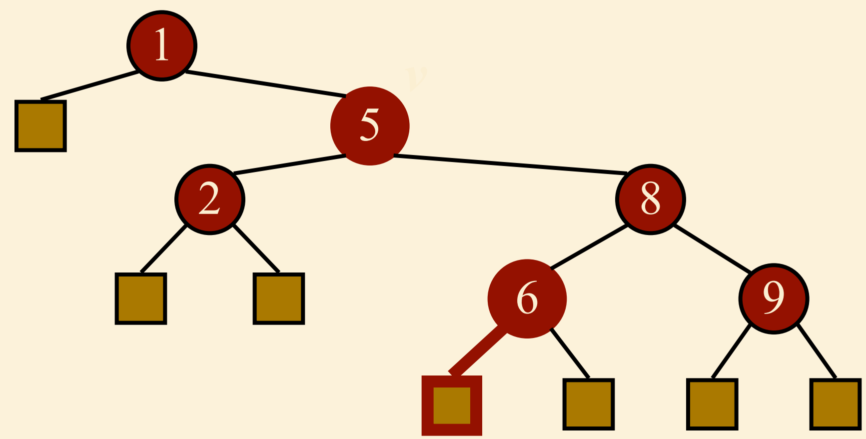
- To perform operation remove(k), we search for key k
- Suppose key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4



Deletion (cont.)

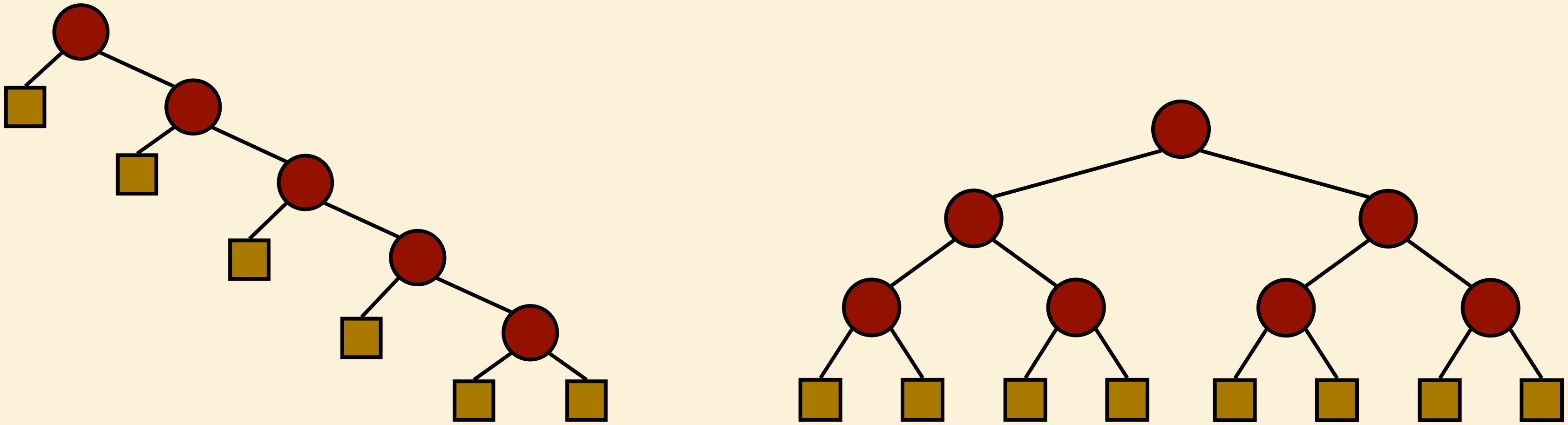
- Now consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy the entry stored at w into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3





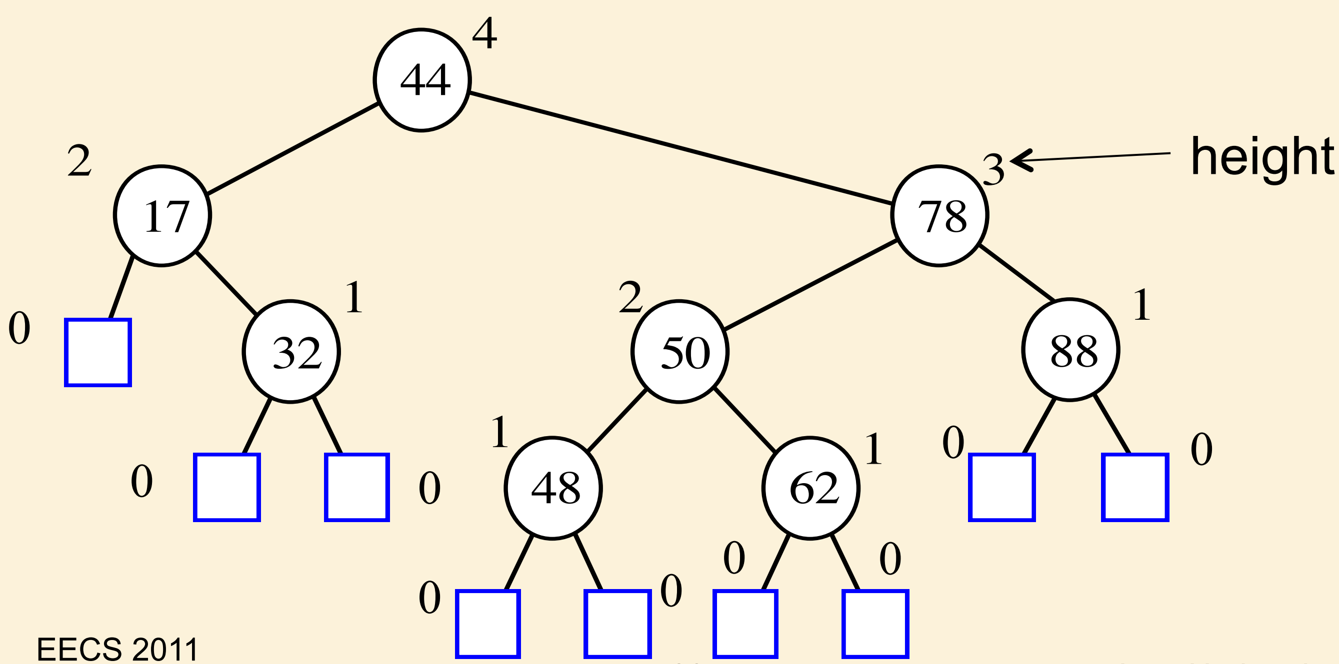
Performance

- 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)!



