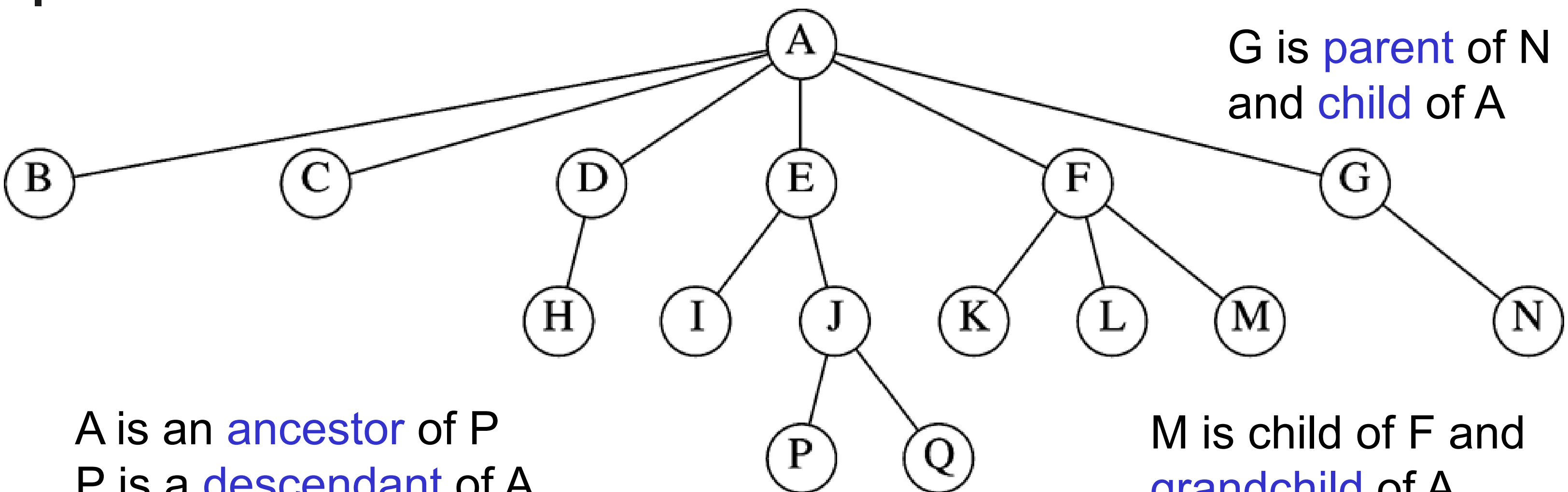
Trees

Overview

- Tree data structure
- Binary search trees
  - Support $O(\log_2 N)$ operations
  - Balanced trees
- STL set and map classes
- B-trees for accessing secondary storage
- Applications

Trees

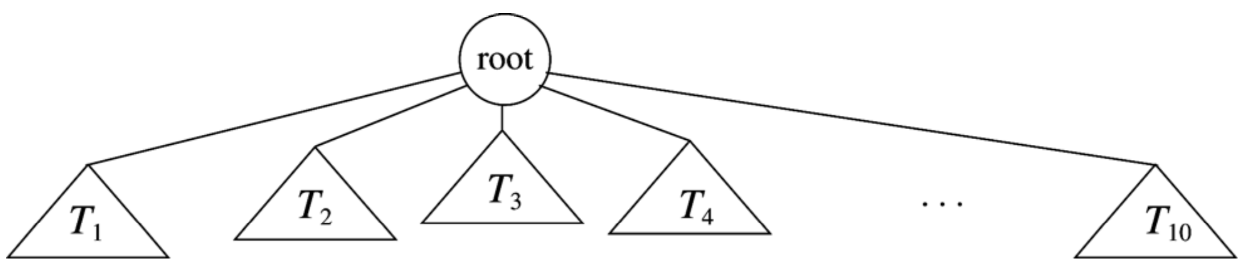


G is parent of N
and child of A

A is an ancestor of P
P is a descendant of A

M is child of F and
grandchild of A

Generic
Tree:



Definitions

- A tree T is a set of nodes that form a directed acyclic graph (DAG) such that:
  - Each non-empty tree has a <u>root</u> node and zero or more sub-trees $T_1, \ldots, T_k$
  - Each sub-tree is a tree
  - An internal node is connected to its children by a directed <u>edge</u>

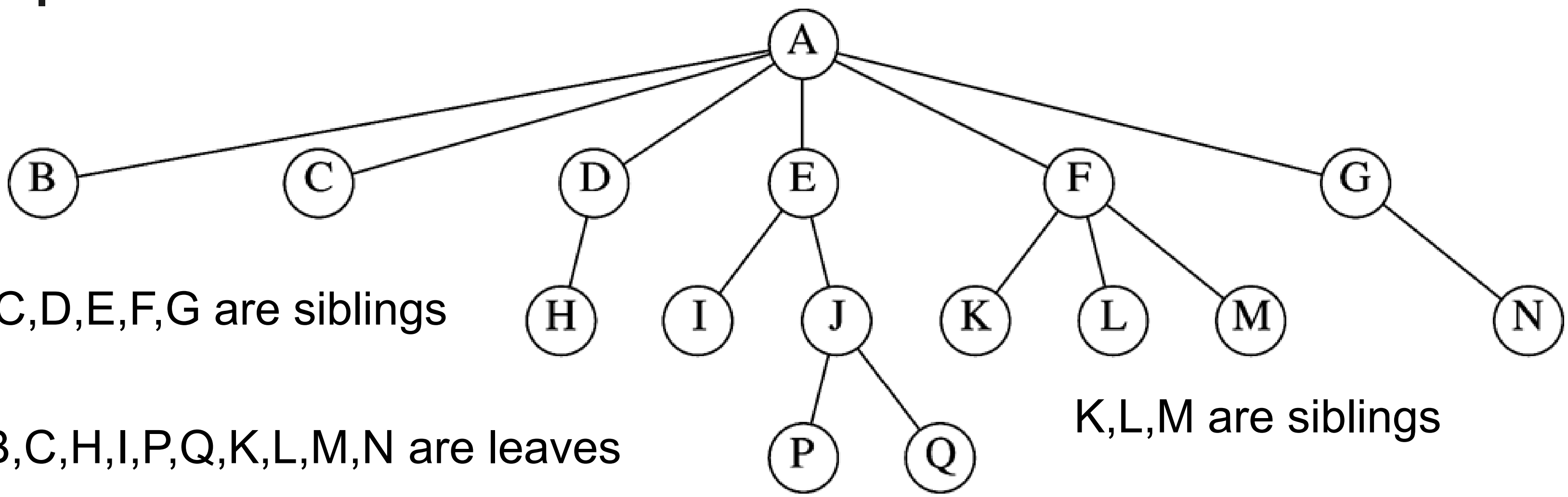
Recursive definition

- Each node in a tree has only one parent
  - Except the root, which has no parent

Definitions

- Nodes with at least one child is an internal node
- Nodes with no children are leaves
- "Nodes" = Either a leaf or an internal node
- Nodes with the same parent are siblings
- 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 $1 \le i < k$
  - The length of a path is the number of edges on the path (i.e., k-1)
  - Each node has a path of length 0 to itself
  - There is exactly one path from the root to each node in a tree
  - Nodes $n_i,\ldots,n_k$ are descendants of n_i and ancestors of n_k
  - Nodes $n_{i+1},\ldots, n_k$ are proper descendants
  - Nodes $n_i,\ldots,n_{k-1}$ are proper ancestors of n_i

Definitions: node relationships



B,C,D,E,F,G are siblings

B,C,H,I,P,Q,K,L,M,N are leaves

K,L,M are siblings

The path from A to Q is A – E – J – Q (with length 3)
A,E,J are proper ancestors of Q
E,J,Q, I,P are proper descendants of A

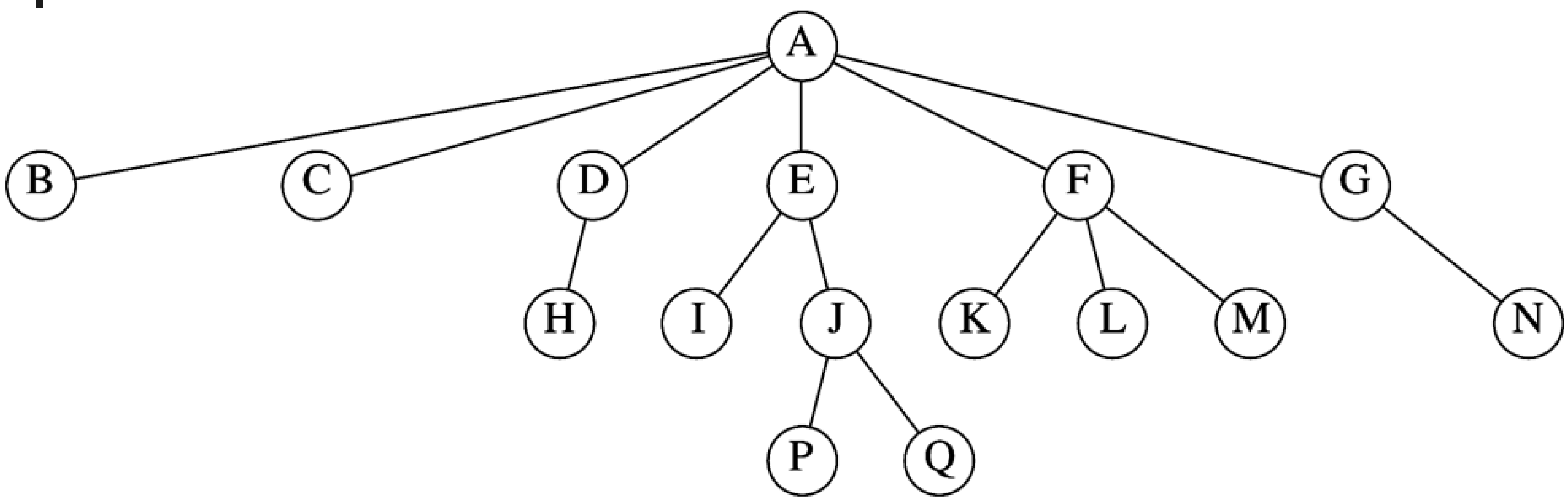
Definitions: Depth, Height

- The <u>depth</u> 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 is the depth of its deepest leaf

Can there be more than one?

- The <u>height</u> 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

Trees



Height of each node? — e.g., height(E)=2, height(L)=0

Height of tree? — = 3 (height of longest path from root)

Depth of each node? — e.g., depth(E)=1, depth(L)=2

Depth of tree? — = 3 (length of the path to the deepest node)

Implementation of Trees

- Solution 1: Vector of children

```
Struct TreeNode
{
  Object element;
  vector<TreeNode> children;
}
```

Direct access to children[i]
but…
Need to know max allowed children in advance
& more space

- Solution 2: List of children

```
Struct TreeNode
{
  Object element;
  list<TreeNode> children;
}
```

Number of children can be dynamically determined but….
more time to access children

Also called “First-child, next-sibling”

Implementation of Trees

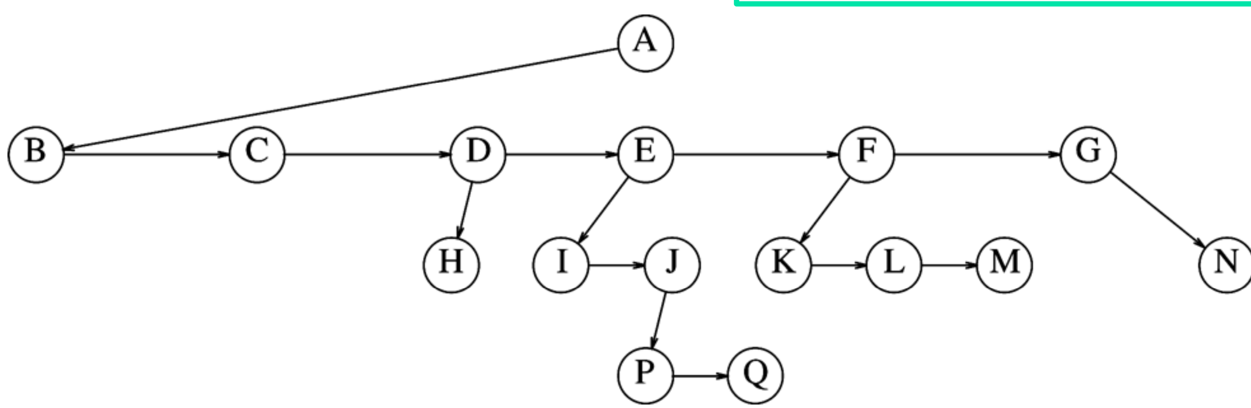
- Solution 3: Left-child, right-sibling

```
Struct TreeNode
{
  Object element;
  TreeNode *firstChild;
  TreeNode *nextSibling;
}
```

Guarantees 2 pointers per node (independent of #children)

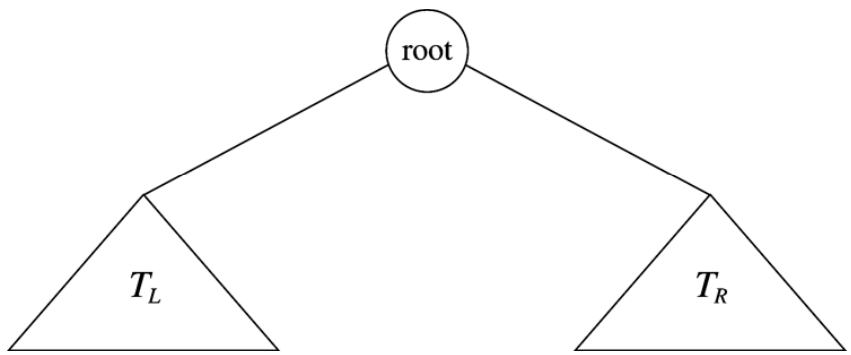
But…

Access time proportional to #children



Binary Trees (aka. 2-way trees)

- A binary tree is a tree where each node has *no more* than two children.

