CPTS 223 Advanced Data Structure C/C++

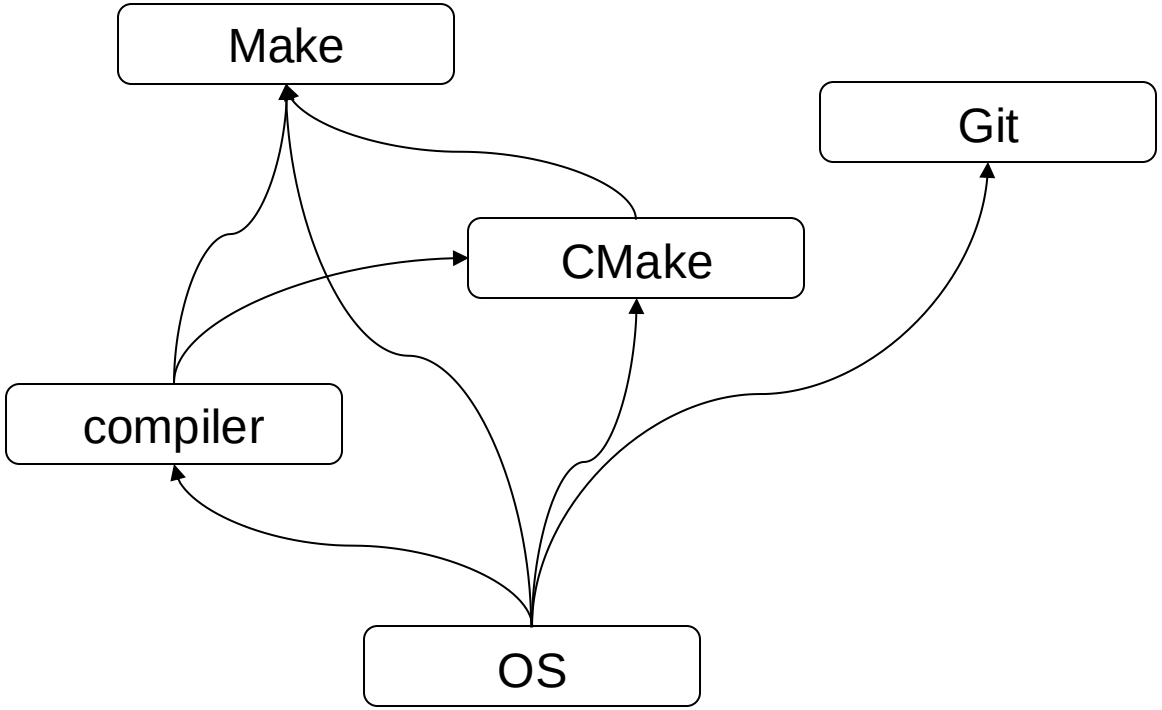
Midterm Exam Review

Overview

- Tree data structure
- Binary search trees
  - Support O(lg(n)) operations
  - Balanced trees
- STL set and map classes
- B-trees for accessing secondary storage
- Applications of Tree

This time: RB trees (practically used self-balanced BSTs)

Motivation



OS + compiler
OS + compiler + Make
OS + compiler + Make + Cmake

→ **executable** program
→ **automation** of executable program
→ **Cross-platform** automation of executable program

C++ templated class

```
template <class T>
class MyPair {
        T values [2];
 public:
        mypair (T first, T second)
        {
        values[0]=first;
        values[1]=second;
    }
};
```

- Templated class
- Type of ‘T’ defined at instantiation time
- Can have multiple templated types
- Works well if ‘T’ follows good OO design

```
MyPair<int> pair1 = new MyPair<int>();
```

Lvalues and Rvalues

- Lvalues
  - Permanent variables or objects
  - Persist beyond immediate use
  - Passed to a function with &
    - string & rstr = str;
- Rvalues
  - Temporary values
  - Could be as simple as a number
    - myfunc(2)
    - ‘2’ is a Rvalue

How about x+y? A lvalue or rvalue?

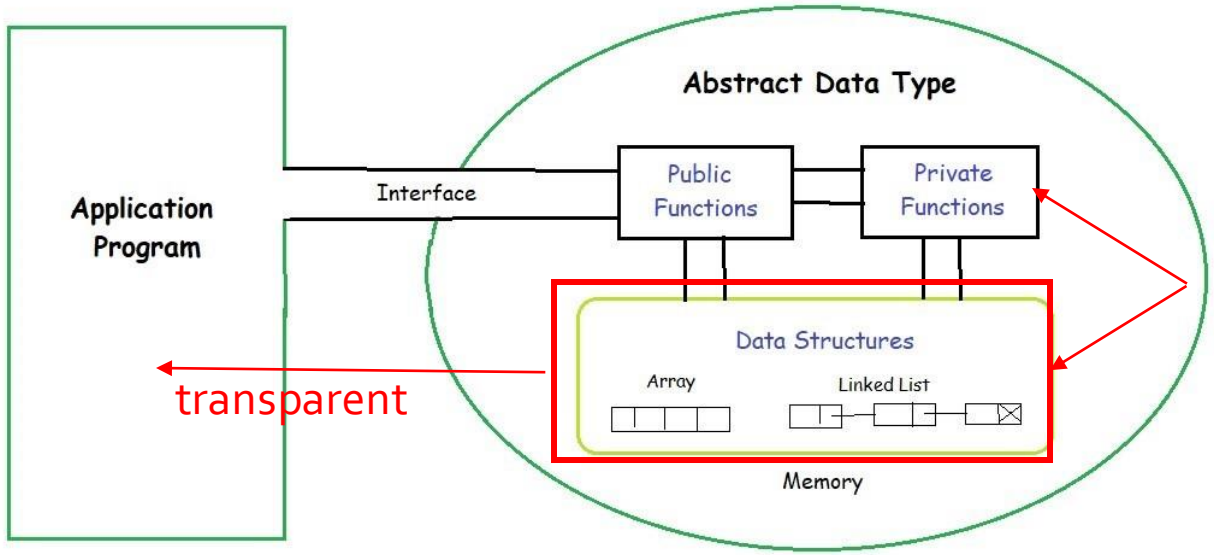
The Big-Five

- Big Five: why and when do you need to do them? Recourse management
  - Copy constructor
  - Move constructor
  - Copy Assignment operator=
  - Move Assignment operator=
  - Destructor

Interface of the Big-Five

```
~IntCell( );                                  // Destructor
IntCell( const IntCell & rhs );               // Copy constructor
IntCell( IntCell && rhs );                    // Move constructor
IntCell & operator= ( const IntCell & rhs );  // Copy assignment operator
IntCell & operator= ( IntCell && rhs );       // Move assignment operator
```

ADTs



Abstract:
hide the details

Image from https://www.geeksforgeeks.org/abstract-data-types/

Examples: running time?

```
// Assume A is an
integer array of size n
```

```
Algorithm1 (A, n)
        max = infinity;
        for (i=1 to n) {
                if(A[i]>max)
max=A[i];
        }
        Output max;
```

T(n) = n + const

```
Algorithm3 (A, 1, n)
        if (n<2) return;
        x = floor(n/2);
        T(1)=1;
        Algorithm3(A, 1, x)
        Algorithm3(A, x+1, n)
```

T(n) = n (1 + const)

Definition: (1) Let *T(n)* denote the time take by an algorithm on an input of size *n*. (2) T(1)=1

```
Algorithm2 (A, 1, n)
        if (n<2) return;
        mid = floor(n/2);
        if (condition#1)
                Algorithm2(A, 1,
mid);
        else
                Algorithm2(A,
mid+1, n);
```

T(n) = 1 + const * $log_2 n$

Is running time comparable?