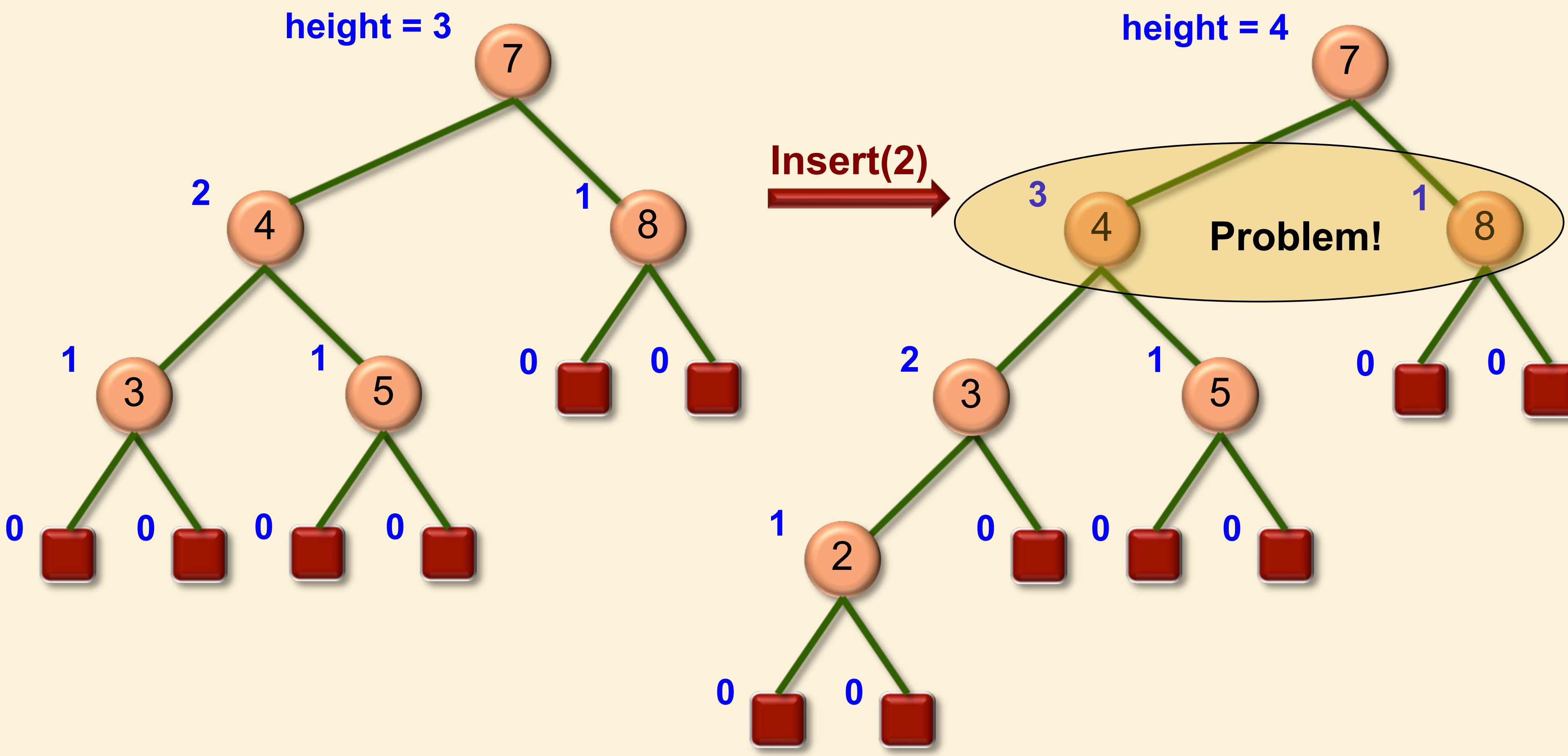
AVL Trees

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



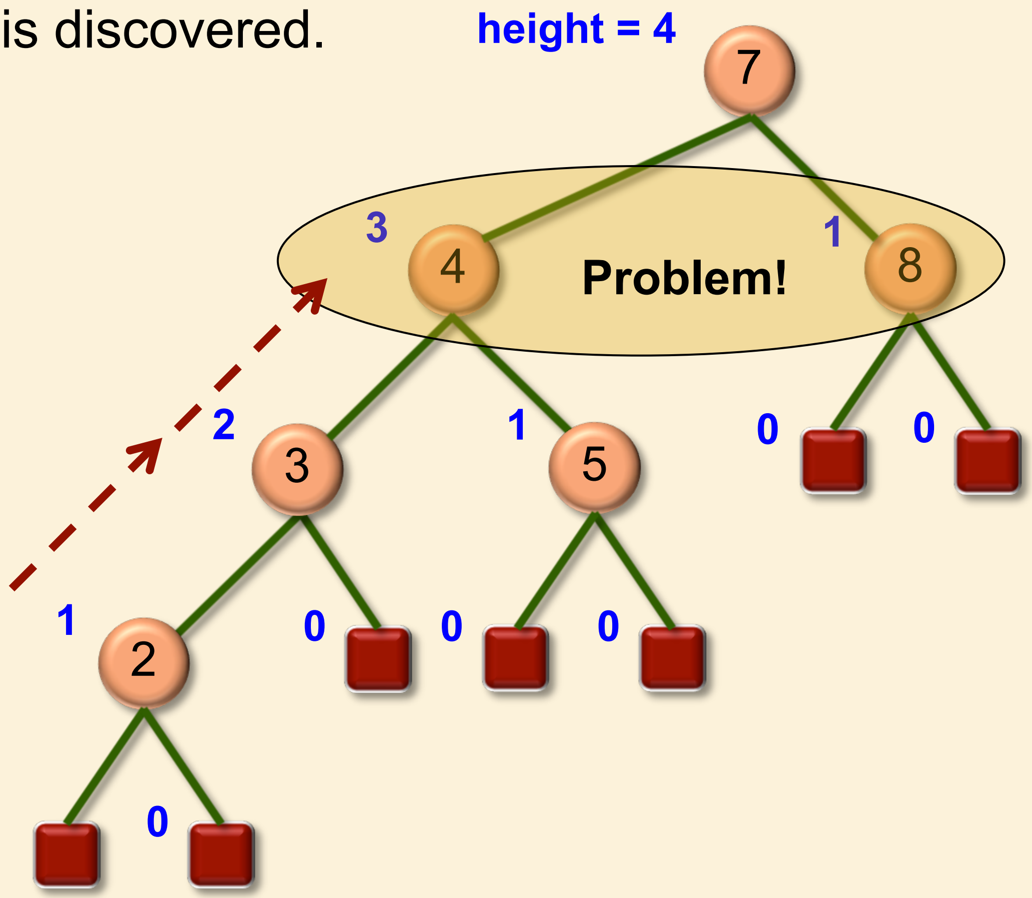
Insertion

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



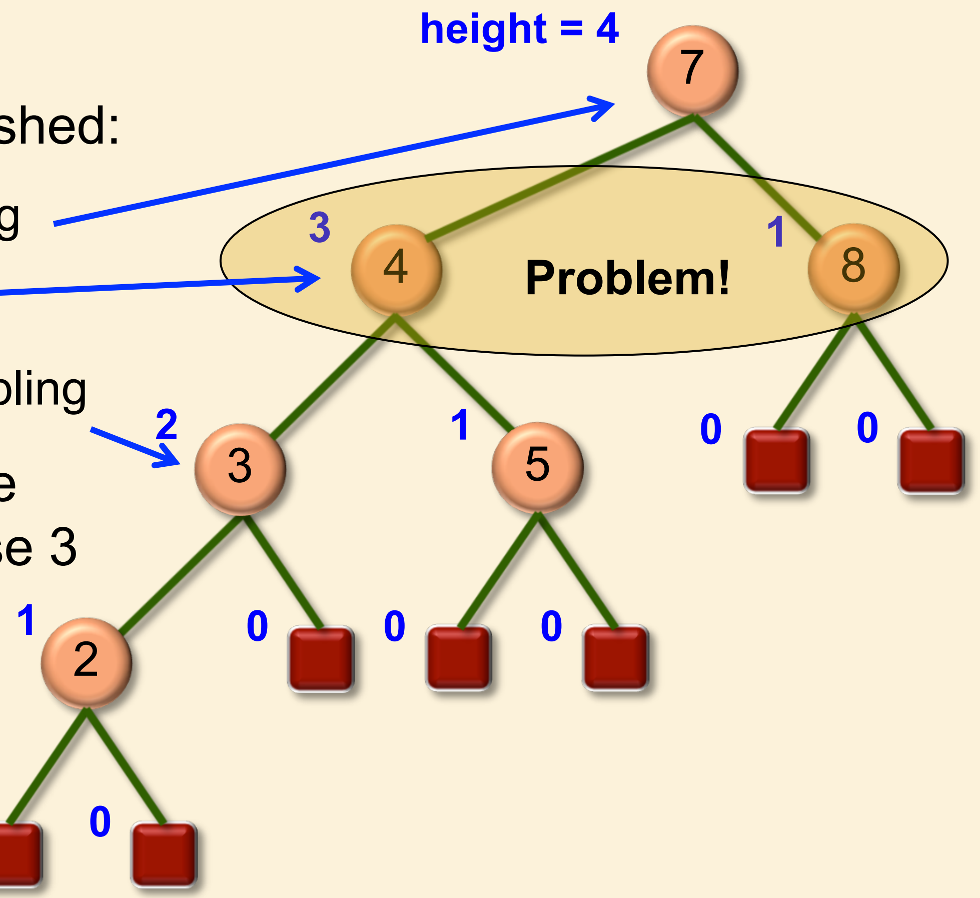
Insertion: Rebalancing Strategy