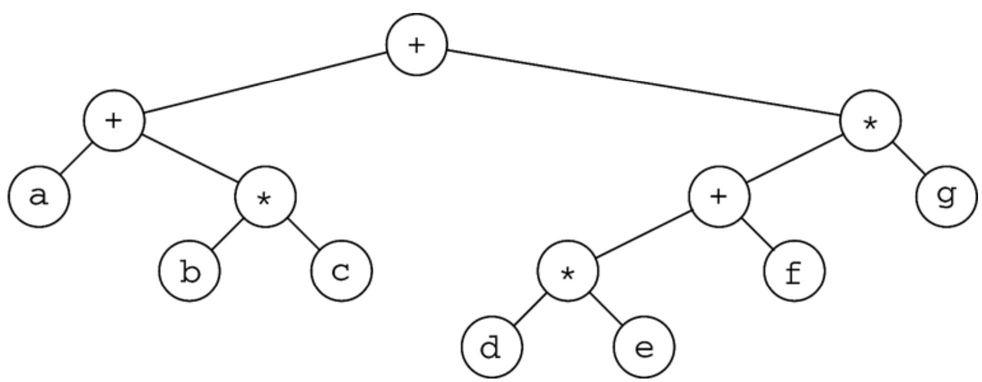



```
struct BinaryTreeNode
{
  Object element;
  BinaryTreeNode *leftChild;
  BinaryTreeNode *rightChild;
}
```

- If a node is missing one or both children, then that child pointer is **NULL**

Example: Expression Trees

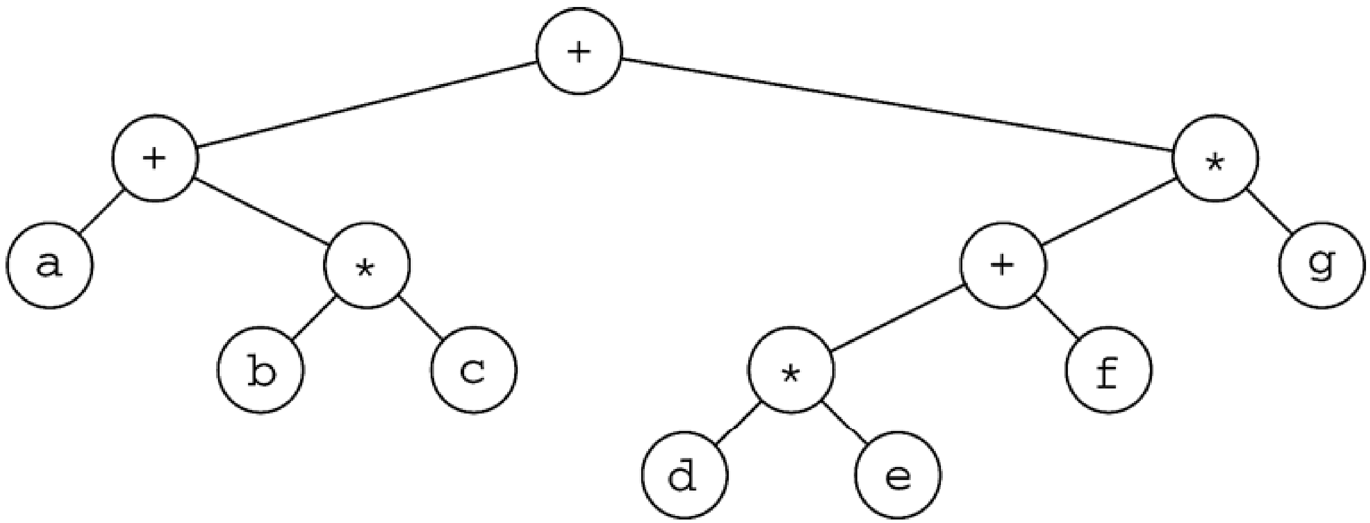
- Store expressions in a binary tree
  - Leaves of tree are operands (e.g., constants, variables)
  - Other internal nodes are unary or binary operators
- Used by compilers to parse and evaluate expressions
  - Arithmetic, logic, etc.
- E.g., (a + b * c)+((d * e + f) * g)



Example: Expression Trees

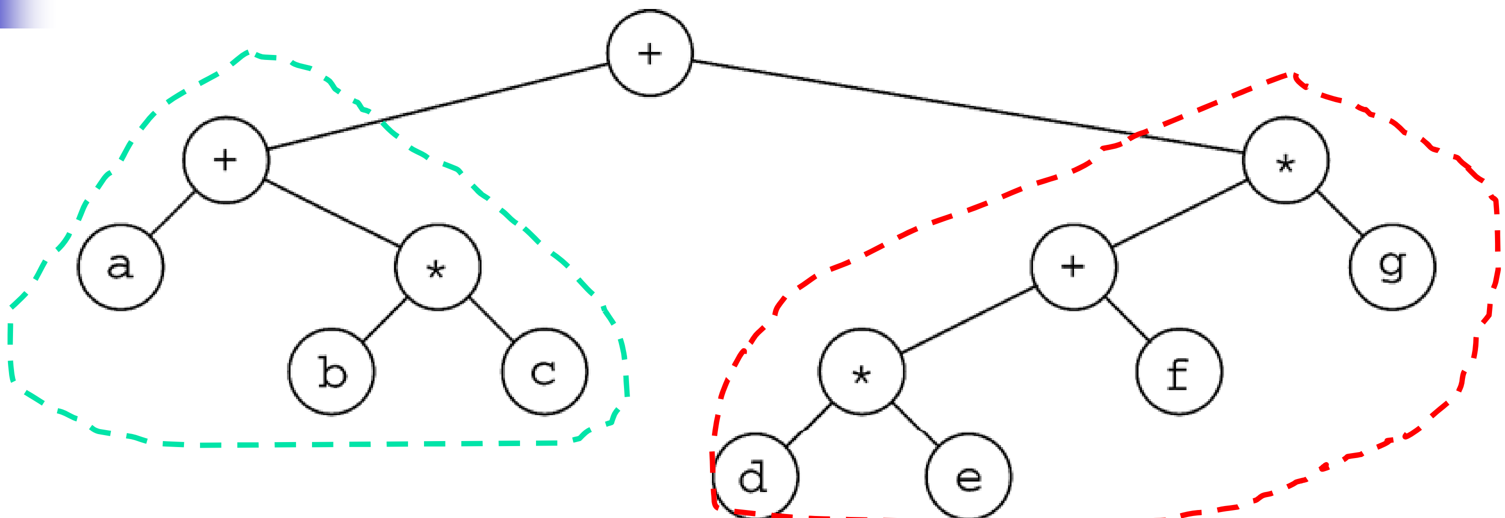
- Evaluate expression
  - Recursively evaluate left and right subtrees
  - Apply operator at root node to results from subtrees
- Traversals (recursive definitions)
  - Post-order: left, right, root
  - Pre-order: root, left, right
  - In-order: left, root, right

Traversals for tree rooted under an arbitrary "node"



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

Traversals



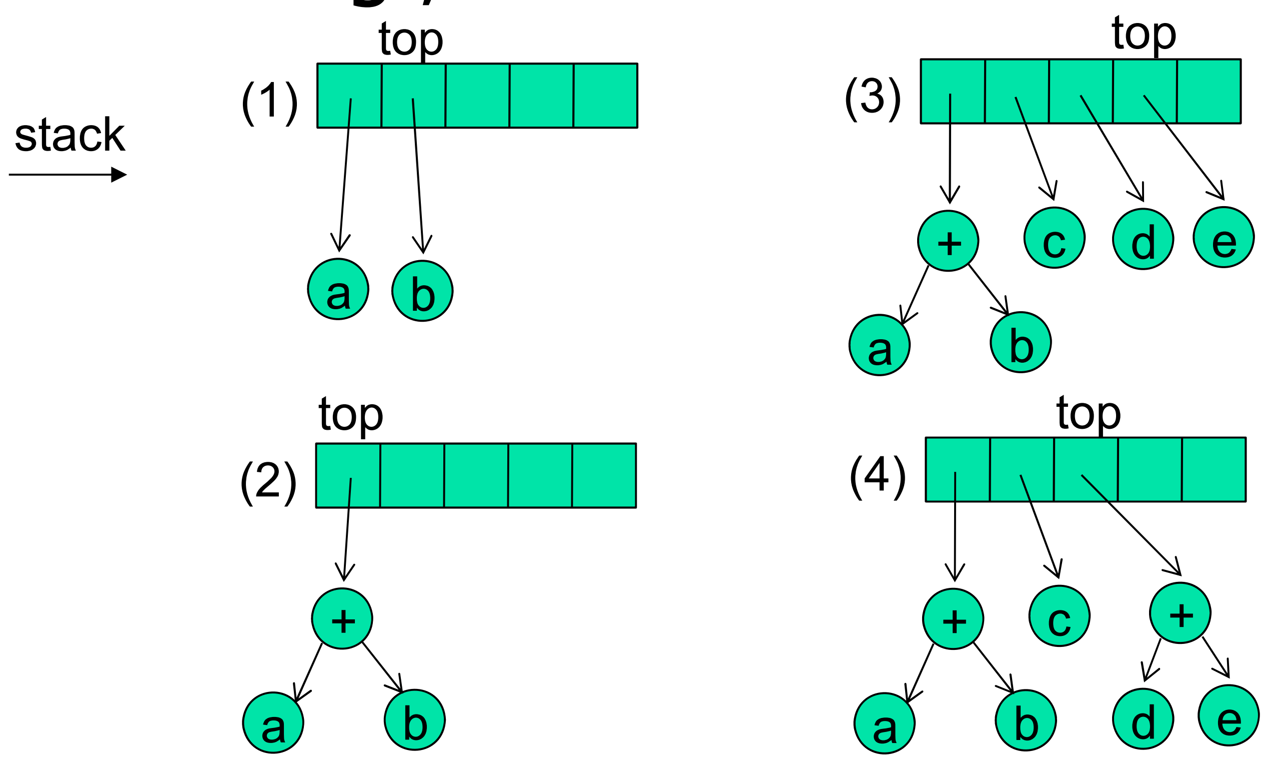
- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

Example: Expression Trees

- Constructing an expression tree from postfix notation
  - Use a stack of pointers to trees
  - Read postfix expression left to right
  - If operand, then push on stack
  - If operator, then:
    - Create a BinaryTreeNode with operator as the element
    - Pop top two items off stack
    - Insert these items as left and right child of new node
    - Push pointer to node on the stack

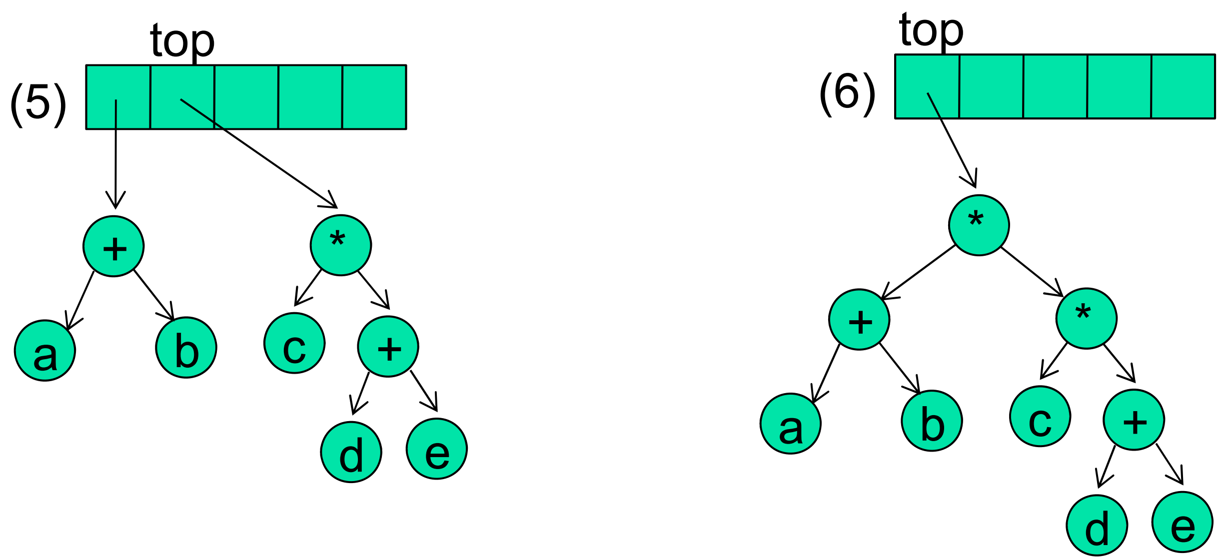
Example: Expression Trees

- E.g., a b + c d e + * *



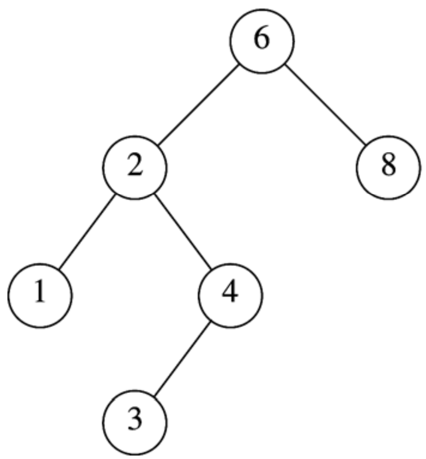
Example: Expression Trees

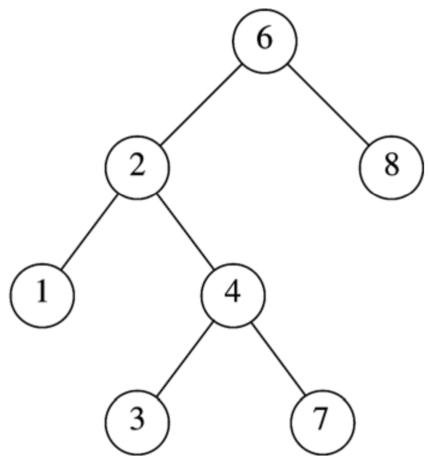
- E.g., a b + c d e + * *



Binary Search Trees

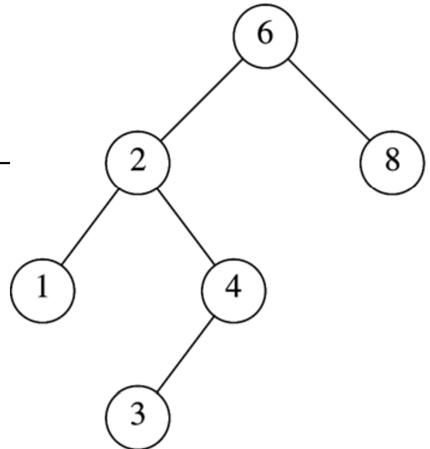
- "Binary search tree (BST)"
  - For any node n, items in left subtree of n
    ≤ item in node n
    ≤ items in right subtree of n





Which one is a BST and which one is not?