Benchmarks can be sensitive

- Given the list {3, 9, 1, 2, 3, 5} as input
- Find(X): return the index of X
- We expect Find(3) to execute faster than Find(5)
  - Which is more representative of this input list, Find(3) or Find(5)?
  - How is our benchmark affected if we decide to use different lists?
    - e.g. {1, 2, 3, 3, 3, 4, 4, 4, 5, 9}
- Benchmarks are great for telling us how an algorithm will execute under a given set of circumstances
- but we ideally want something more generalizable

What is algorithm analysis?

- A mathematical technique for estimating the rate at which execution time grows relative to the size of its input parameters
- A formal name: asymptotic analysis

Is there non-asymptotic analysis?

- Asymptotic analysis is a method of estimation that groups algorithms based on their growth rate
- Asymptotic analysis is unable to tell us for sure how one algorithm will perform exactly in absolute timed execution relative to another
- but it does give us some good hints

Algorithm complexity

- T(n) is time to run given an input size of n elements
- T(n) = O(f(n)): exist [c, n_o] such that T(n) <= cf(n) when n >= n_o
  - e.g., T(n) <= 2.45 n^2, where f(n)=n^2
- T(n) = Ω(g(n)) when +[c, n_o] such that T(n) >= cg(n) when n >= n_o
  - e.g., T(n) >= 1.03 n^2, where g(n)=n^2
- T(n) = Θ(h(n)) if and only if T(n) = O(h(n)) and T(n) = Ω(h(n))
  - e.g., 1.03 n^2 <= T(n) <= 2.45 n^2, where h(n)=n^2

Bounds

- O(f(n)) is an UPPER bound of T(n) -- "Worst case can be no more than" ←T(n) <= cf(n)
- Ω(g(n)) is a LOWER bound of T(n) -- "Best case can be no faster than" ←T(n) >= cg(n)
- Only the order of the algorithm
- No details, e.g., constants (asymptotic analysis)
- T(n) = Θ(g(n)) is when O(g(n)) = Ω(g(n)) -- "It must be exactly"
- You will find Theta (Θ) also used as an average case

Nested loops

- for(int i = 0; i < num_items; i++) {
  for(int j = 0; j < num_items; j++) {
  swap(items[i], items[j])
  }
  }
- In this case, we multiply the effect that num_items has on the growth rate, yielding O(n^2)

Unrelated loops

```
for(int i = 0; i < num_items; i++) {
        cout << "hello";
}
for(int j = 0; j < num_items; j++) {
        cout << "goodbye";
}
```

- O(n + n) or O(2n) → simplify to O(n)

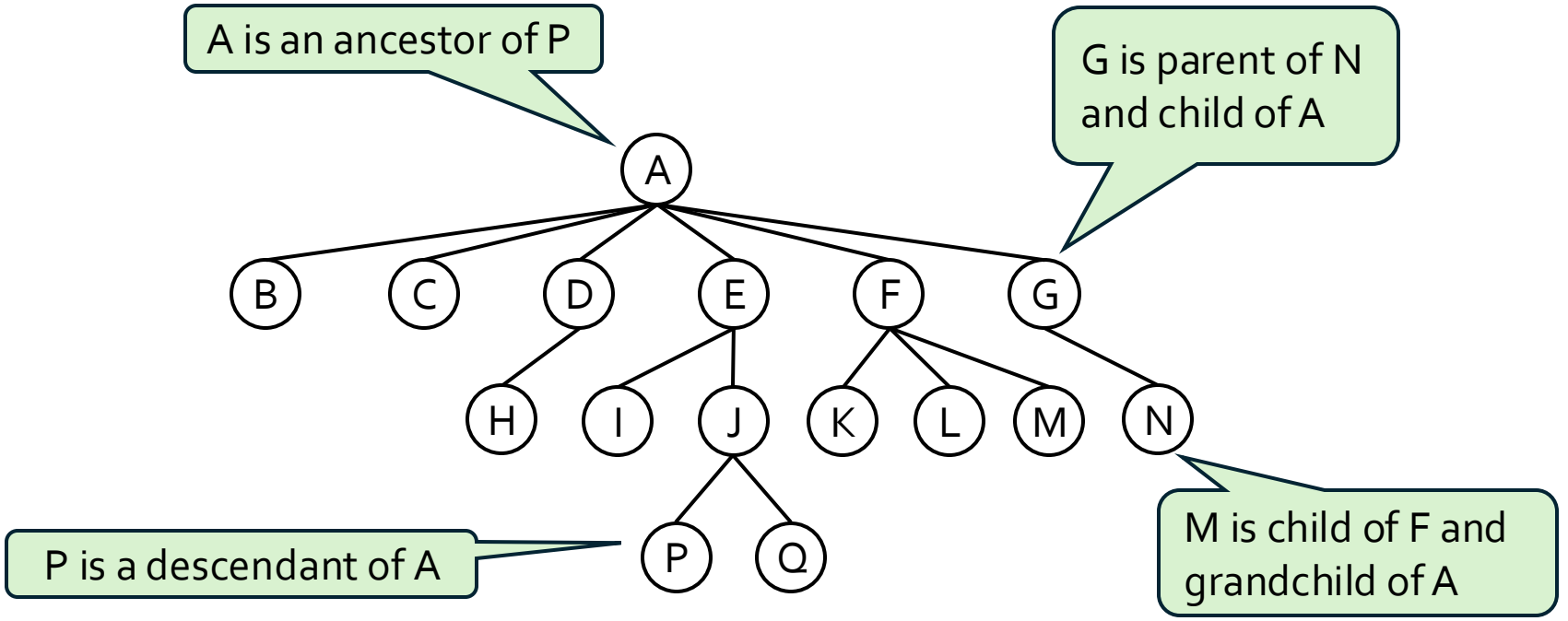
Reduction of non-constant time

- In Big-O analysis, we always drop coefficients:
  - O(2n) → O(n)
  - O(4on) → O(n)
  - O(1000000n) → O(n)
- This is because Big-O cares about placing algorithms into performance groups, not absolute T(n) calculations

How to compare algorithms?