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



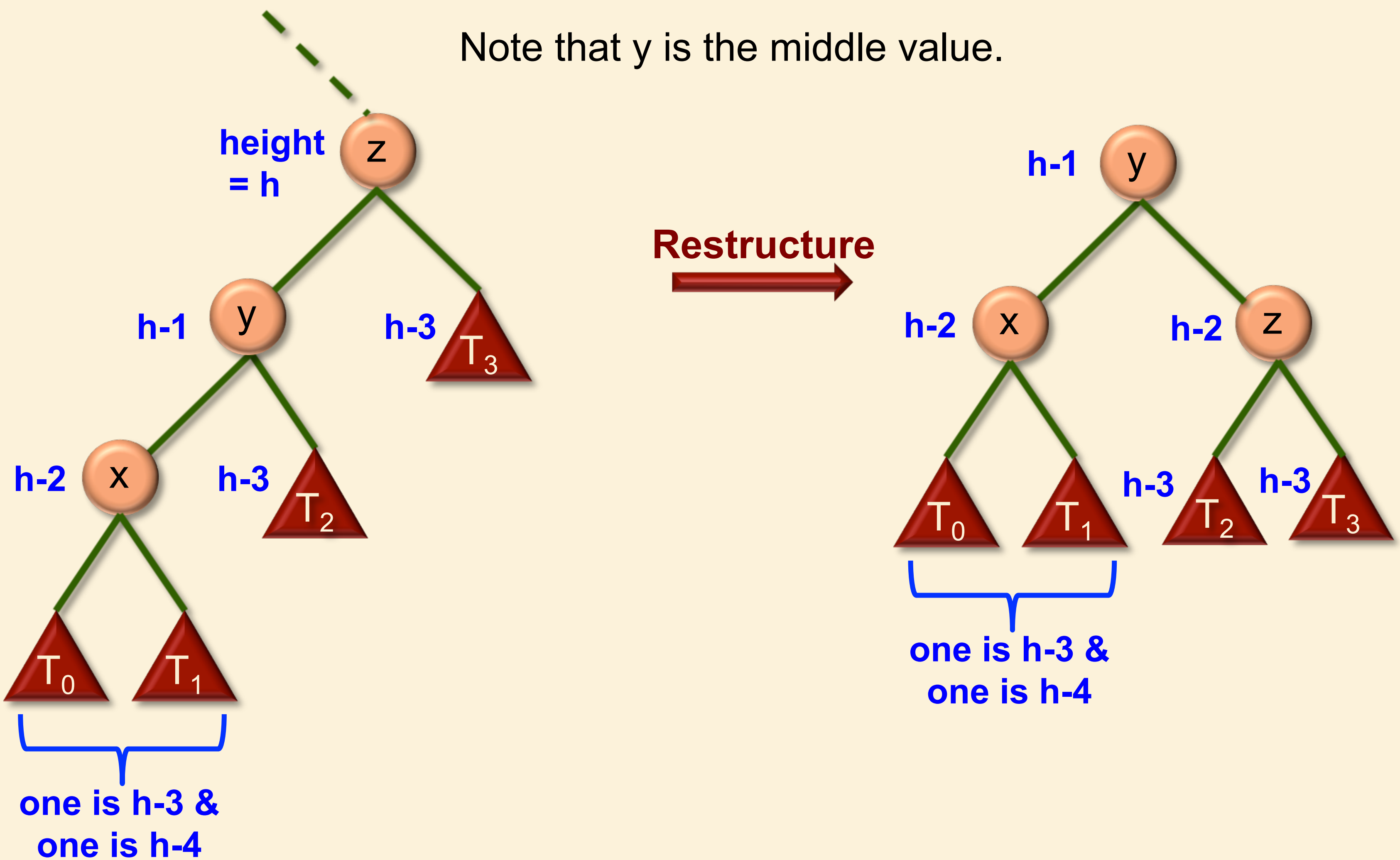
Insertion: Rebalancing Strategy

- 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



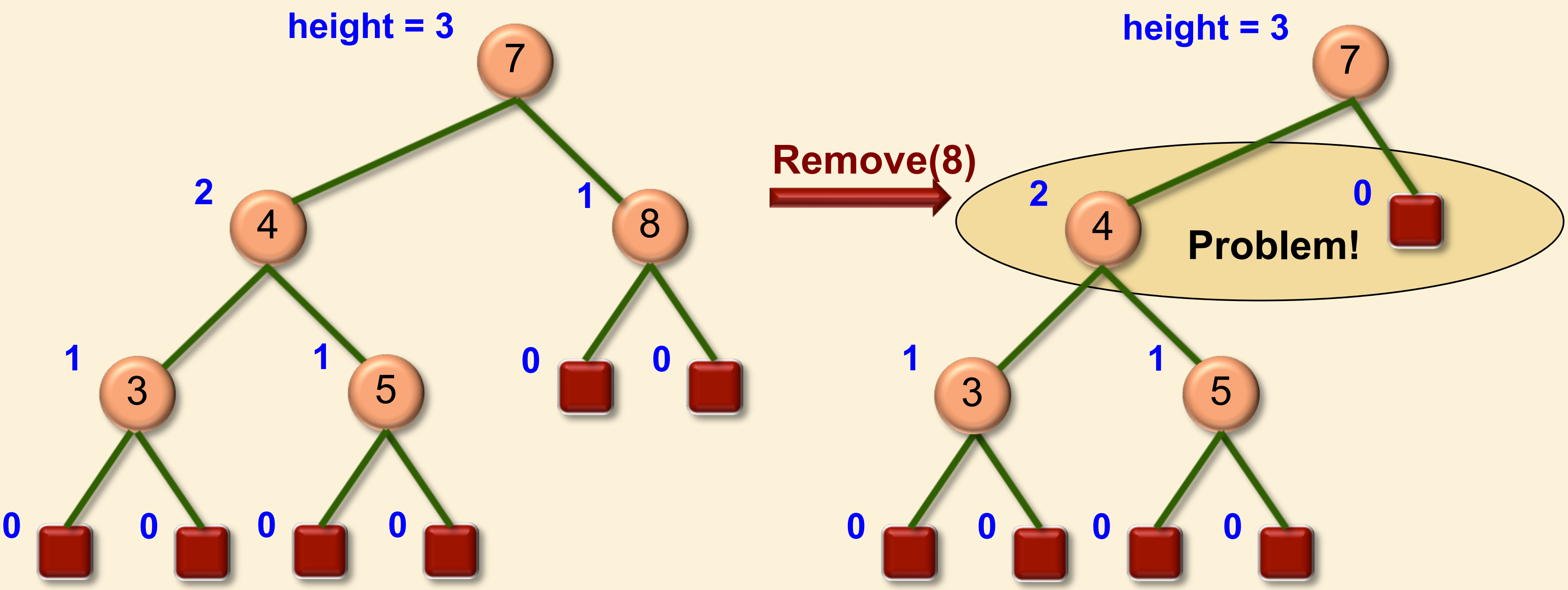
Insertion: Trinode Restructuring Example

Note that y is the middle value.