Searching in BSTs

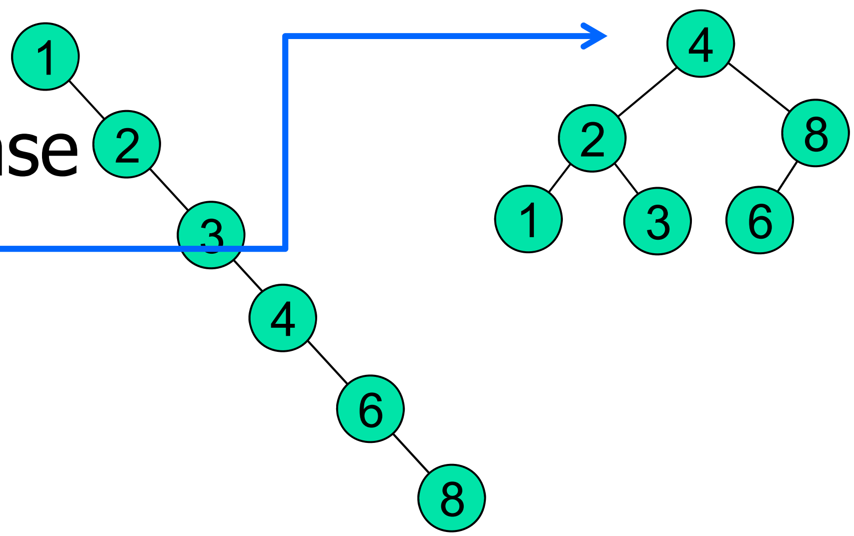



```
Contains (T, x)
{
  if (T == NULL)
  then return NULL
  if (T->element == x)
  then return T
  if (x < T->element)
  then return Contains (T->leftChild, x)
  else return Contains (T->rightChild, x)
}
```

Typically assume no duplicate elements.
If duplicates, then store counts in nodes, or
each node has a list of objects.

Searching in BSTs

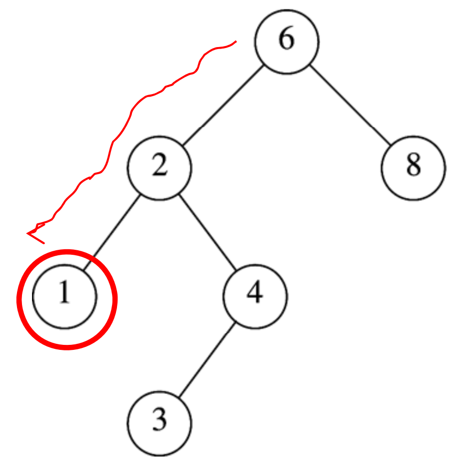
- Time to search using a BST with N nodes is O(?)
  - For a BST of height h, it is O(h)
  - And, h = O(N) worst-case
  - If the tree is balanced, then h=O(lg N)



Searching in BSTs

- Finding the minimum element
  - Smallest element in left subtree

```
findMin (T)
{
  if (T == NULL)
  then return NULL
  if (T->leftChild == NULL)
  then return T
  else return findMin (T->leftChild)
}
```

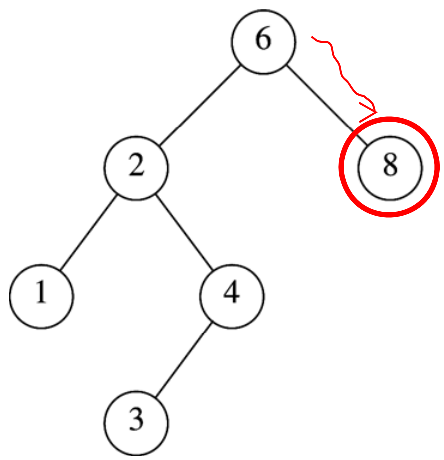


- Complexity ?   *O(h)*

Searching in BSTs

- Finding the maximum element
  - Largest element in right subtree

```
findMax (T)
{
  if (T == NULL)
  then return NULL
  if (T->rightChild == NULL)
  then return T
  else return findMax (T->rightChild)
}
```

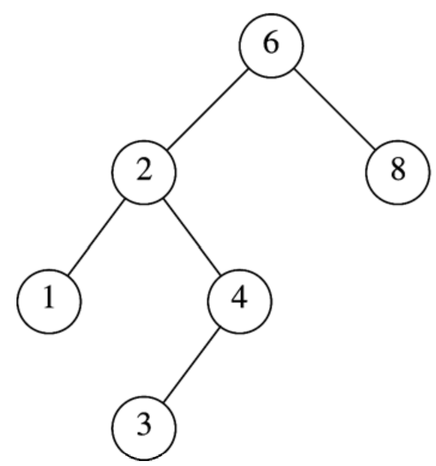


- Complexity ? *O(h)*

Printing BSTs

- In-order traversal ==> sorted

```
PrintTree (T)
{
  if (T == NULL)
  then return
  PrintTree (T->leftChild)
  cout << T->element
  PrintTree (T->rightChild)
}
```

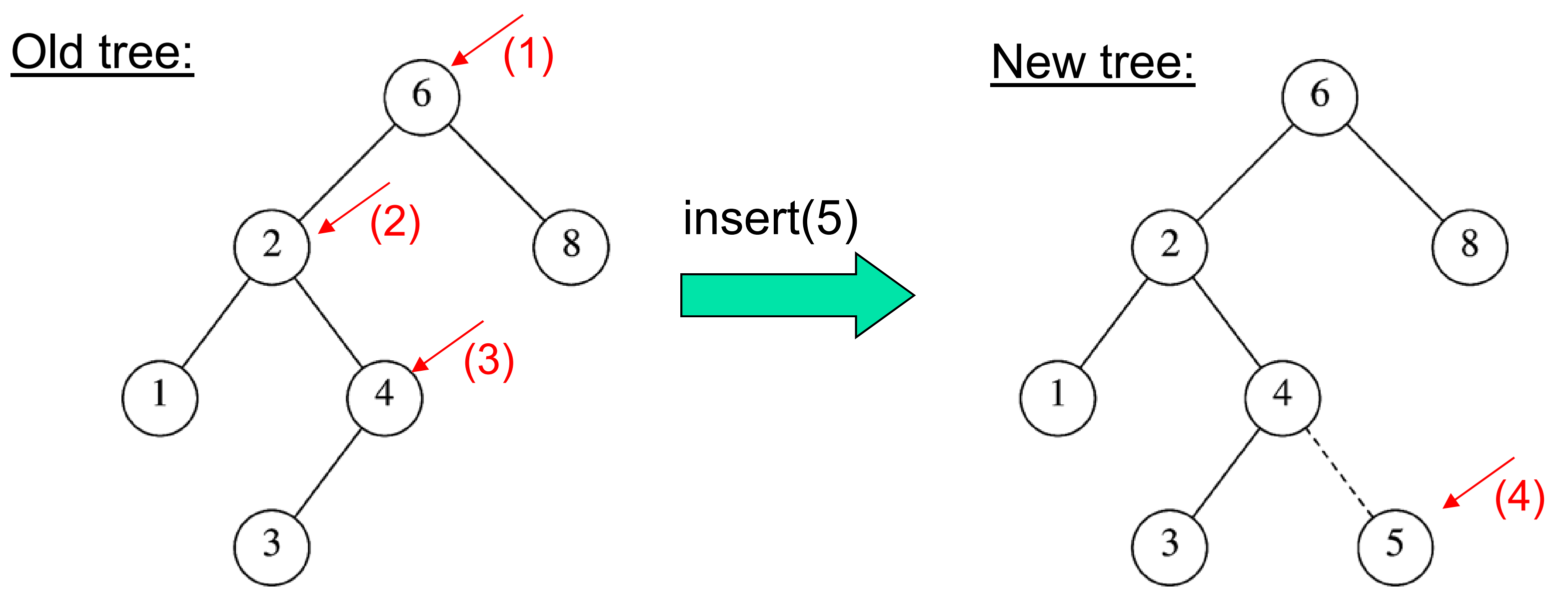


1 2 3 4 6 8

- Complexity? $\Theta(n)$

Inserting into BSTs

- E.g., insert 5



Inserting into BSTs

- "Search" for element until reach end of tree; insert new element there

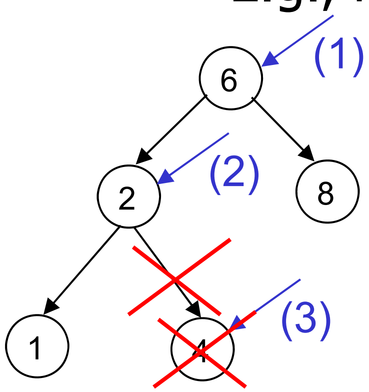
```
Insert (x, T)
{
  if (T == NULL)
  then T = new Node(x)
  else
       if (x < T->element)
       then if (T->leftChild == NULL)
              then T->leftChild = new Node(x)
              else Insert (x, T->leftChild)
       else if (T->rightChild == NULL)
              then (T->rightChild = new Node(x)
              else Insert (x, T->rightChild)
}
```

Complexity?

Removing from BSTs

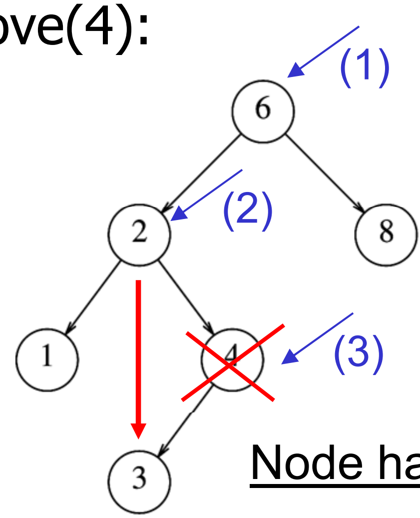
There are two cases for removal

- *Case 1: Node to remove has 0 or 1 child*
  - Action: Just remove it and make appropriate adjustments to retain BST structure
- E.g., remove(4):



Node has no children

remove(4):



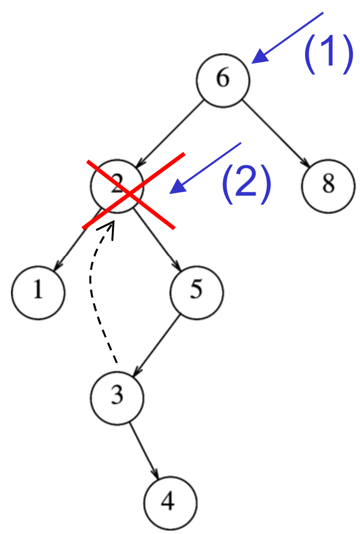
Node has 1 child

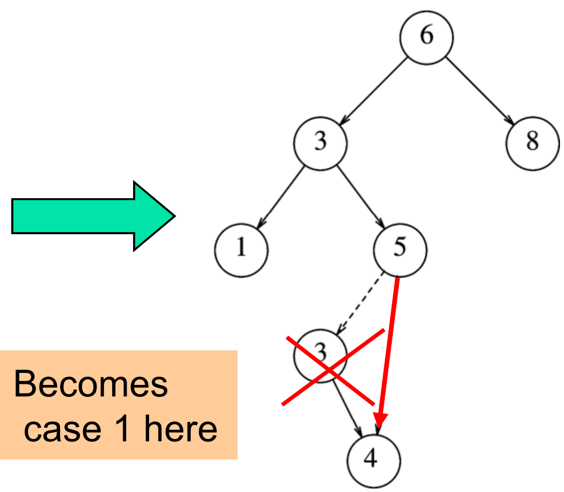
Removing from BSTs

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

Can the predecessor be used instead?

Old tree:





Becomes case 1 here

New tree:

Removing from BSTs

```
Remove (x, T)
{
  if (T == NULL)
  then return
  if (x == T->element)
  then if ((T->left == NULL) && (T->right != NULL))
       then T = T->right
       else if ((T->right == NULL) && (T->left != NULL))
            then T = T->left
        else if ((T->right == NULL) && (T->left == NULL))
            then T = NULL
       else {
                successor = findMin (T->right)
                T->element = successor->element
                Remove (T->element, T->right)
        }
  else if (x < T->element)
  then Remove (x, T->left)      // recursively search
  else Remove (x, T->right) // recursively search
}
```

CASE 1

CASE 2

Complexity?