Function	Name
c	Constant
log(n)	Logarithmic
log^2(n)	Log-squared
n	Linear
n log(n)	(Will see this in sorting *a lot*)
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Trees


A is an ancestor of P
G is parent of N and child of A
A
B
C
D
E
F
G
H
I
J
K
L
M
N
P
Q
P is a descendant of A
M is child of F and grandchild of A

Basic definitions

- A tree T is a set of nodes that form a directed acyclic graph (DAG) such that:
  - Each non-empty tree has a root node and zero or more sub-trees T1, …, Tk
  - Each sub-tree is a tree
  - An internal node is connected to its children by a directed edge
- Each node in a tree has only one parent
  - Except the root, which has no parent

Recursive definition

Basic definitions

- Internal node: nodes with at least one child
- Leaf node: nodes with no children

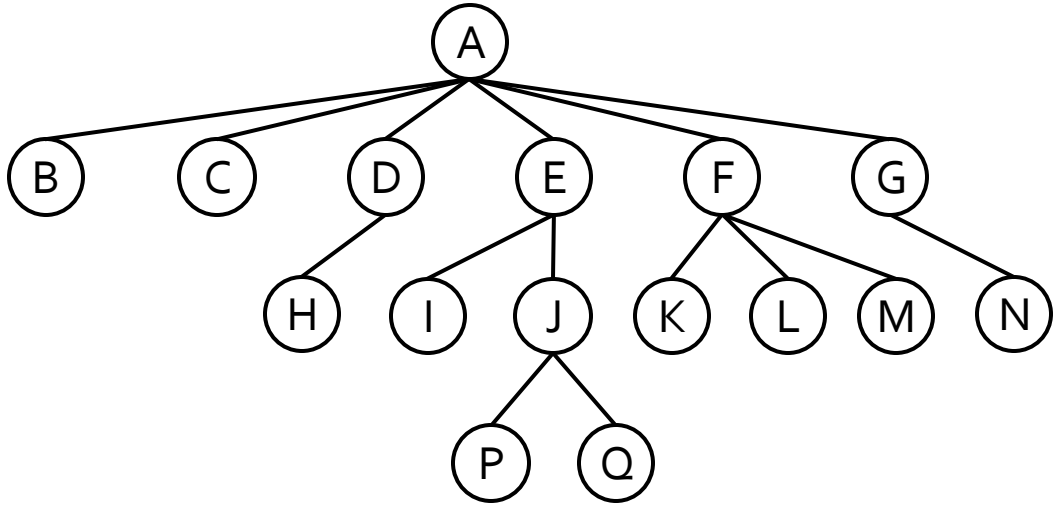
Nodes → either a leaf or an internal node

- Siblings: nodes with the same parent
- A path from node n_1 to n_k is a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k$
  - The length of a path is the number of edges on the path (i.e., k-1)
  - Each node has a path of length o to itself
  - There is exactly one path from the root to each node in a tree
  - Nodes n_i, …, n_k are descendants of n_i and ancestors of n_k
  - Nodes n_{i+1}, …, n_k are proper descendants of n_i
  - Nodes n_i, …,n_{k-1} are proper ancestors of n_k

Basic definitions

- The depth of a node n_i is the length of the path from the root to n_i
  - The root node has a depth of 0
  - The depth of a tree == the depth of its deepest leaf
- The height of a node n_i is the length of the longest path under n_i's subtree
  - All leaves have a height of 0
- height of tree = height of root = depth of tree

Height and depth

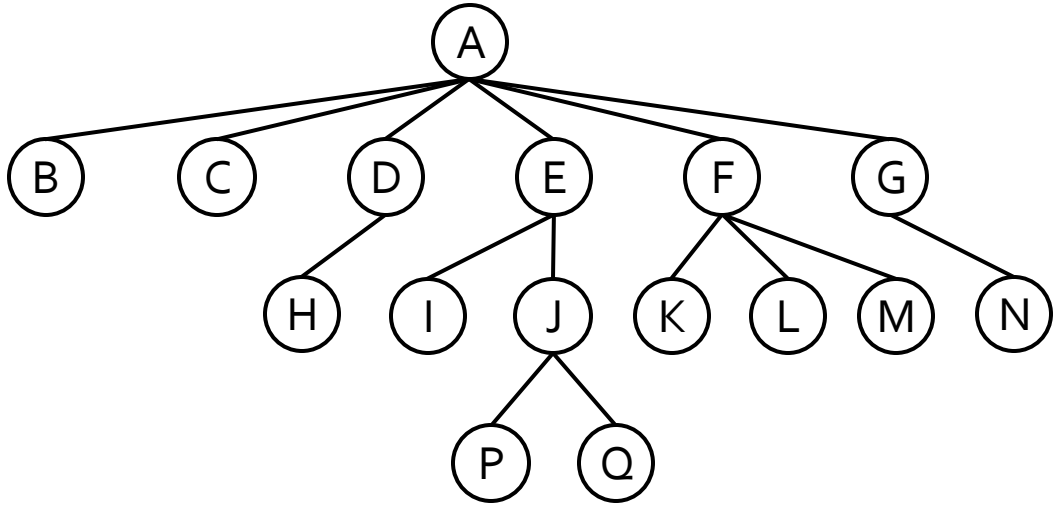


1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

e.g., height(E)=2, height(L)=0

WSU

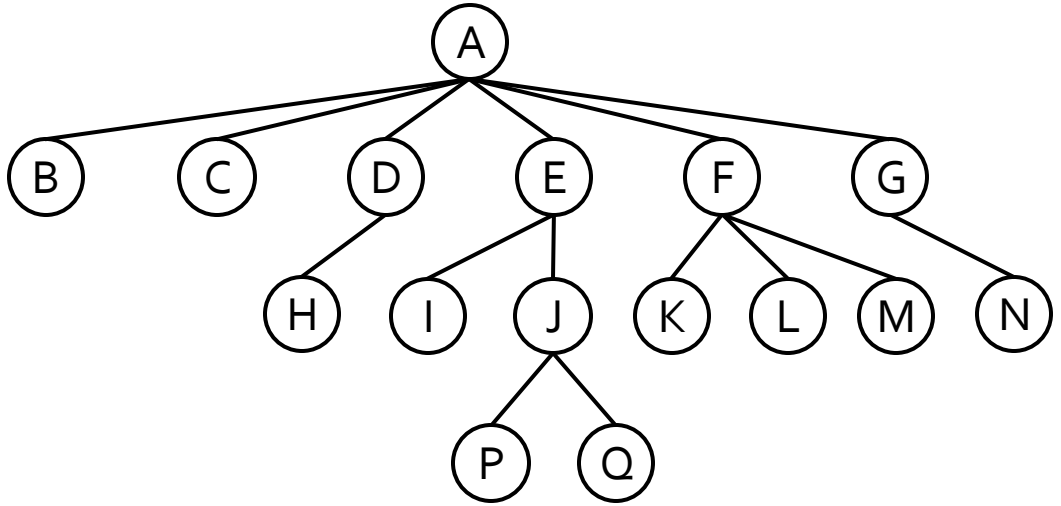
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

= 3 (height of longest path from root)

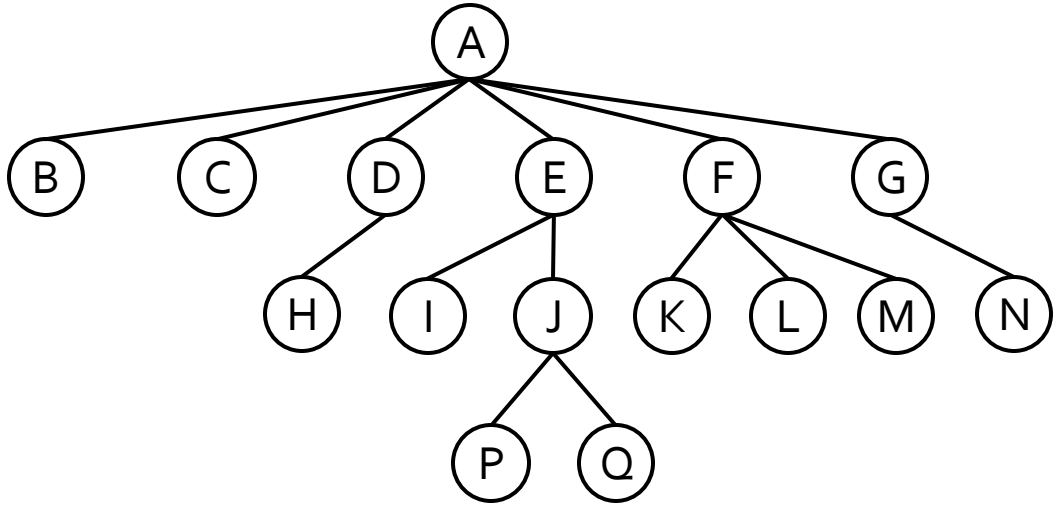
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