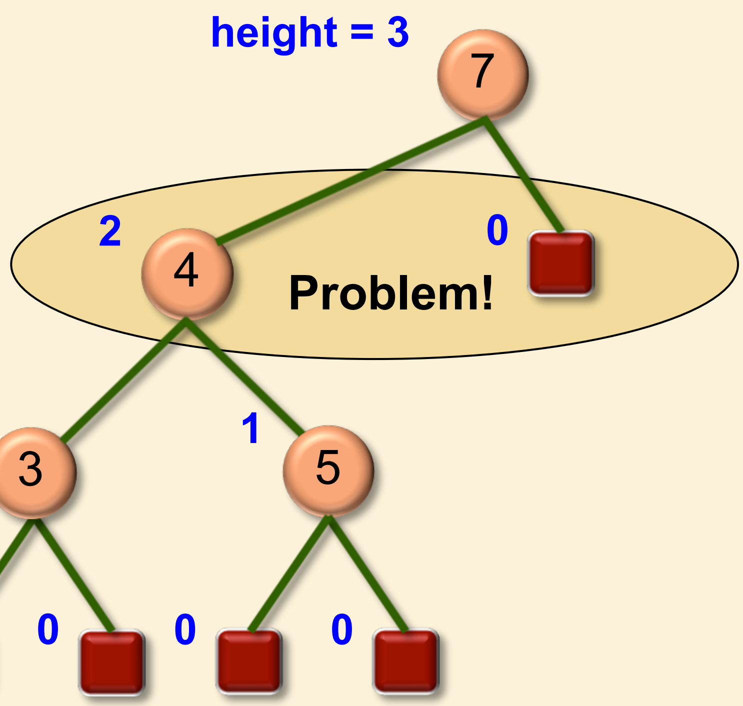
Removal

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



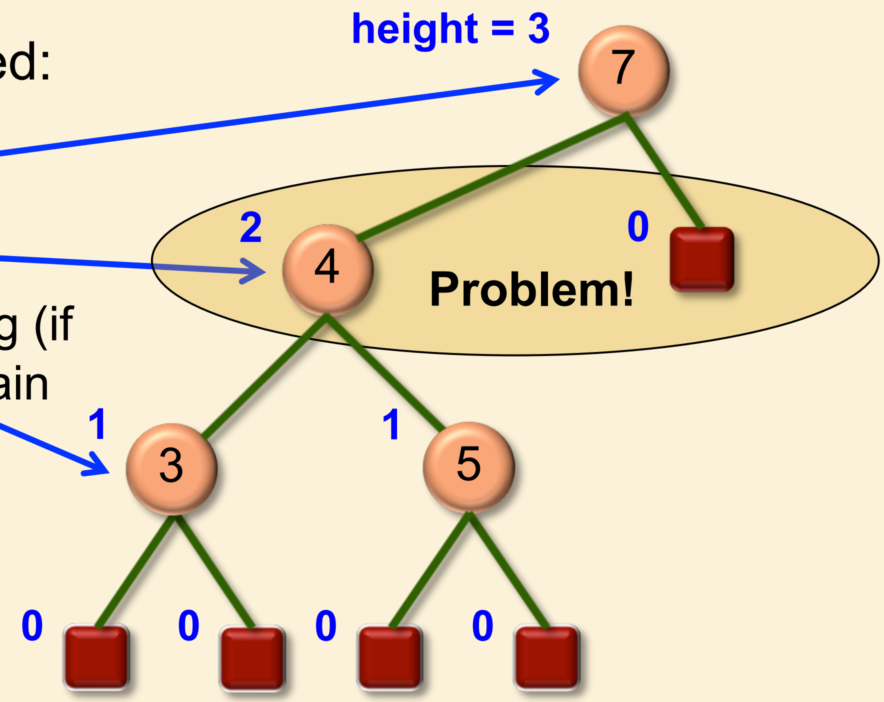
Removal: Rebalancing Strategy

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



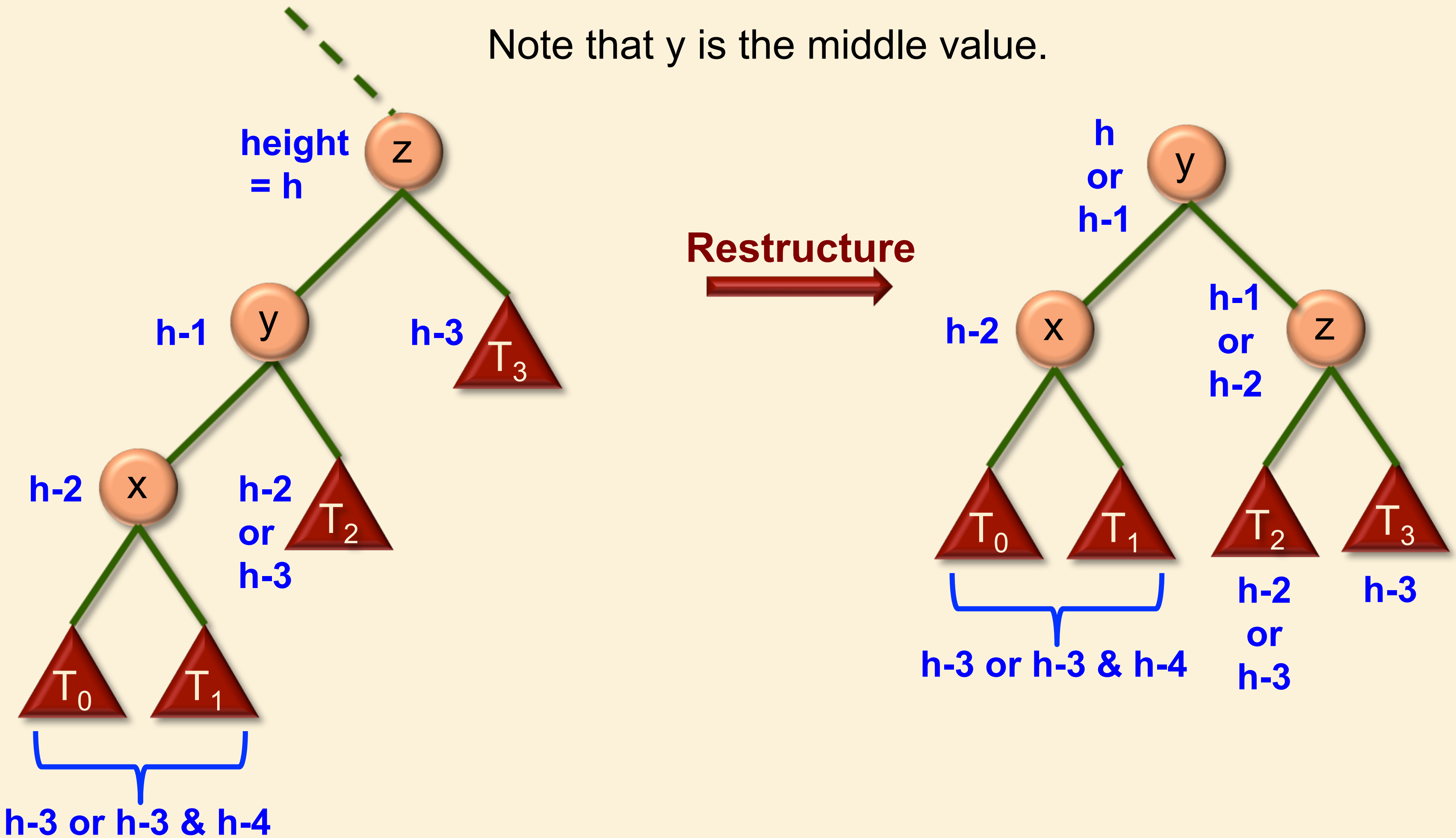