Implementation of BST

```
template <typename Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    ~BinarySearchTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void remove( const Comparable & x );

    const BinarySearchTree & operator=( const BinarySearchTree & rhs );
```

```cpp
    private:
      struct BinaryNode
      {
         Comparable element;
         BinaryNode *left;
         BinaryNode *right;

         BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
           : element( theElement ), left( lt ), right( rt ) { }
      };

      BinaryNode *root;

      void insert( const Comparable & x, BinaryNode * & t ) const;
      void remove( const Comparable & x, BinaryNode * & t ) const;
      BinaryNode * findMin( BinaryNode *t ) const;
      BinaryNode * findMax( BinaryNode *t ) const;
      bool contains( const Comparable & x, BinaryNode *t ) const;
      void makeEmpty( BinaryNode * & t );
      void printTree( BinaryNode *t ) const;
      BinaryNode * clone( BinaryNode *t ) const;
};
```

What's the difference between a **`struct`** and a **`class`**?

const ?

Pointer to tree node passed by reference so it can be reassigned within function.

```
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```

Public member functions calling private recursive member functions.

```
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
```

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

```
/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t;
}
```

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        t = new BinaryNode( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ;  // Duplicate; do nothing
}
```

```cpp
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Case 2:
Copy successor data
Delete successor

Case 1: Just delete it

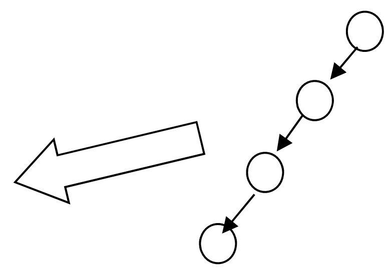
```
/**
 * Destructor for the tree
 */
~BinarySearchTree( )
{
    makeEmpty( );
}
/**
 * Internal method to make subtree empty.
 */
void makeEmpty( BinaryNode * & t )
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}
```

Post-order traversal

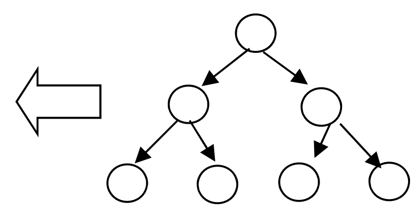
Can pre-order be used here?

BST Analysis

- **`printTree`**, **`makeEmpty`** and **`operator=`**
  - Always Θ(N)
- **`insert`**, **`remove`**, **`contains`**, **`findMin`**, **`findMax`**
  - O(h), where h = height of tree
- Worst case: h = ? **Θ(N)**



- Best case: h = ? **Θ(lg N)**
- Average case: h = ? **Θ(lg N)**



BST Average-Case Analysis

- Define *"Internal path length" of a tree:*
  = Sum of the depths of all nodes in the tree
  - Implies: average depth of a tree = Internal path length/N
- But there are lots of trees possible (one for every unique insertion sequence)
  - ==> Compute *average* internal path length over all possible insertion sequences
  - Assume all insertion sequences are equally likely
  - Result: $O(N \log_2 N)$
  - Thus, average depth = O(N lg N) / N = O(lg N)

HOW?

Calculating Avg. Internal Path Length

- Let D(N) = int. path. len. for a tree with N nodes

  = D(left) + D(right) + D(root)

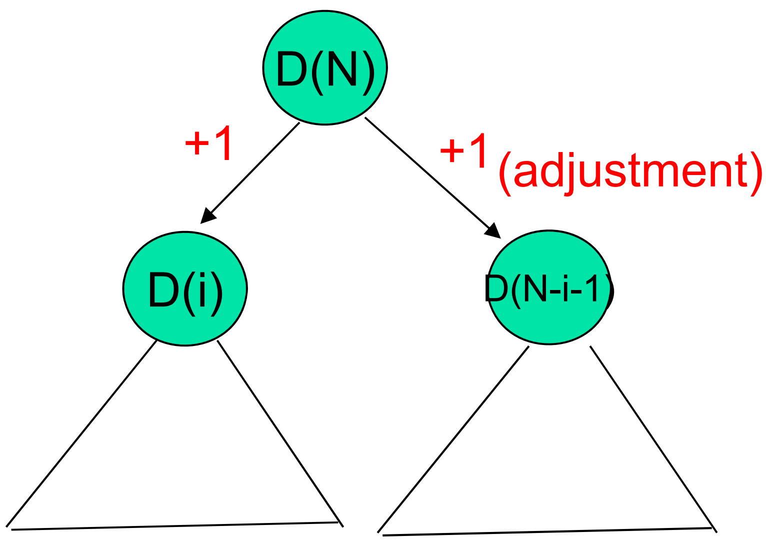
  = D(i) + i + D(N-i-1) + N-i-1 + 0

  = D(i) + D(N-i-1) + N-1

- If all tree sizes are equally likely,

  =>avg. D(i) = avg. D(N-i-1)

  $= 1/N \sum_{j=0}^{N-1} D(j)$