e.g., depth(E)=1, depth(L)=2

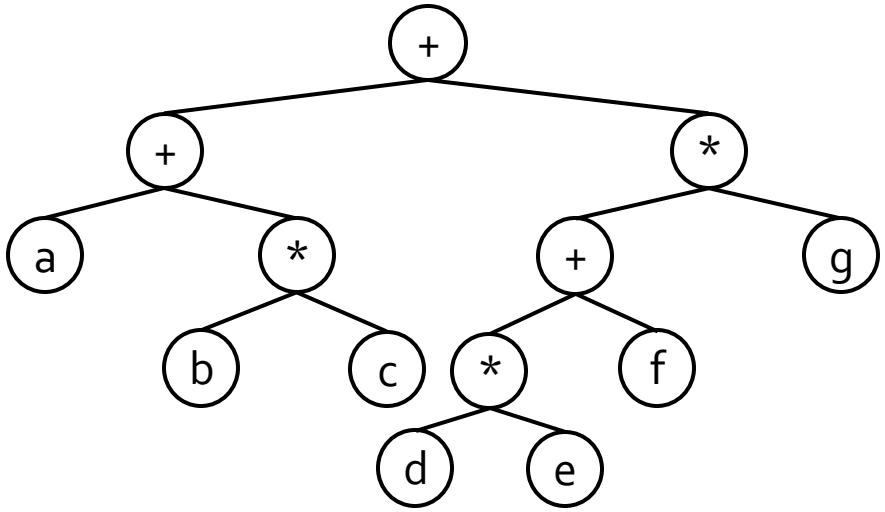
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

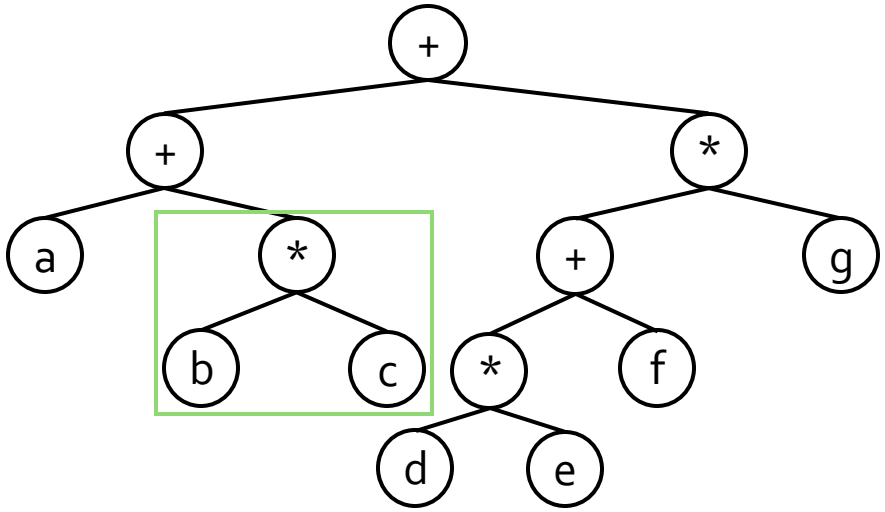
= 3 (length of the path to the deepest node)

Traversals



- Pre-order: root - left - right
- Post-order: left - right - root
- In-order: left - root - right