Removal: Rebalancing Strategy

- 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: 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

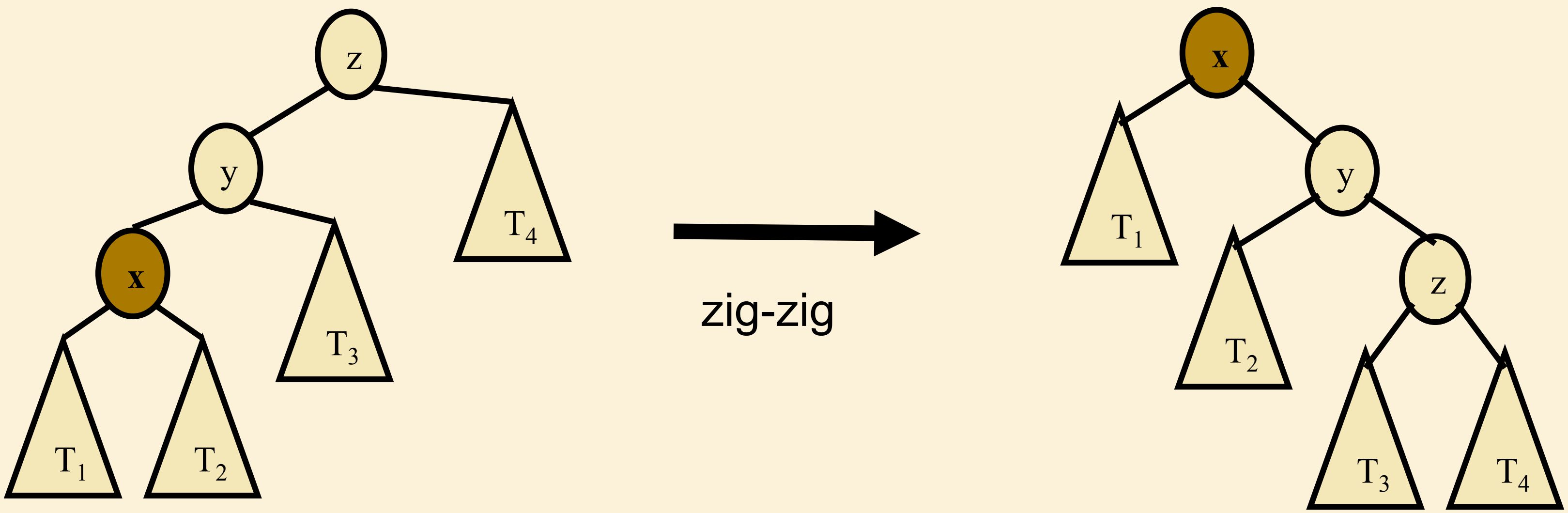
R. Tarjan

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.