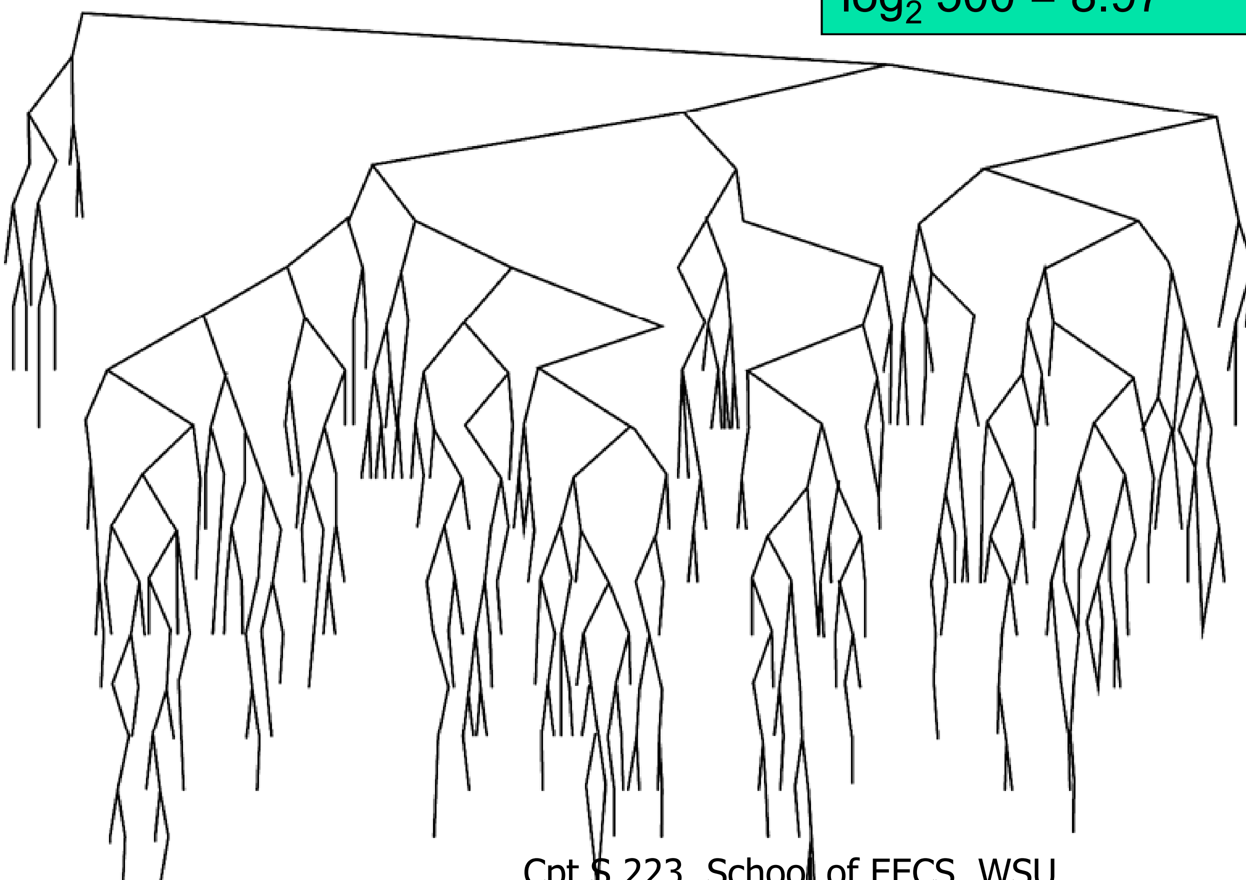
⇒ Avg. D(N) = $2/N \sum_{j=0}^{N-1} D(j)$ + N-1

⇒ **O(N lg N)**

A similar analysis will be used in QuickSort

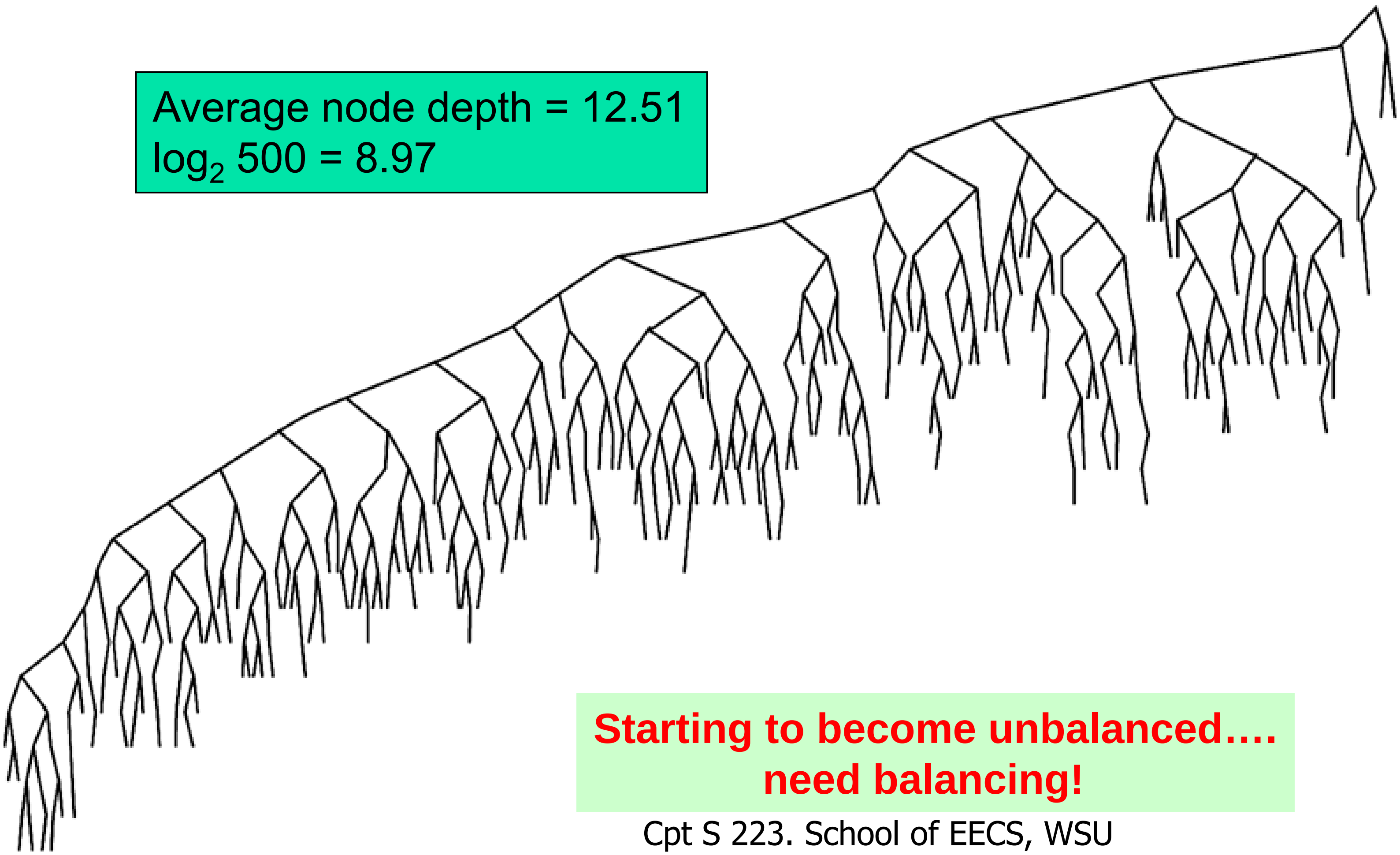
Randomly Generated 500-node BST (insert only)

Average node depth = 9.98
$\log_2 500 = 8.97$



Previous BST after 500^2 Random Mixture of Insert/Remove Operations

Average node depth = 12.51
$\log_2 500 = 8.97$



Starting to become unbalanced…. need balancing!

Balanced Binary Search Trees

BST Average-Case Analysis

- After randomly inserting N nodes into an empty BST
  - Average depth = $O(\log_2 N)$
- After $\Theta(N^2)$ random insert/remove pairs into an N-node BST
  - Average depth = $\Theta(N^{1/2})$
- Why?
- Solutions?
  - Overcome problematic average cases?
  - Overcome worst case?

Balanced BSTs

- AVL 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 $O(\log_2 N)$

AVL Trees

- AVL (Adelson-Velskii and Landis, 1962)

- <u>Definition:</u>

Every AVL tree is a BST such that:

1. For *every* node in the BST, the heights of its left and right subtrees differ by at most 1

AVL Trees

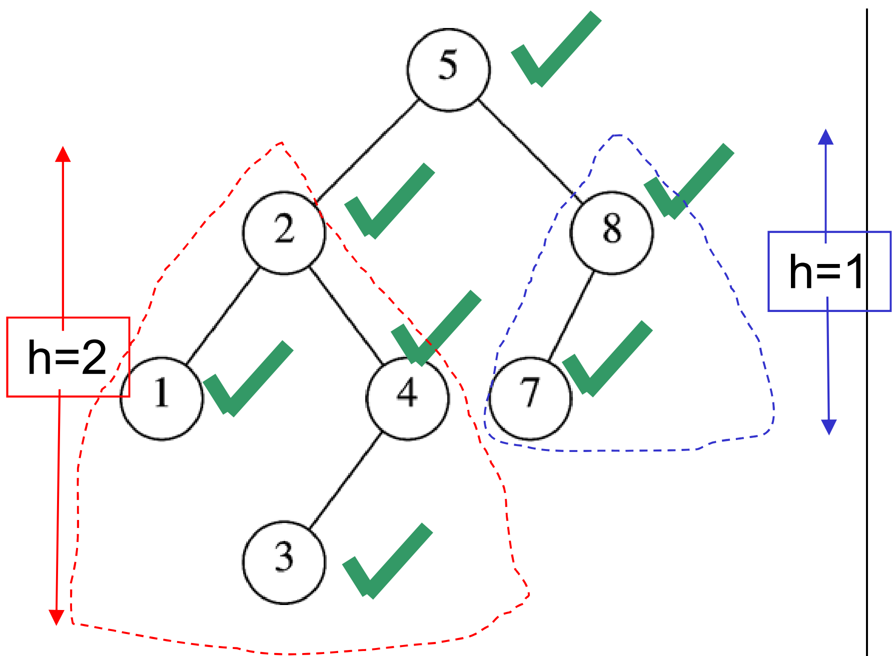
- Worst-case Height of AVL tree is $\Theta(\log_2 N)$
  - Actually, $1.44 \log_2(N+2) - 1.328$

- Intuitively, enforces that a tree is "sufficiently" populated before height is grown
  - Minimum #nodes S(h) in an AVL tree of height h :
    - $S(h) = S(h-1) + S(h-2) + 1$
      - (Similar to Fibonacci recurrence)
      - $= \Theta(2^h)$

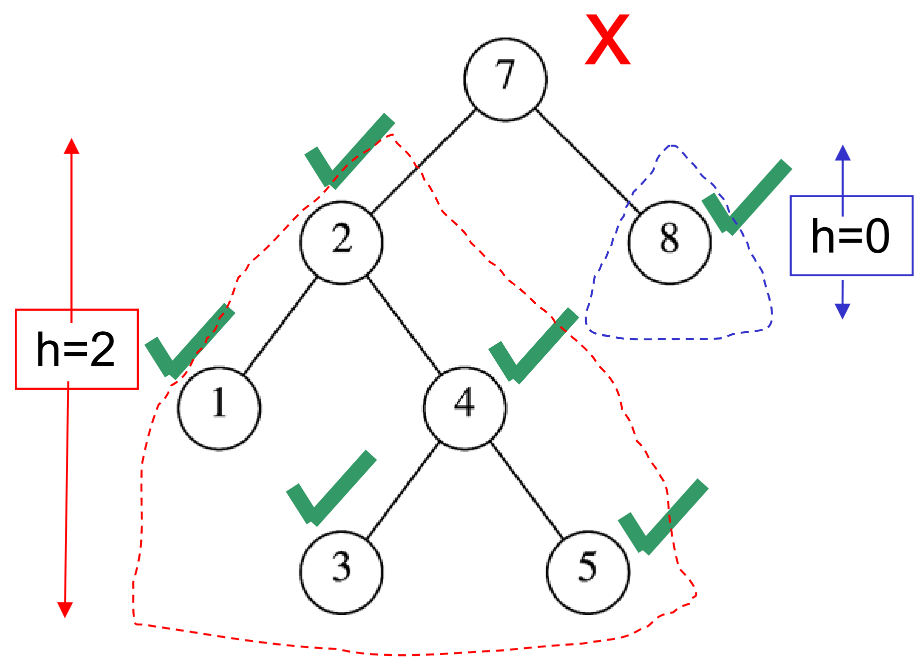
AVL Trees

<u>Note:</u> height violation not allowed at ANY node

<u>Which of these is a valid AVL tree?</u>



This is an AVL tree