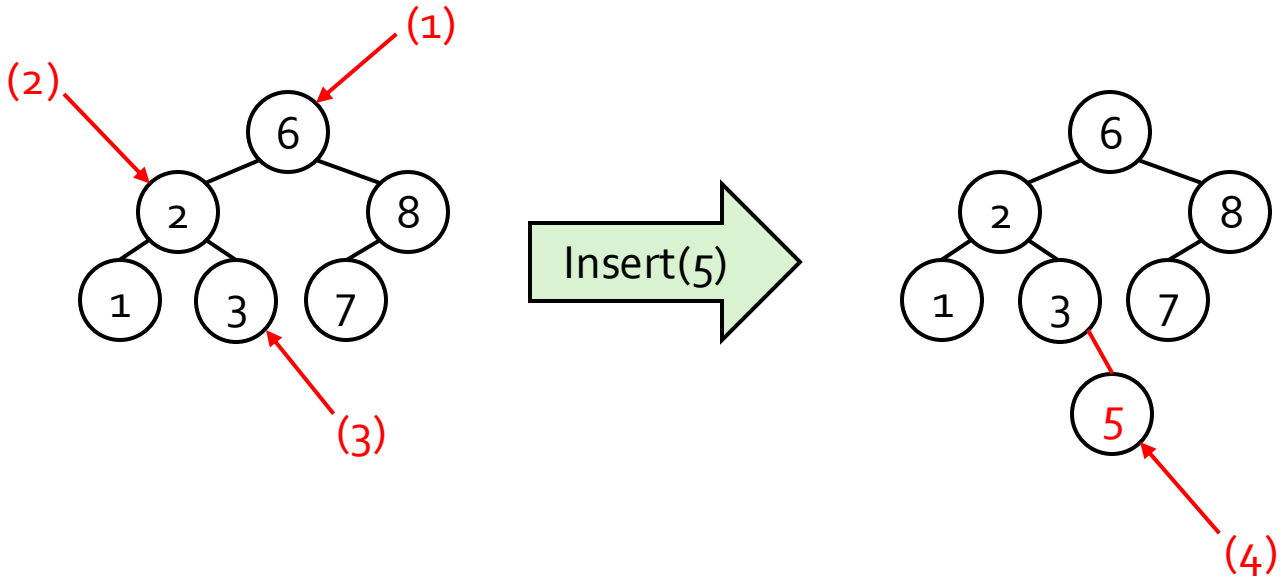
Traversals



- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

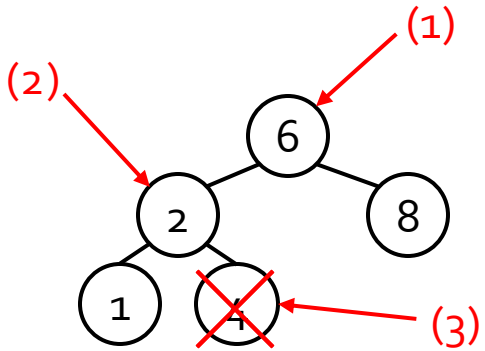
Inserting into BSTs

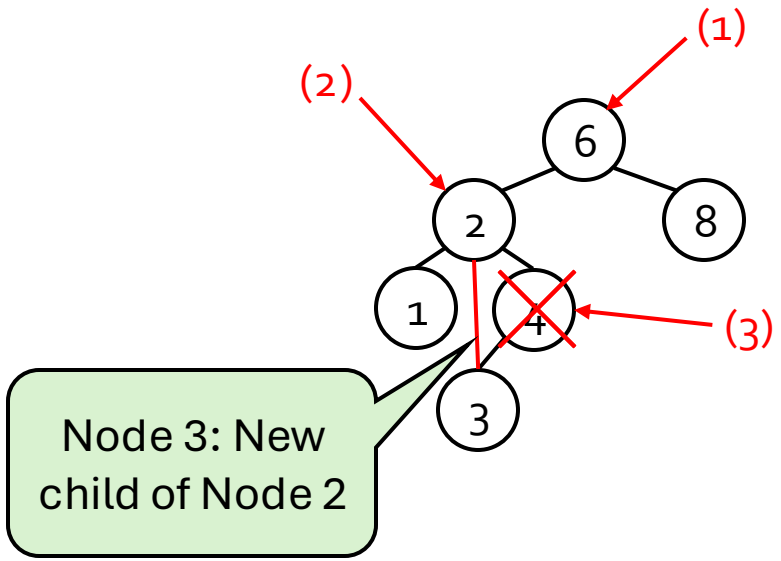
- e.g., insert 5



Removing from BSTs

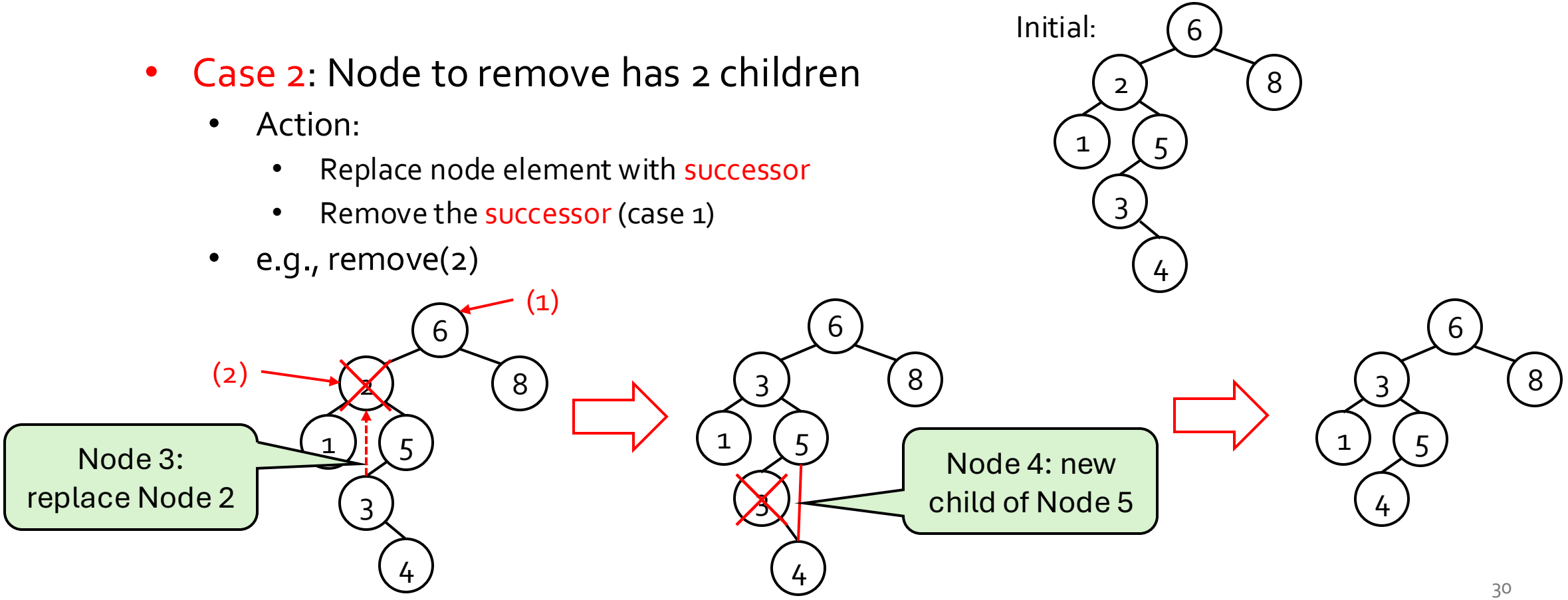
- There are two cases for removal
- Case 1: Node to remove has 0 or 1 child
  - Action: remove it and make appropriate adjustments to retain BST structure
  - e.g., remove(4)





Removing from BSTs

- Case 2: Node to remove has 2 children
  - Action:
    - Replace node element with successor
    - Remove the successor (case 1)
  - e.g., remove(2)