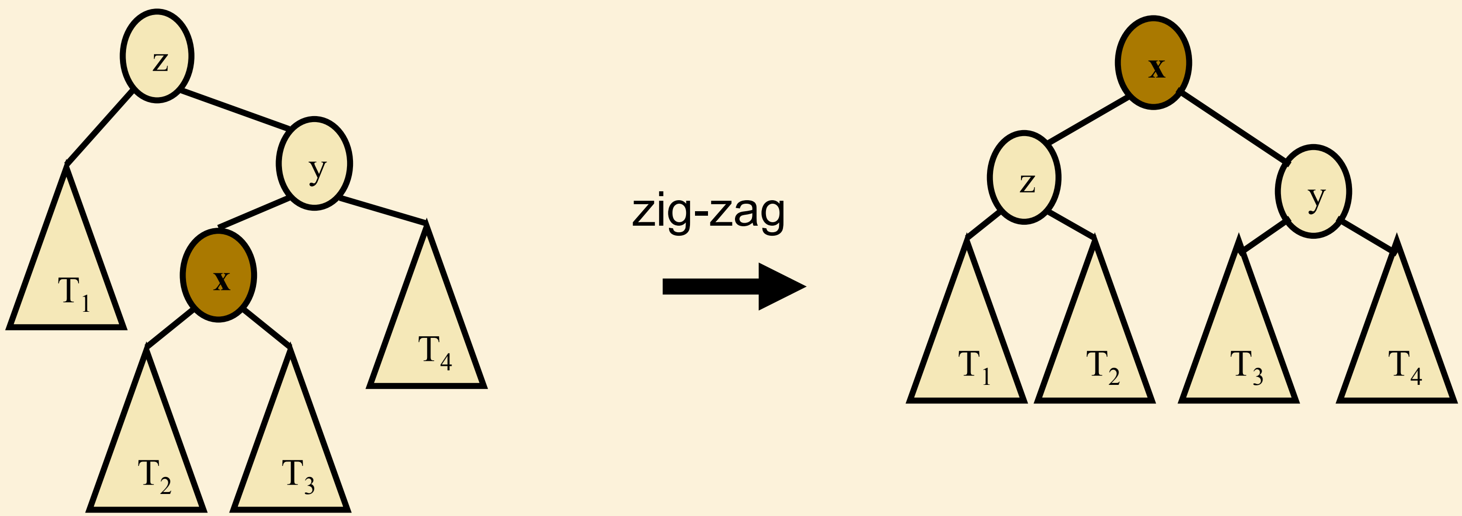
Zig-Zig

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



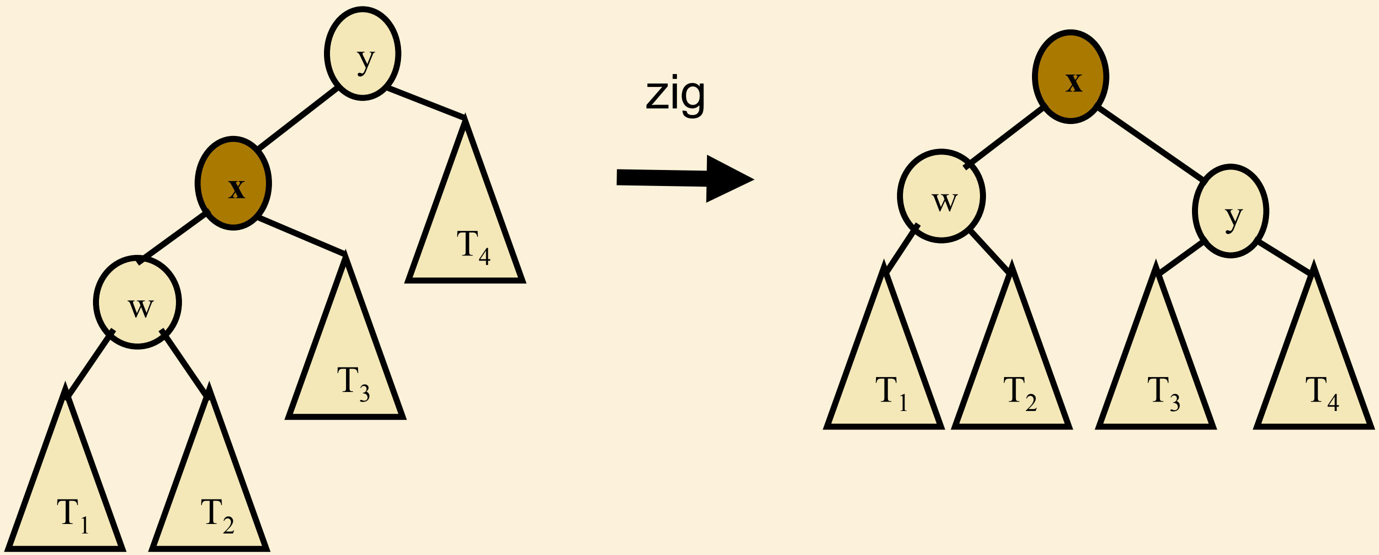
Zig-Zag

- 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



Summary of Topics

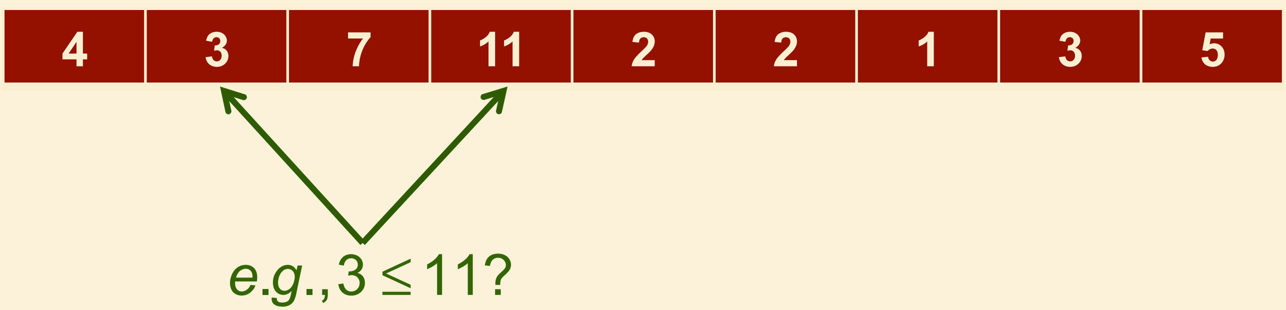
1. Maps & Hash Tables
2. Binary Search & Loop Invariants
3. Binary Search Trees
4. **Sorting**
5. Graphs