This is NOT an AVL tree

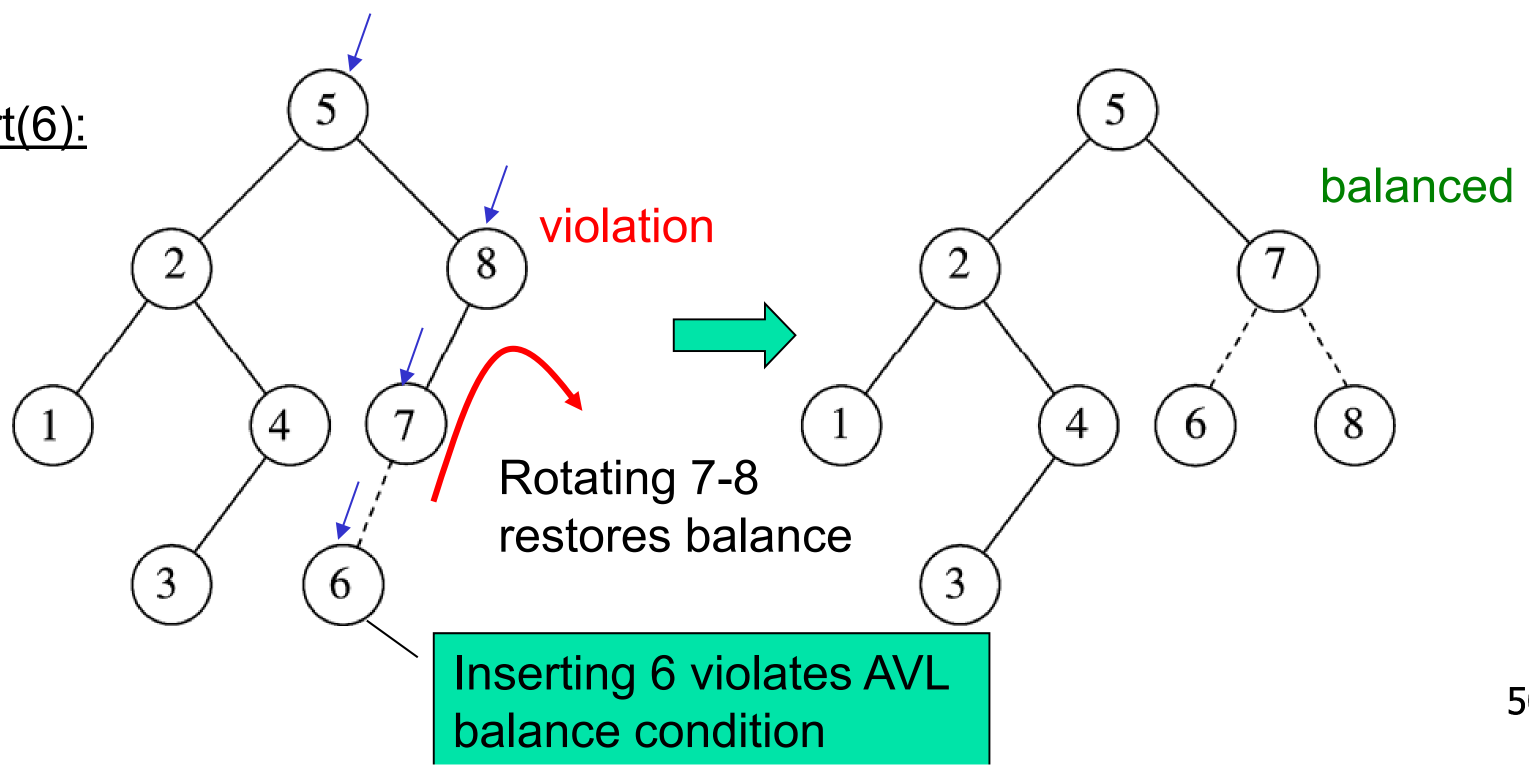
Maintaining Balance Condition

- If we can maintain balance condition, then the insert, remove, find operations are O(lg N)
  - How?
    - $N = \Omega(2^h) \quad => \quad h = O(\lg(N))$
- Maintain height h(t) at each node t
  - h(t) = max {h(t->left), h(t->right)} + 1
  - h(empty tree) = -1
- Which operations can upset balance condition?

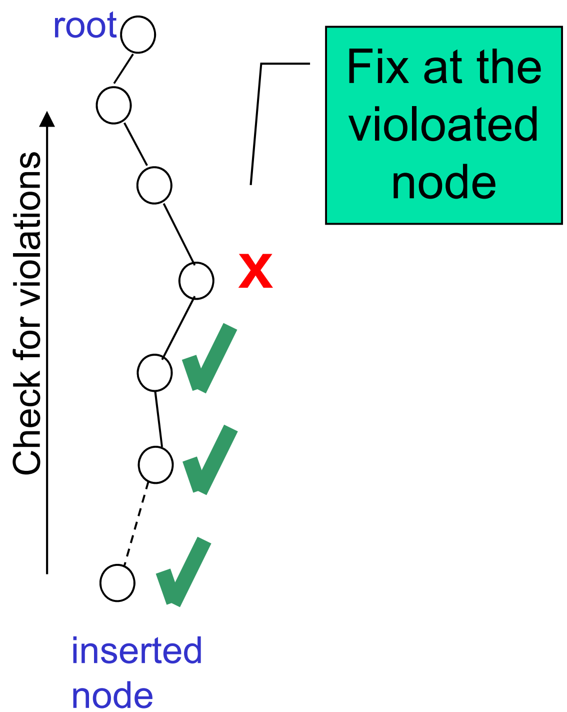
AVL Insert

- Insert can violate AVL balance condition
- Can be fixed by a <u>rotation</u>

<u>Insert(6):</u>



AVL Insert



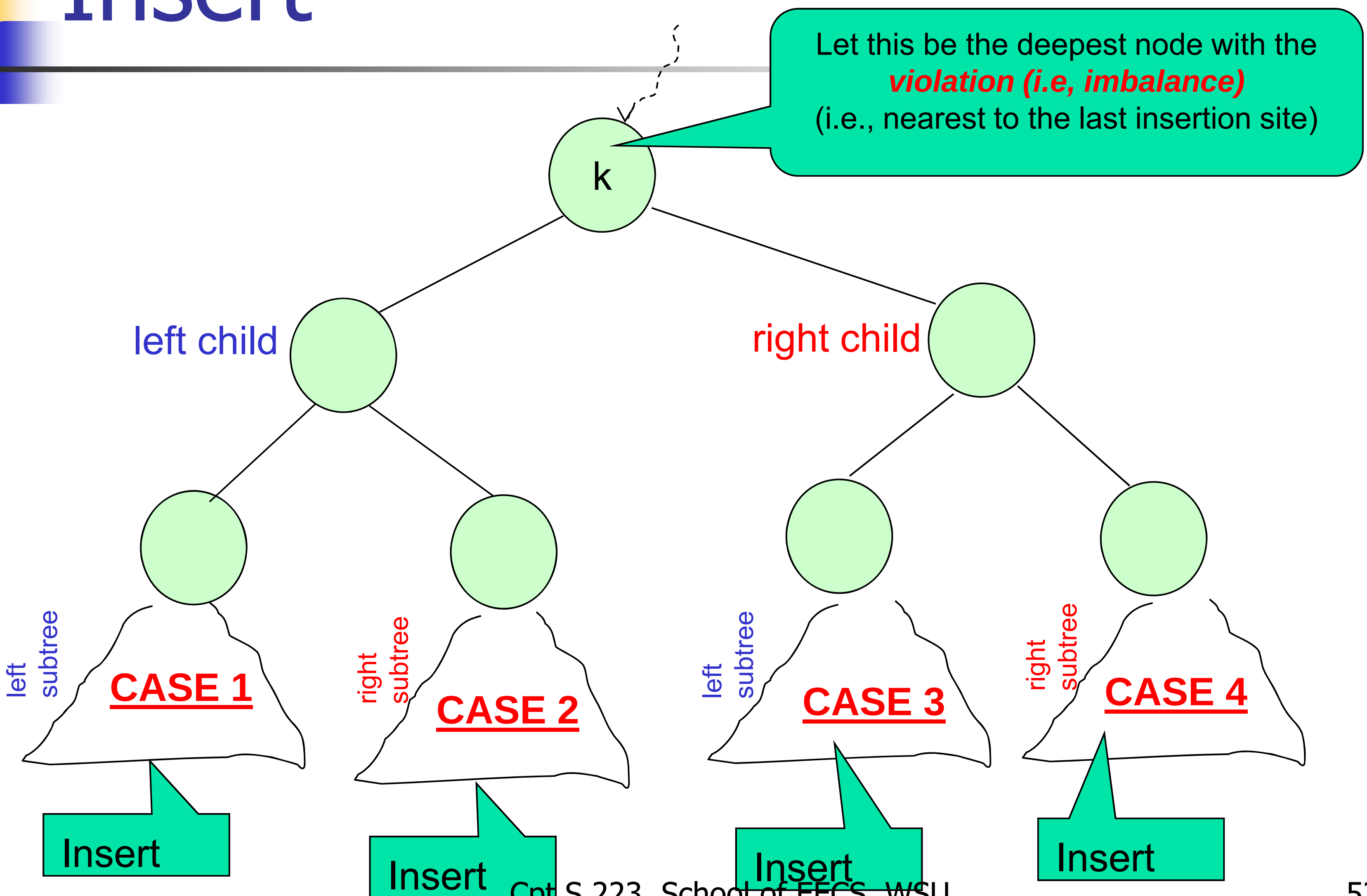
- Only nodes along path to insertion could have their balance altered
- Follow the path back to root, looking for violations
- Fix the deepest node with violation using single or double rotations

Q) Why is fixing the deepest node with violation sufficient?

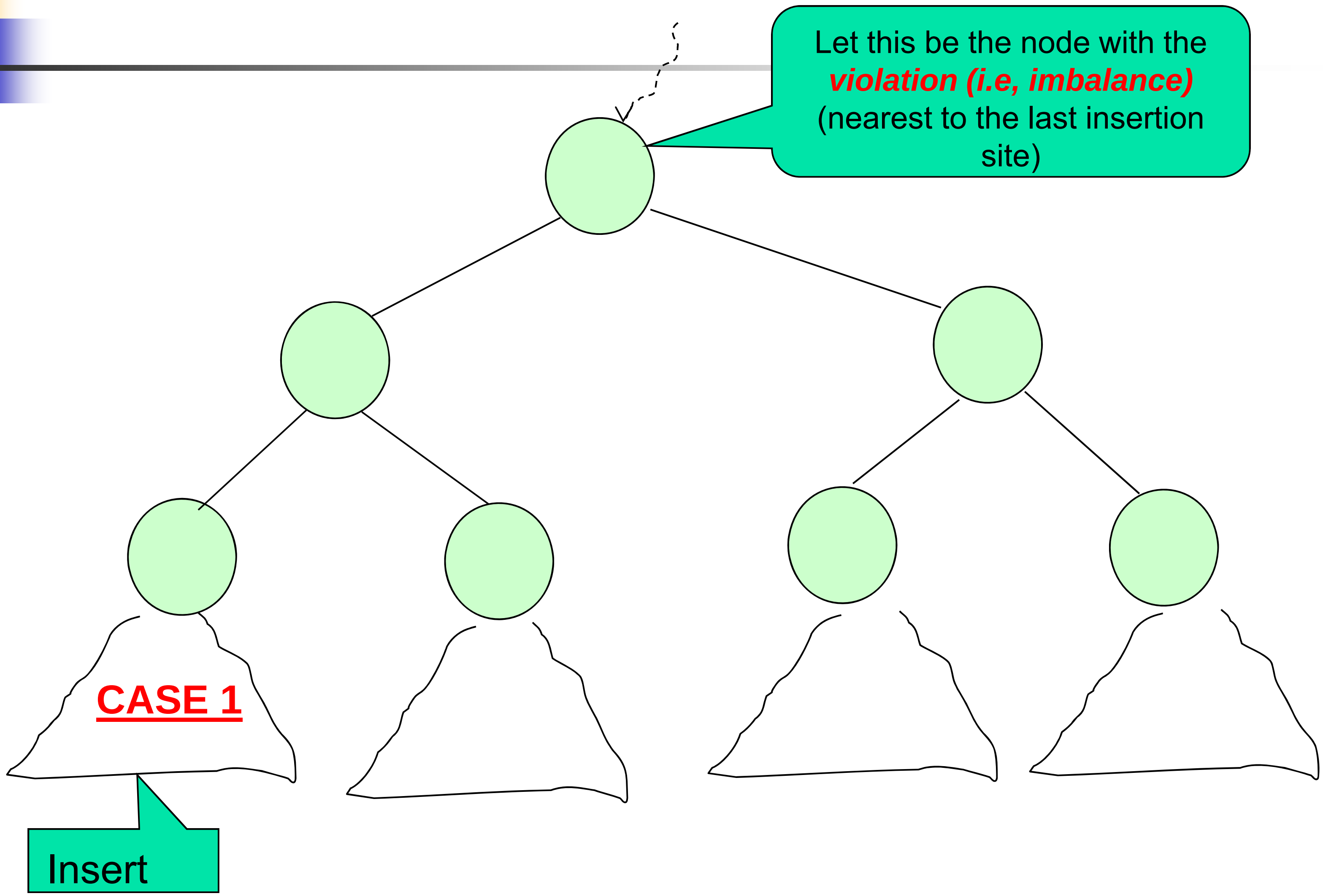
AVL Insert – how to fix a node with height violation?

- Assume the violation after insert is at node k
- Four cases leading to violation:
  - CASE 1: Insert into the left subtree of the left child of k
  - CASE 2: Insert into the right subtree of the left child of k
  - CASE 3: Insert into the left subtree of the right child of k
  - CASE 4: Insert into the right subtree of the right child of k
- Cases 1 and 4 handled by "single rotation"
- Cases 2 and 3 handled by "double rotation"

Identifying Cases for AVL Insert



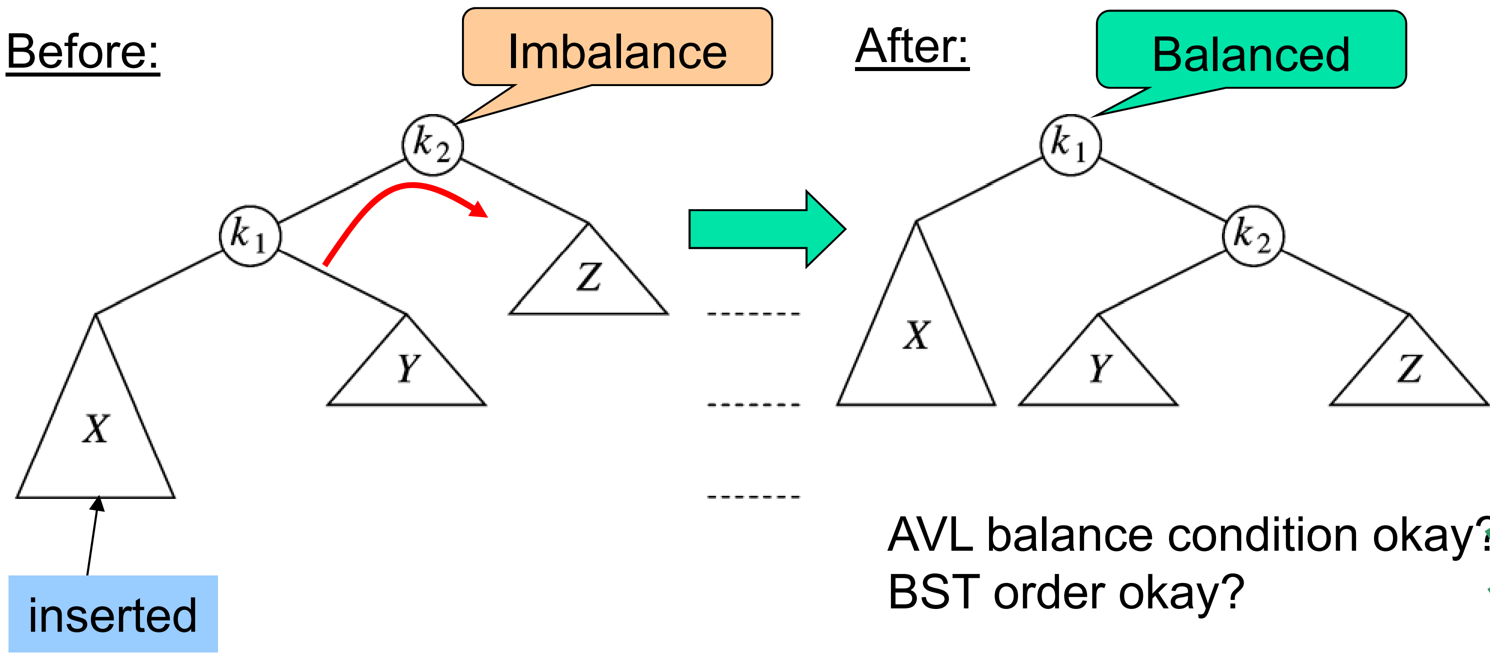
Case 1 for AVL insert



Remember: X, Y, Z could be empty trees, or single node trees, or mulltiple node trees.

AVL Insert (single rotation)

- Case 1: Single rotation right