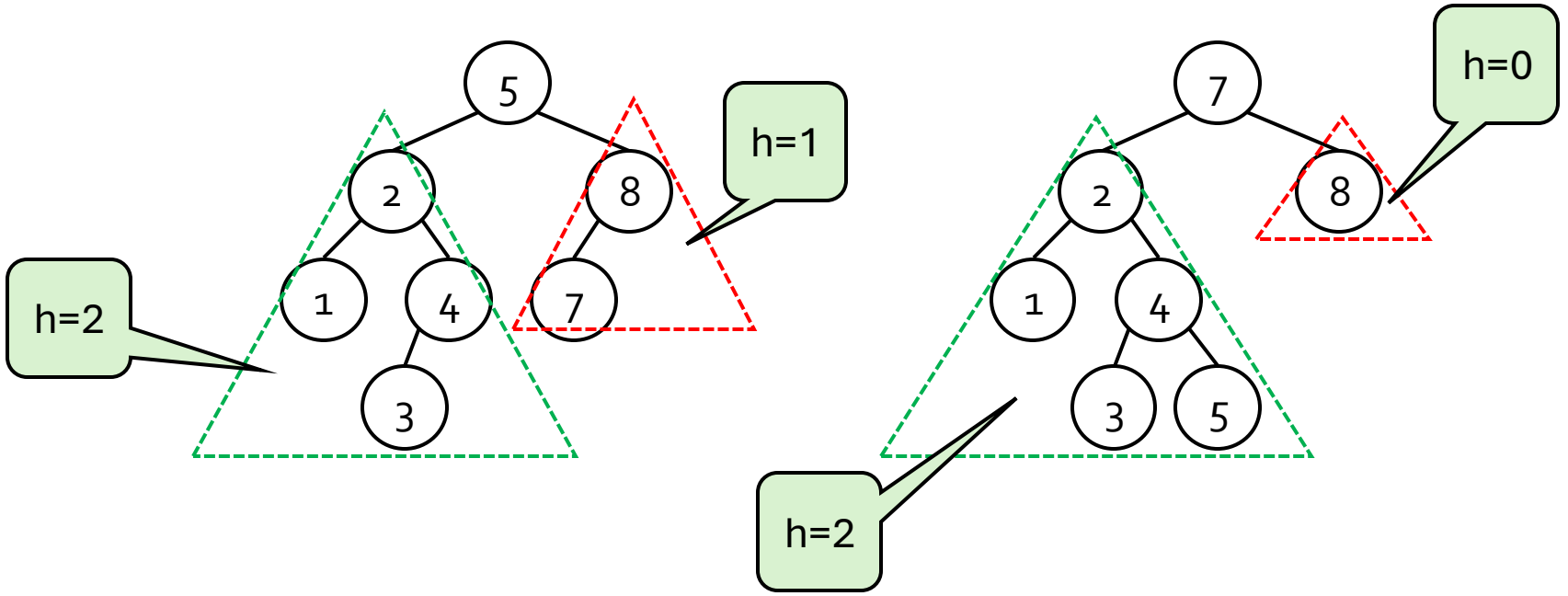


Balanced BSTs

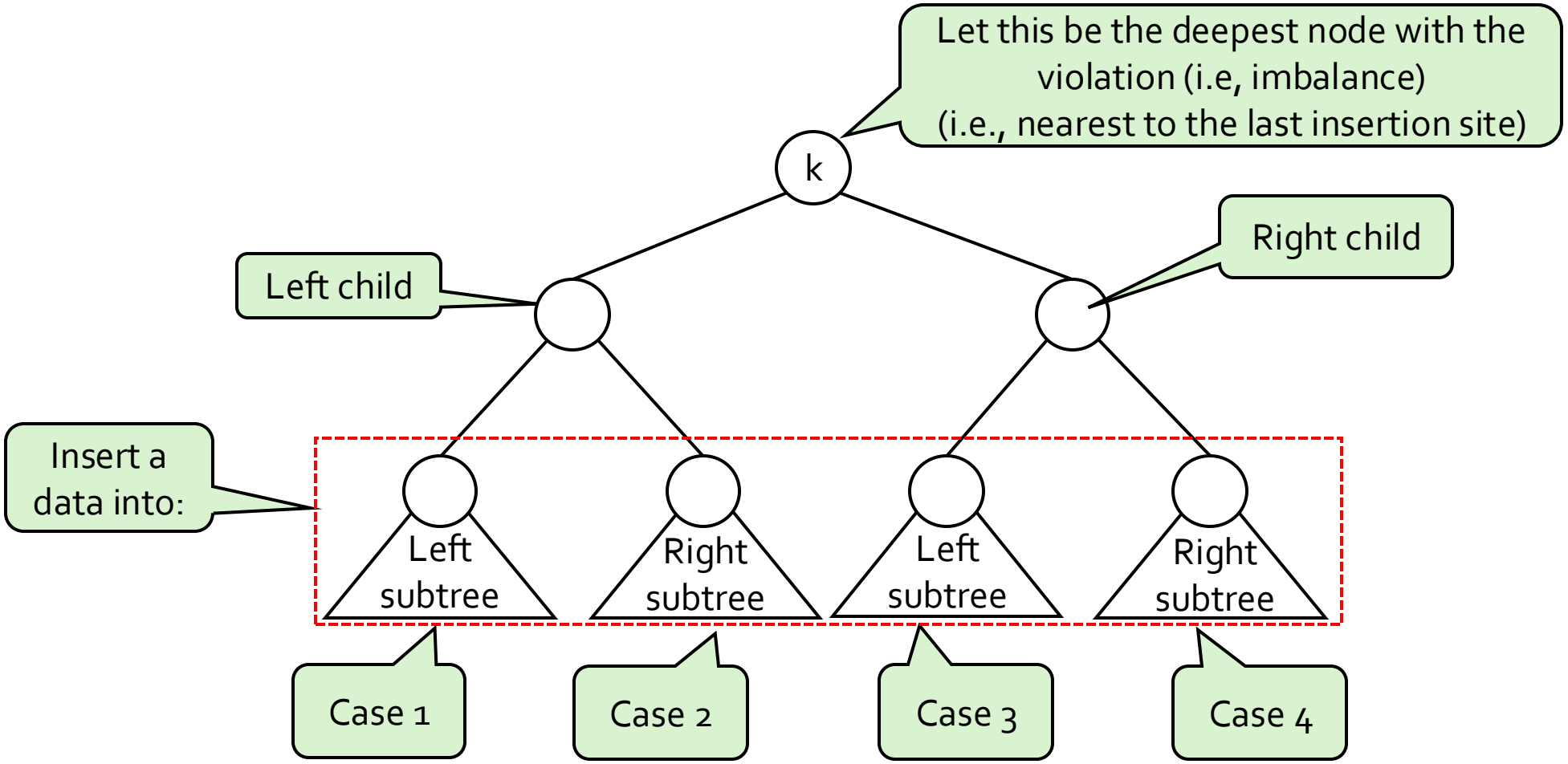
- AVL (Adelson-Velskii and Landis) trees
  - Height of left and right subtrees at every node in BST differ by at most 1
  - Balance forcefully maintained for every update (via rotations)
  - BST depth always O(lg(n))

AVL trees

- Which of these is a valid AVL tree?



Identify cases for AVL insert



AVL insert (single rotation)

- Case 1: single rotation right

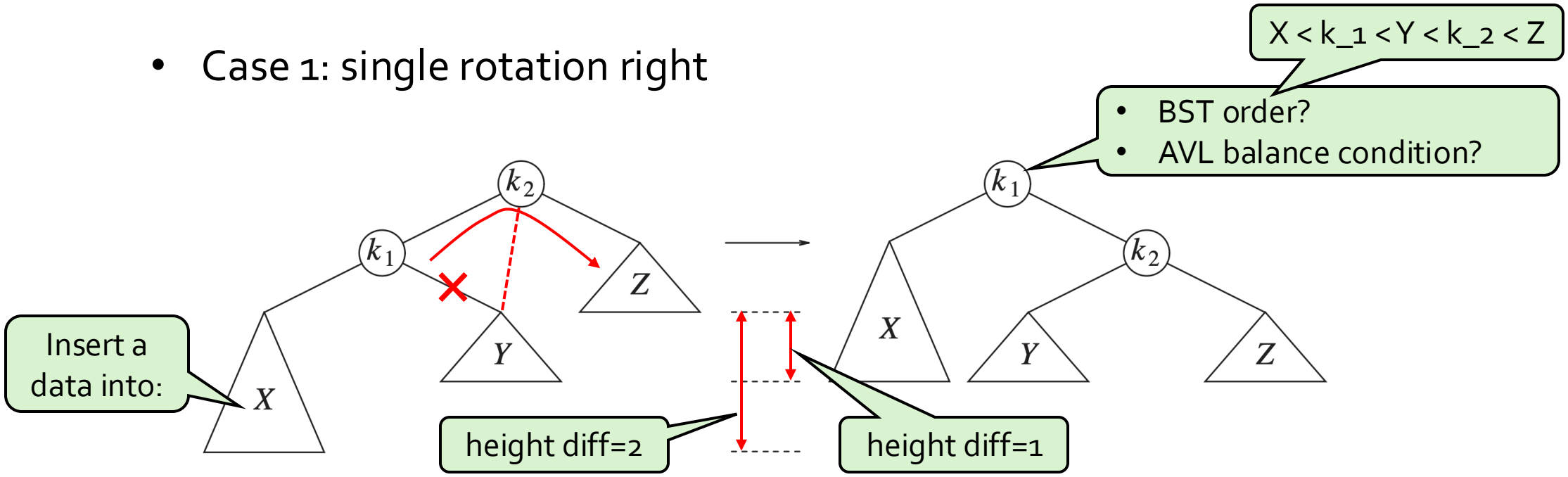


Figure 4.34 Single rotation to fix case 1

AVL insert (double rotation)