Sorting Algorithms

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

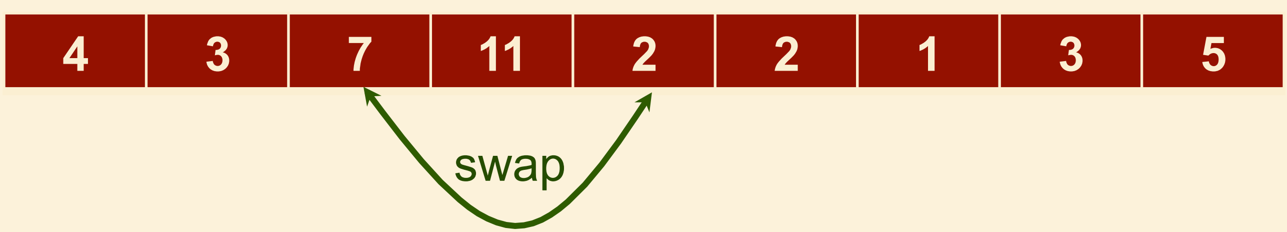
Comparison Sorts

- 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.



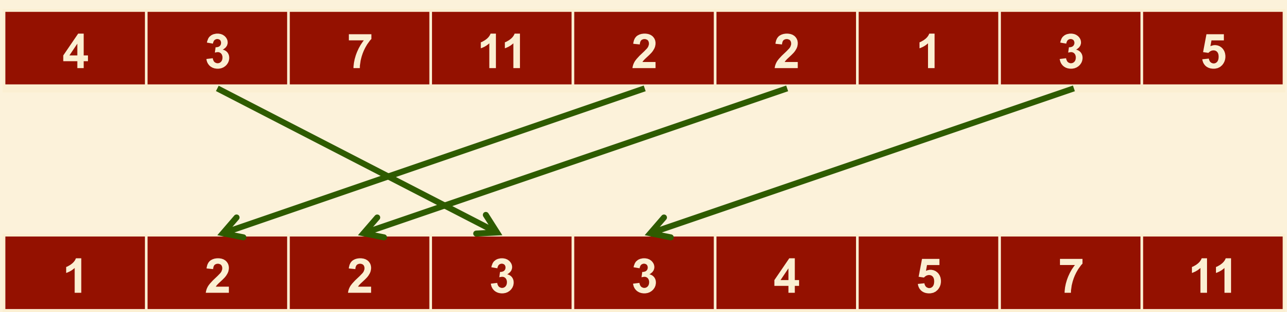
Sorting Algorithms and Memory

- 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.)

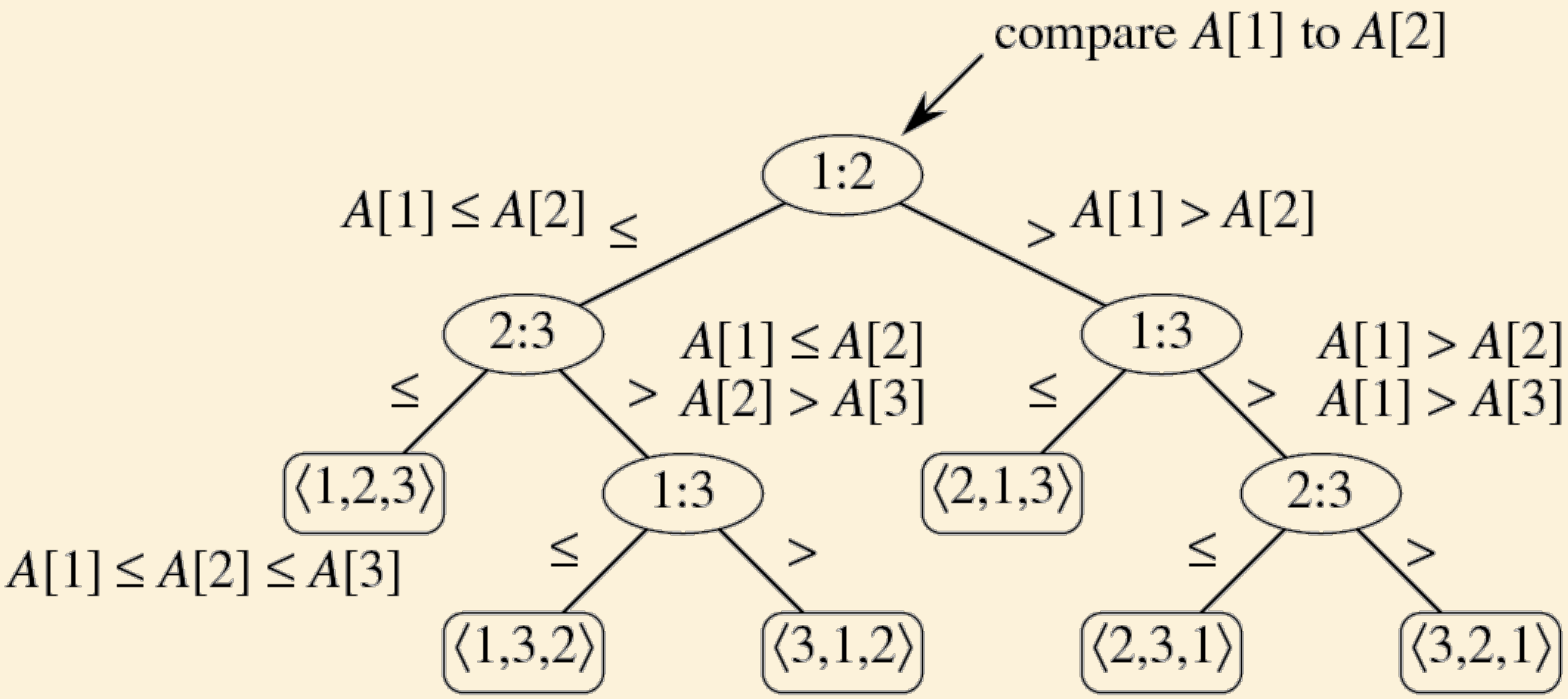


Summary of Comparison Sorts

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

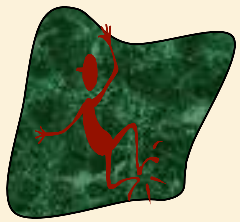
Input:	~~1~~	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:						1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

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.

RadixSort

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



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

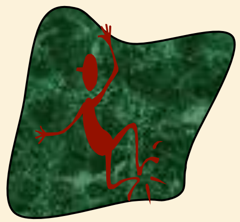
Is sorted wrt first i+1 digits.

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.

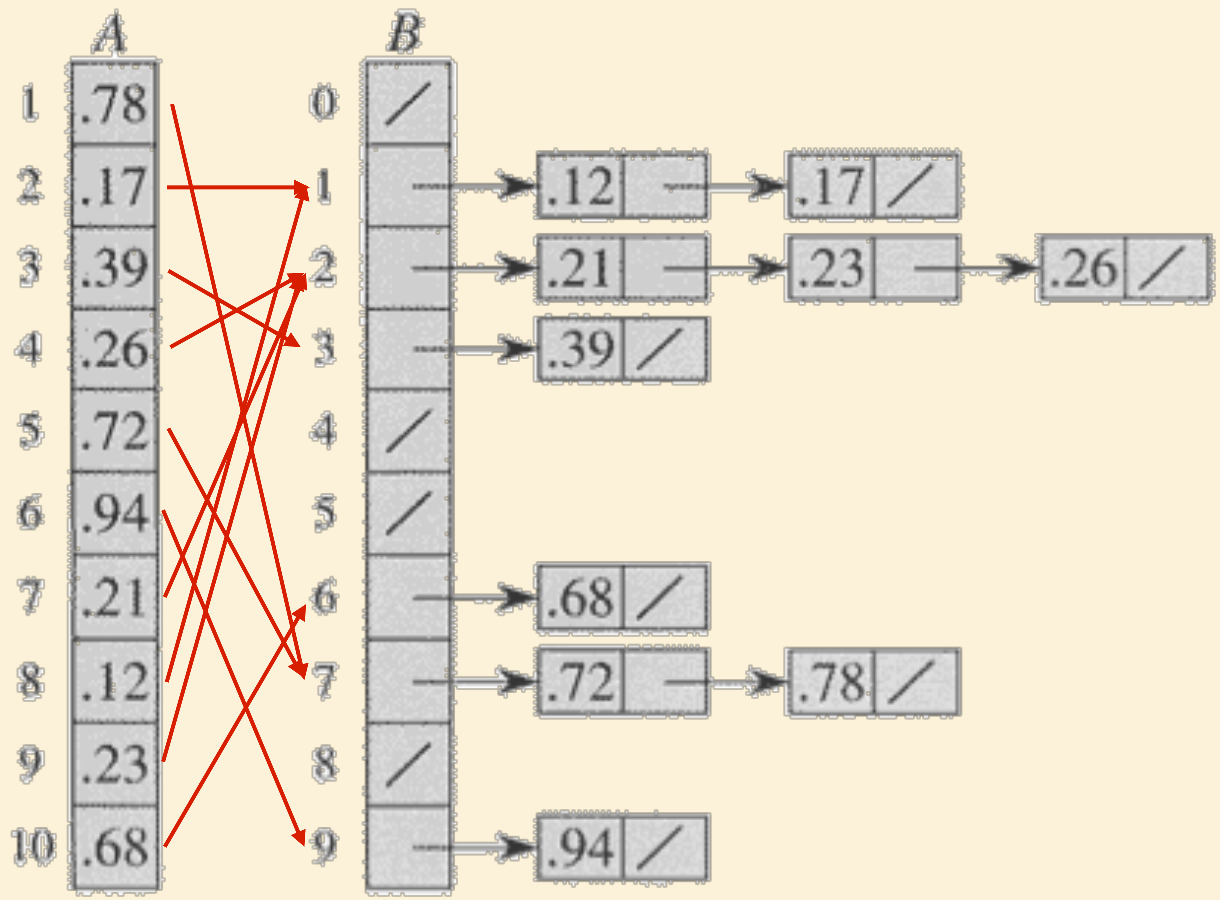
1	25
1	34
1	43
2	24
2	25
2	43
3	25
3	33
3	34
3	44

Is sorted wrt first i+1 digits.

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

Bucket Sort

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



Summary of Topics

1. Maps & Hash Tables
2. Binary Search & Loop Invariants
3. Binary Search Trees
4. Sorting
5. **Graphs**

Graphs

- Definitions & Properties
- Implementations
- Depth-First Search

Properties

Property 1

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

Proof: each edge is counted twice

Property 2

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

$$|E| \le |V|\,(|V| - 1)/2$$

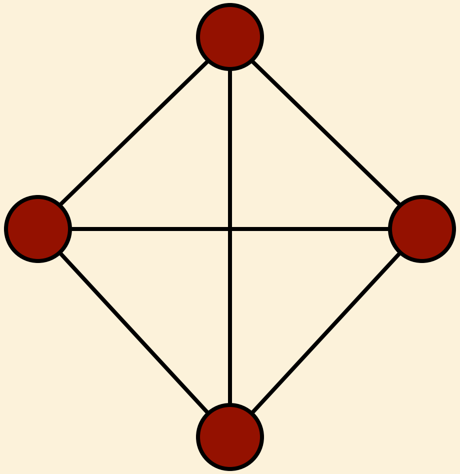
Proof: each vertex has degree at most $(|V| - 1)$

Q: What is the bound for a digraph?

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

Notation

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



Example

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

DFS Algorithm Pattern

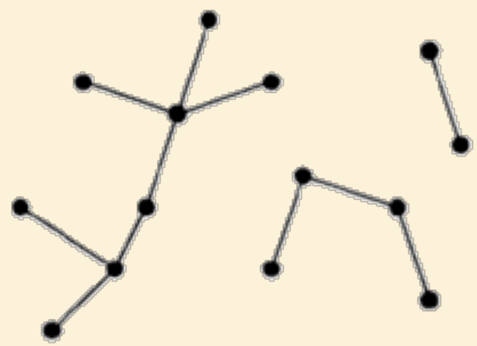
DFS(G)

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] $\leftarrow$ RED

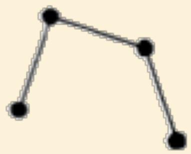
for each $v \in$ Adj[u] //explore edge (u,v)

if color[v] = BLACK

DFS-Visit(v)

$colour[u] \leftarrow GRAY$

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

Summary of Topics

1. Maps & Hash Tables
2. Binary Search & Loop Invariants
3. Binary Search Trees
4. Sorting
5. Graphs