- Case 2: left-right double rotation

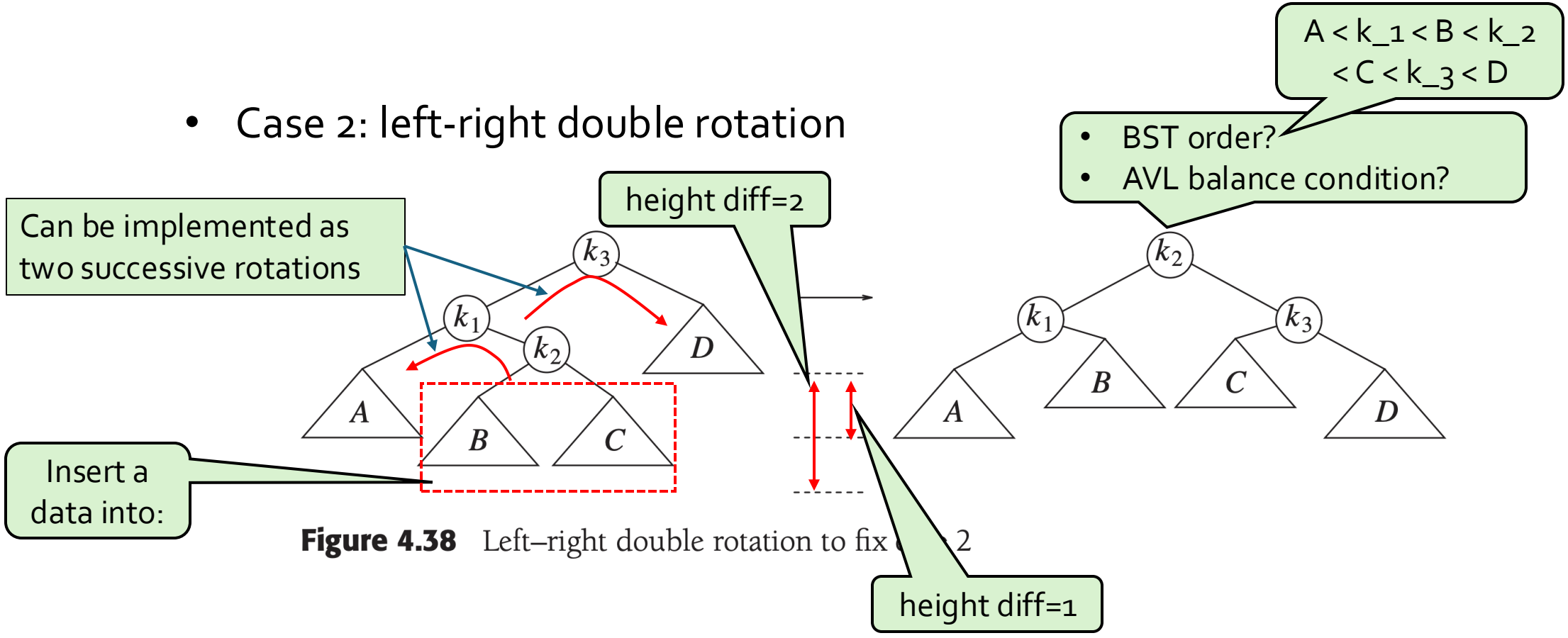


Figure 4.38 Left–right double rotation to fix

Splaying: zig-zag

- Node X is right-child of parent, which is left-child of grandparent (or vice-versa)
- Perform double rotation (left, right)

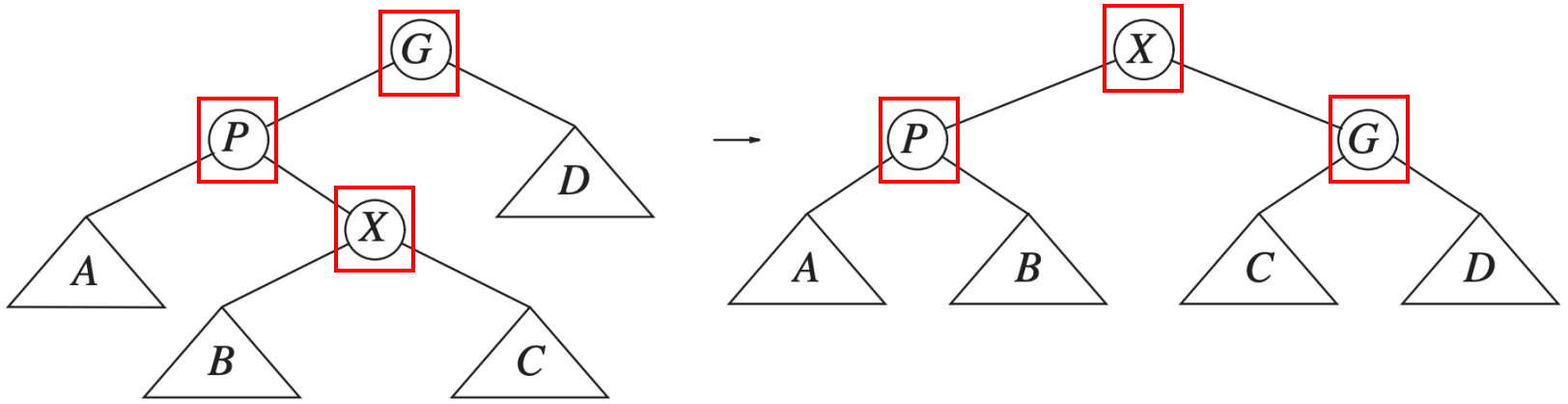


Figure 4.48 Zig-zag

Splaying: zig-zig

- Node X is left-child of parent, which is left-child of grandparent (or right-right)
- Perform double rotation (right-right)

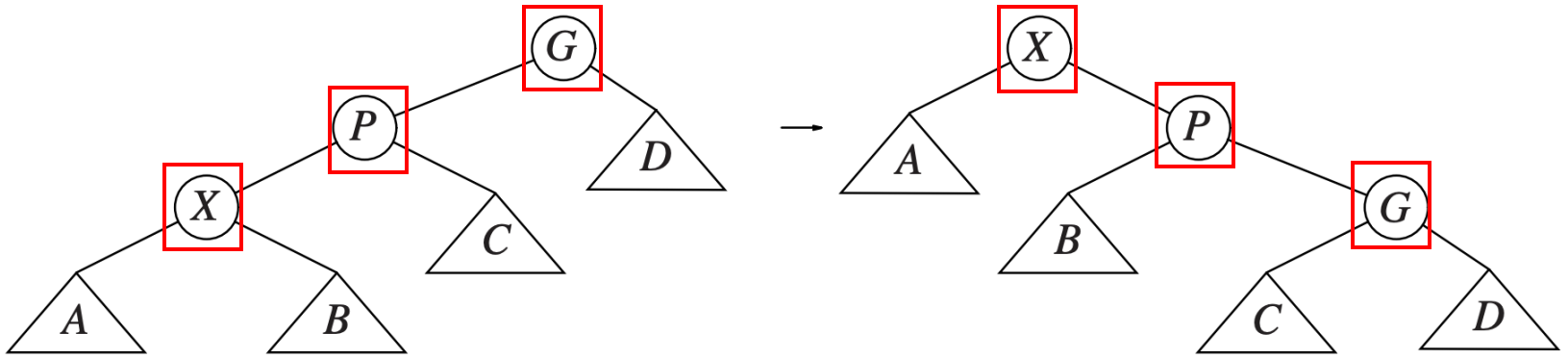


Figure 4.49 Zig-zig

B-tree (B+ tree) definition

Parameters: M, L

- Leaves store the real data items
- Internal nodes store up to M-1 keys
  - s.t., key i is the smallest key in subtree i+1
- Root can have between 2 to M children
- Each internal node (except root) has between ceil(M/2) to M children
- All leaves are at the same depth
- Each leaf has between ceil(L/2) and L data items, for some L

B-tree of order 5

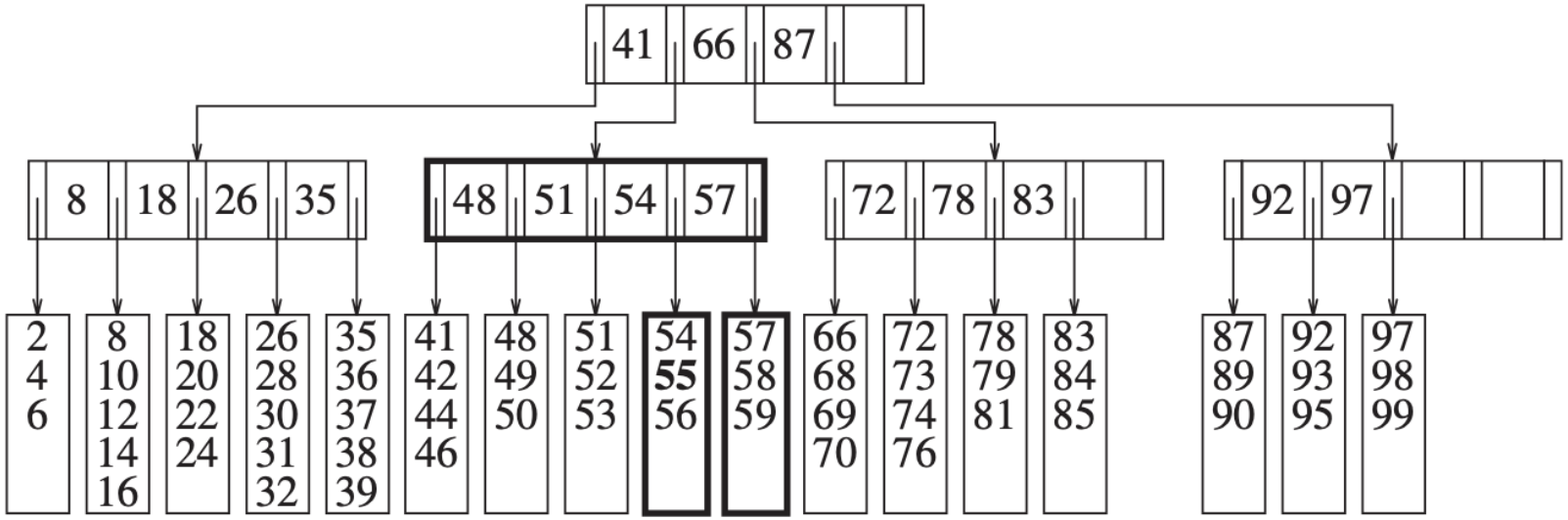


Figure 4.65 Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves

B-tree of order 5: insert(40)

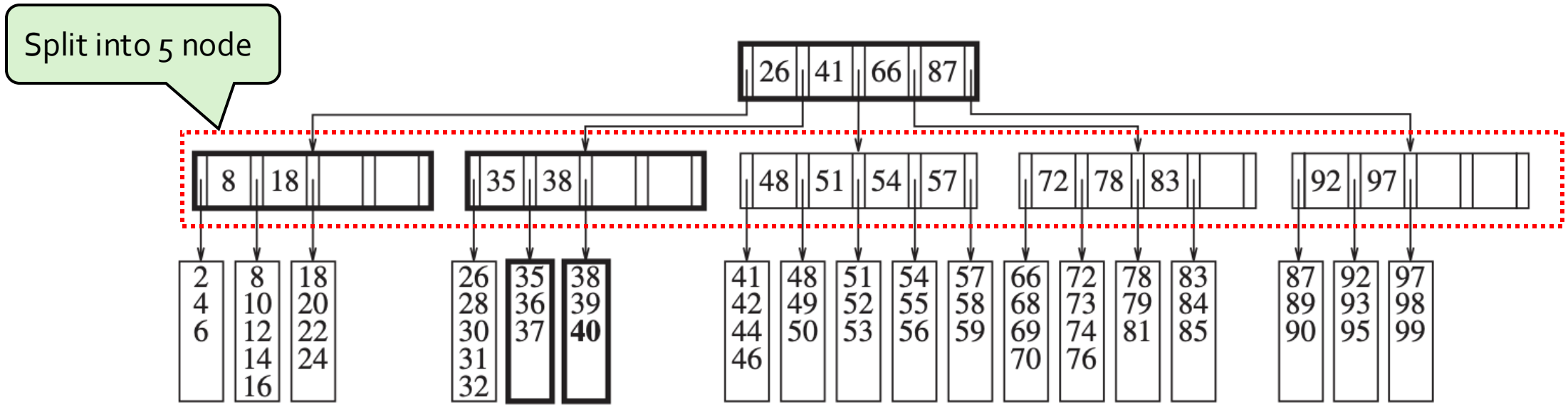


Figure 4.66 Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

B-tree of order 5: delete(99)

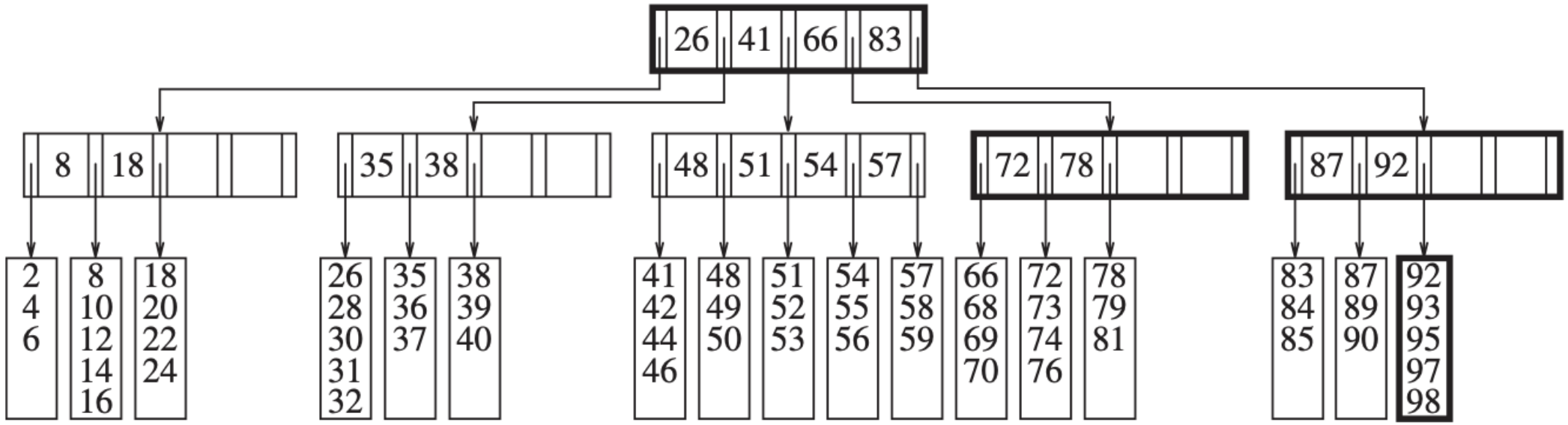


Figure 4.67 B-tree after the deletion of 99 from the B-tree in Figure 4.66

STL set and map classes