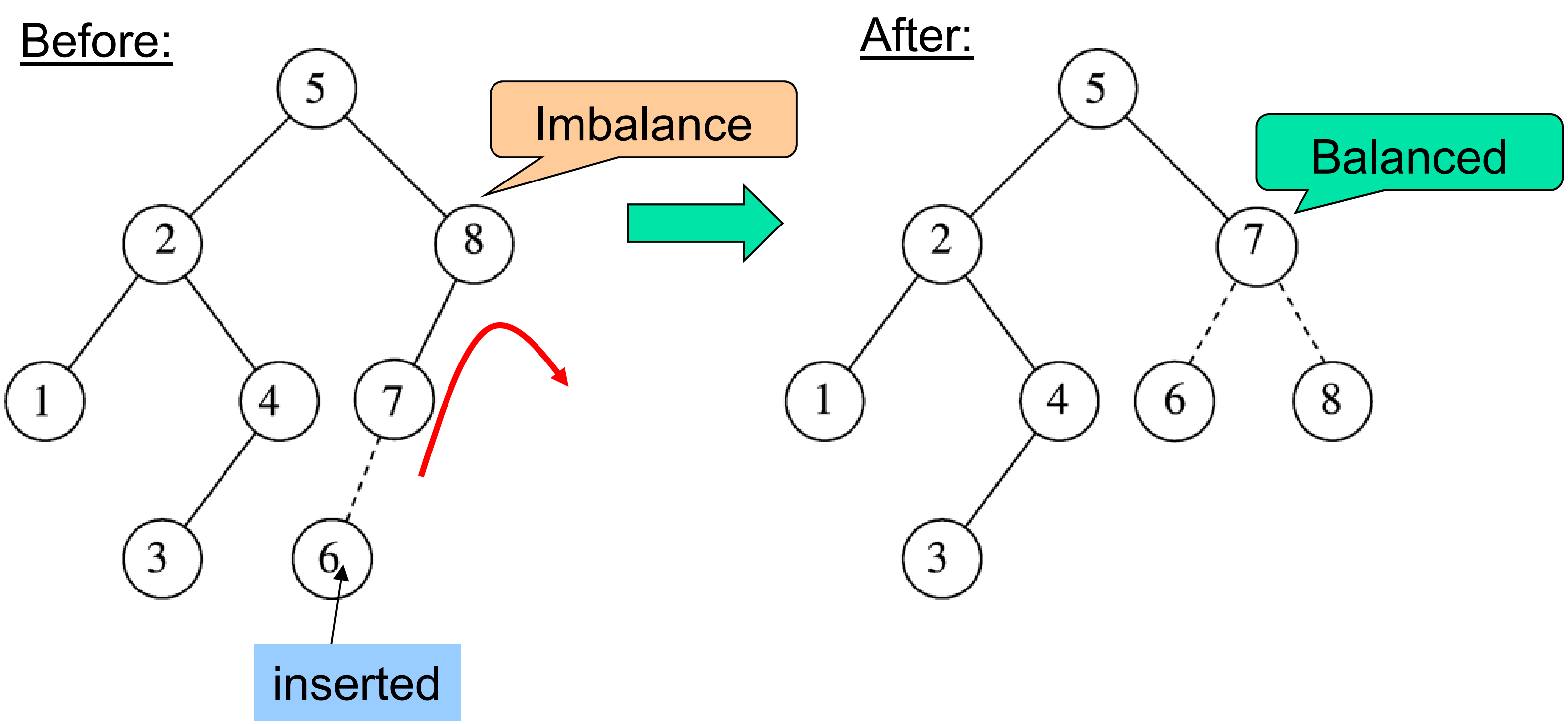


AVL balance condition okay? ✓
BST order okay? ✓

Invariant: $X < k_1 < Y < k_2 < Z$

AVL Insert (single rotation)

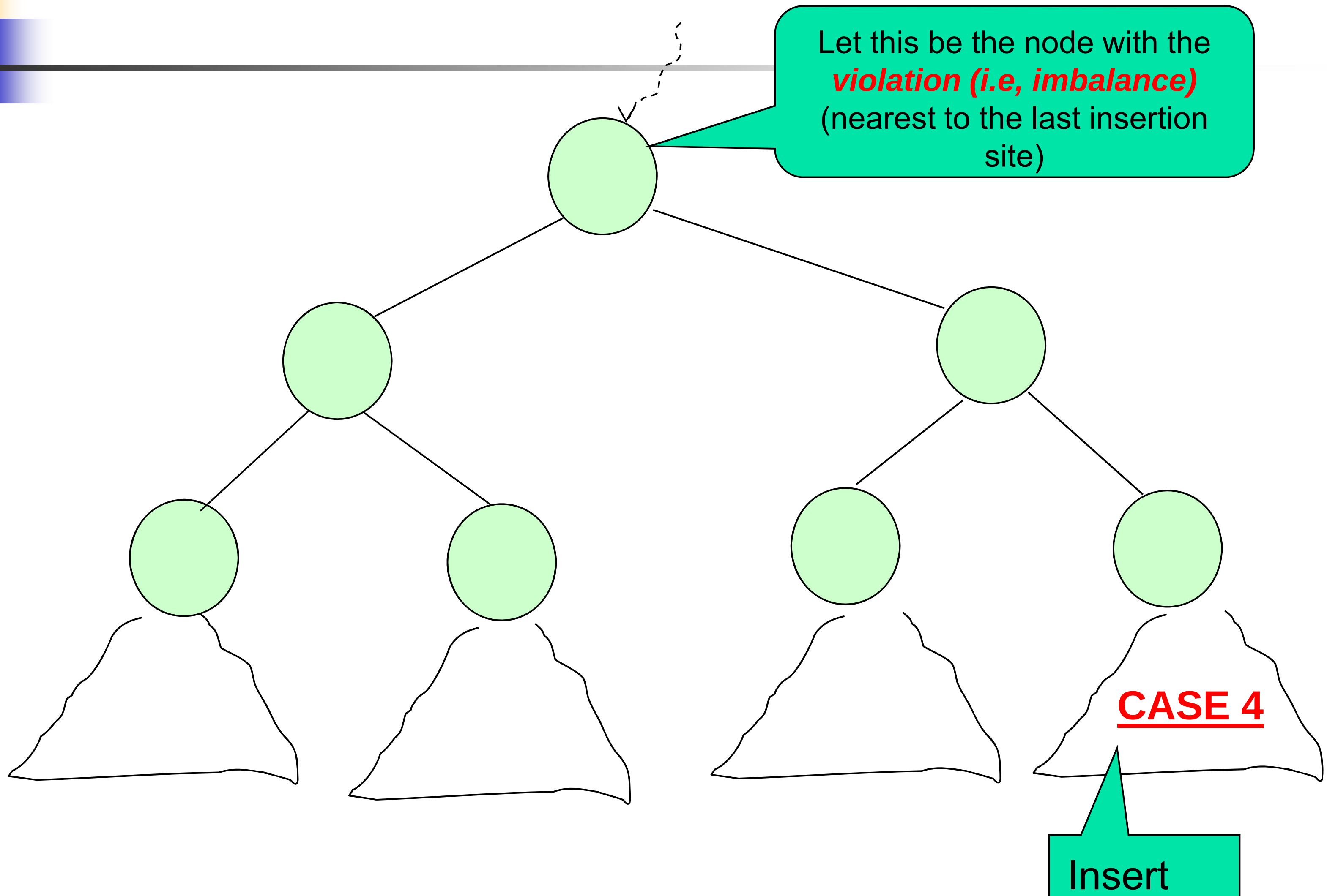
- Case 1 example



General approach for fixing violations after AVL tree insertions

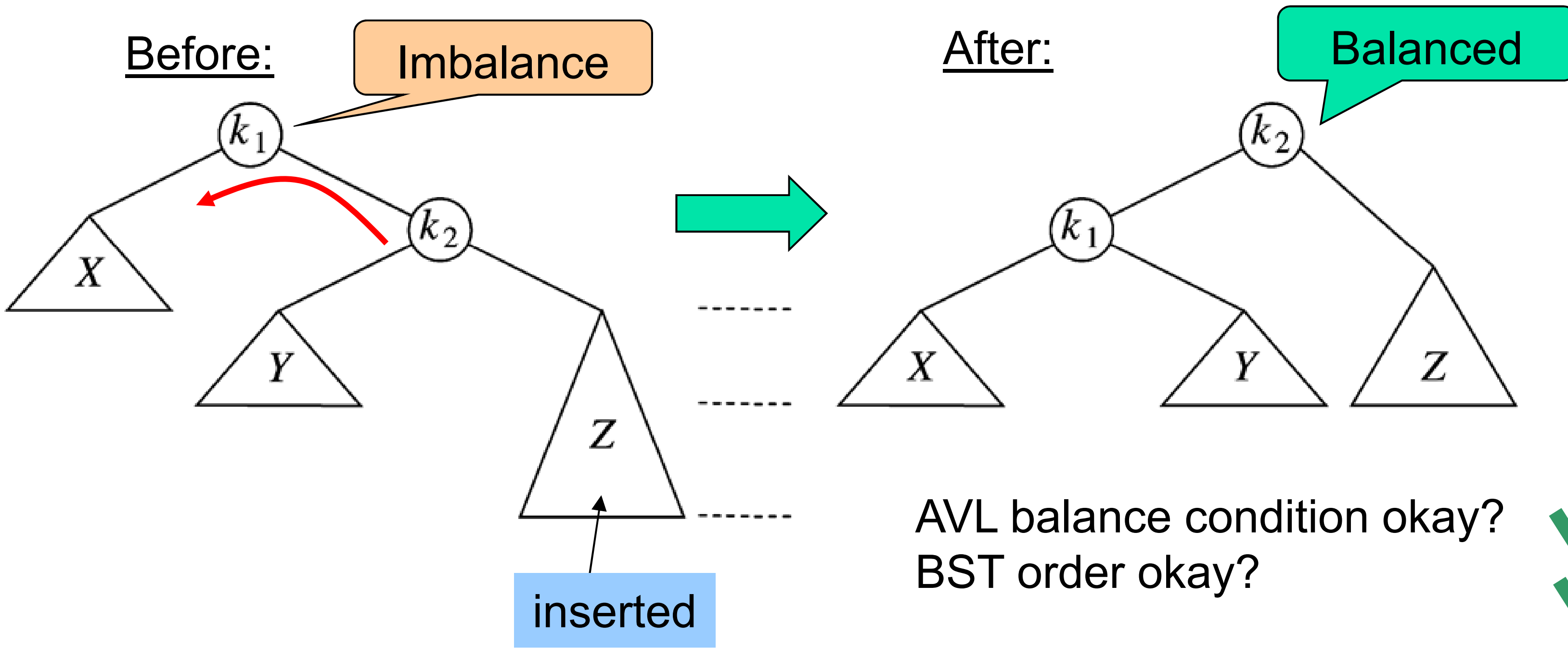
1. Locate the deepest node with the height imbalance
2. Locate which part of its subtree caused the imbalance
   - This will be same as locating the subtree site of insertion
3. Identify the case (1 or 2 or 3 or 4)
4. Do the corresponding rotation.

Case 4 for AVL insert



AVL Insert (single rotation)

- Case 4: Single rotation left

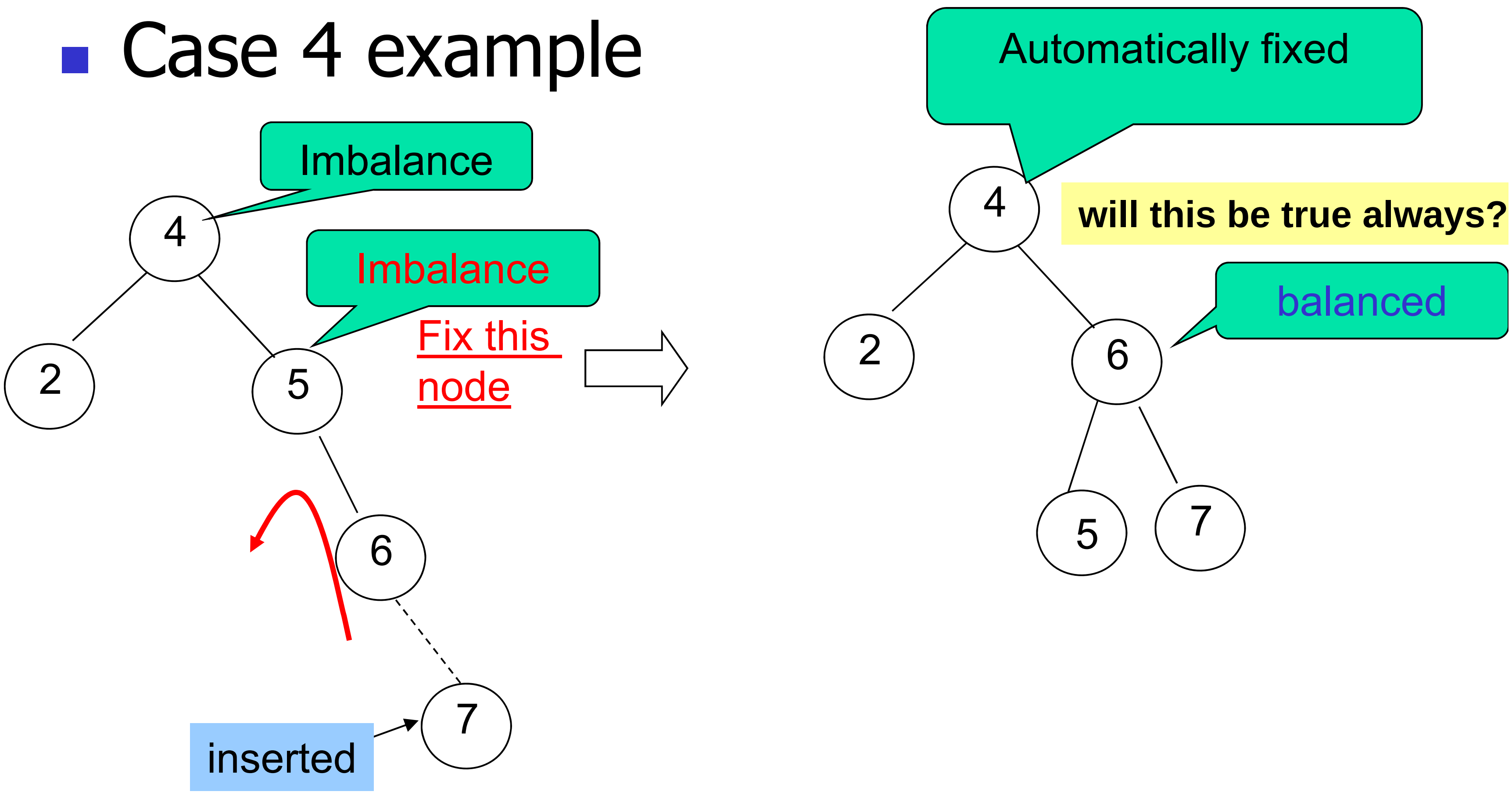


AVL balance condition okay?
BST order okay?

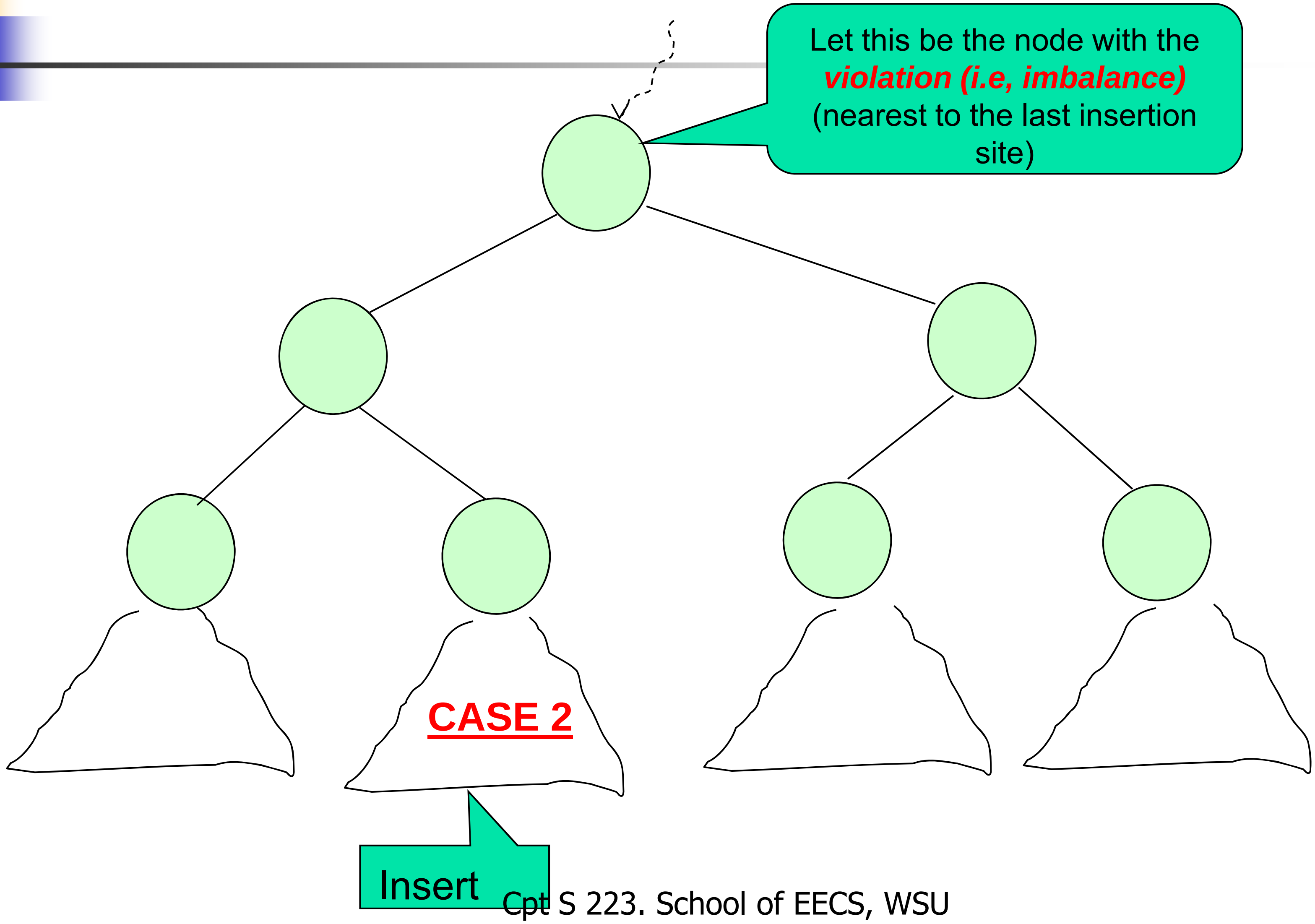
Invariant: $X < k_1 < Y < k_2 < Z$

AVL Insert (single rotation)

- Case 4 example



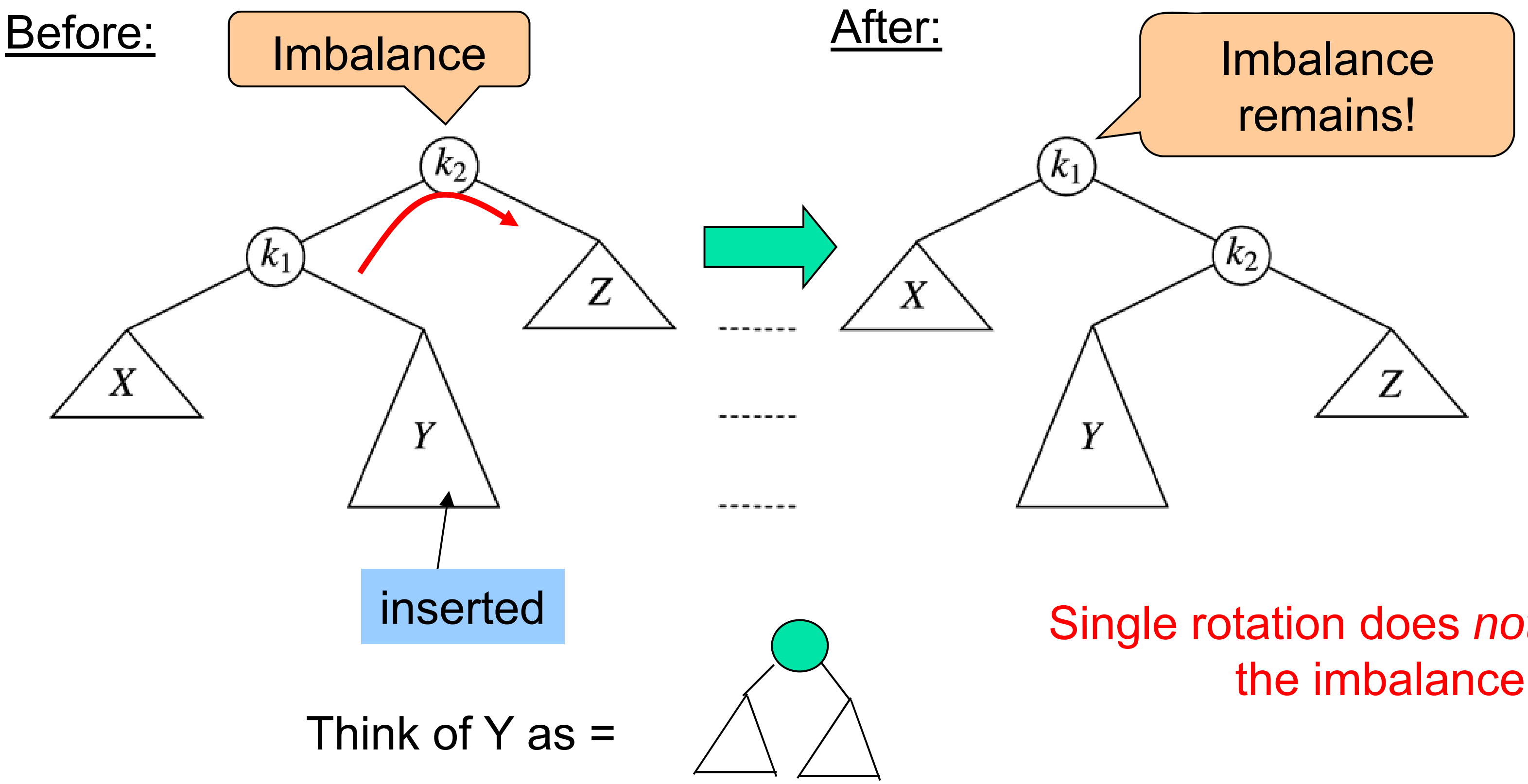
Case 2 for AVL insert



Note: X, Z can be empty trees, or single node trees, or mulltiple node trees
But Y should have at least one or more nodes in it because of insertion.

AVL Insert

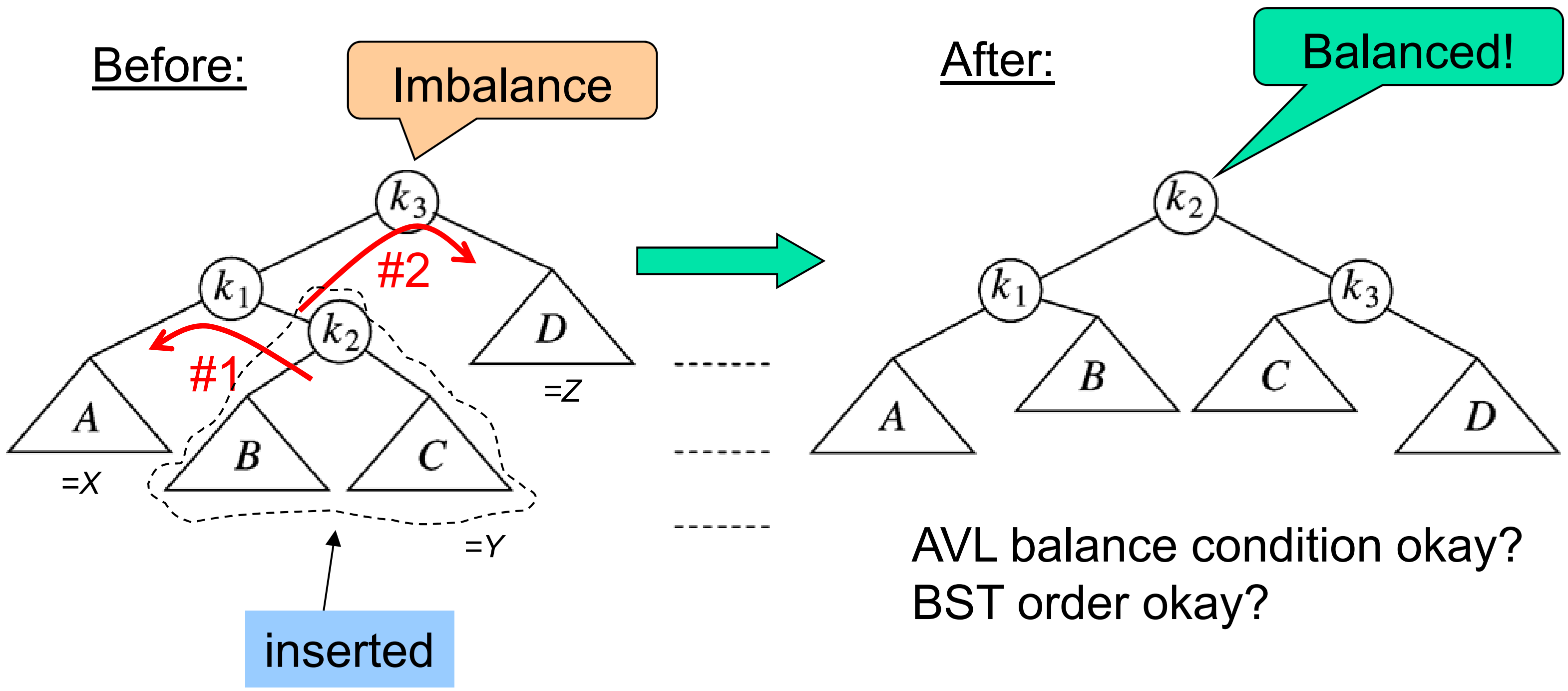
- Case 2: Single rotation fails



Single rotation does *not* fix the imbalance!

AVL Insert

- <u>Case 2:</u> Left-right double rotation



AVL balance condition okay? ✓
BST order okay? ✓

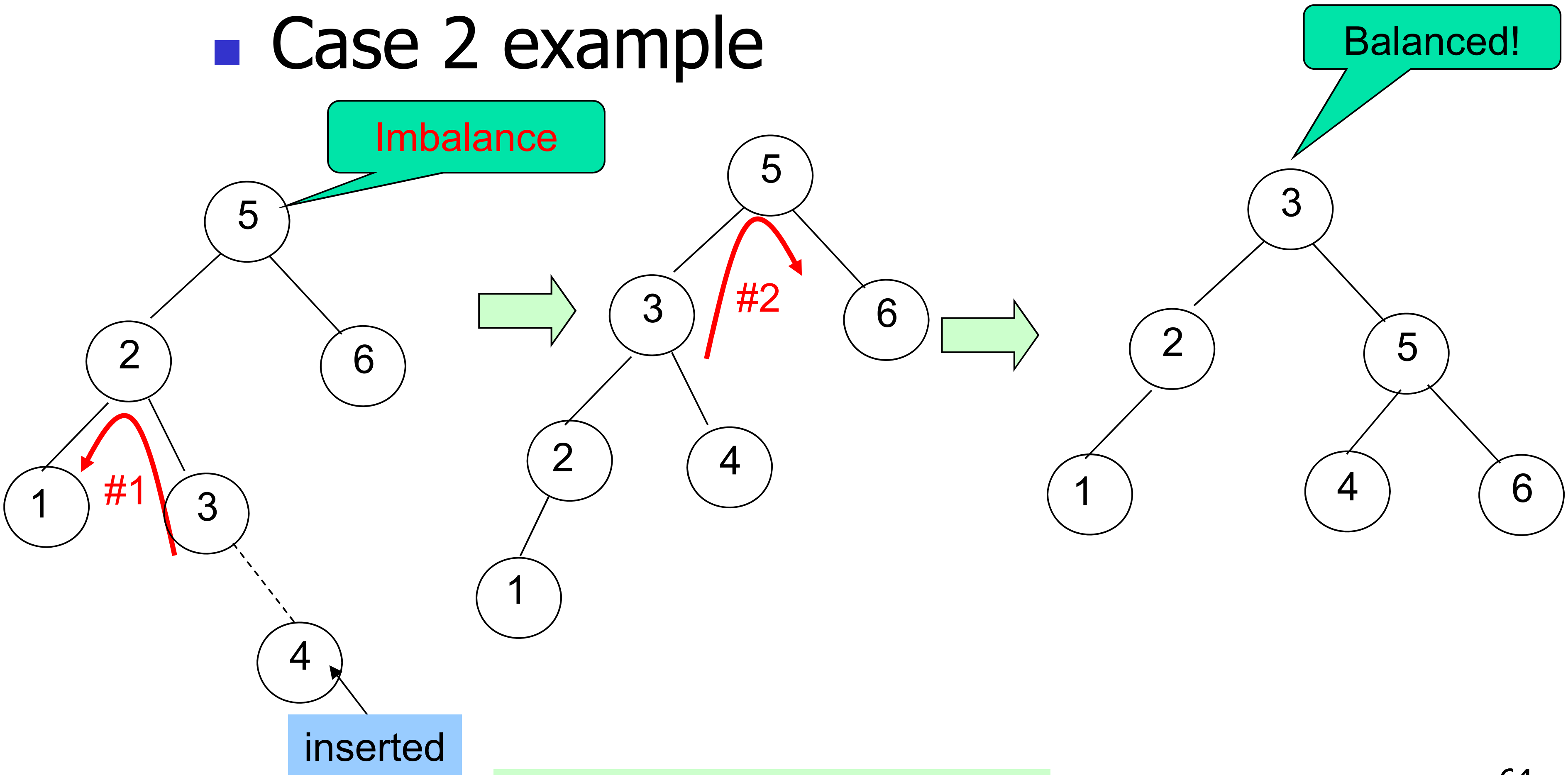
<u>Invariant:</u> A < k_1 < B < k_2 < C < k_3 < D

Can be implemented as
two successive single rotations

=> Make k_2 take k_3's place

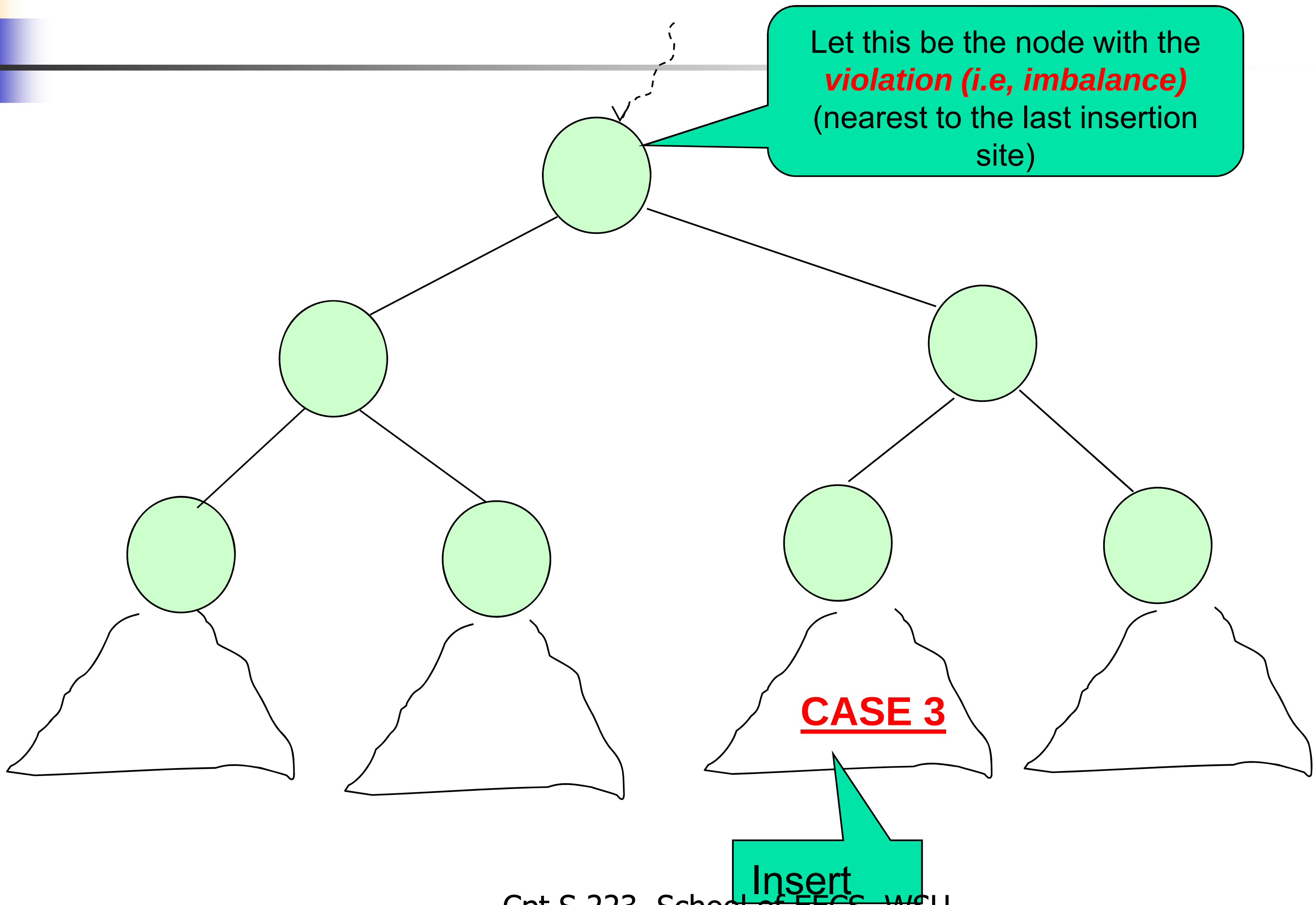
AVL Insert (double rotation)

- Case 2 example



Approach: push 3 to 5's place

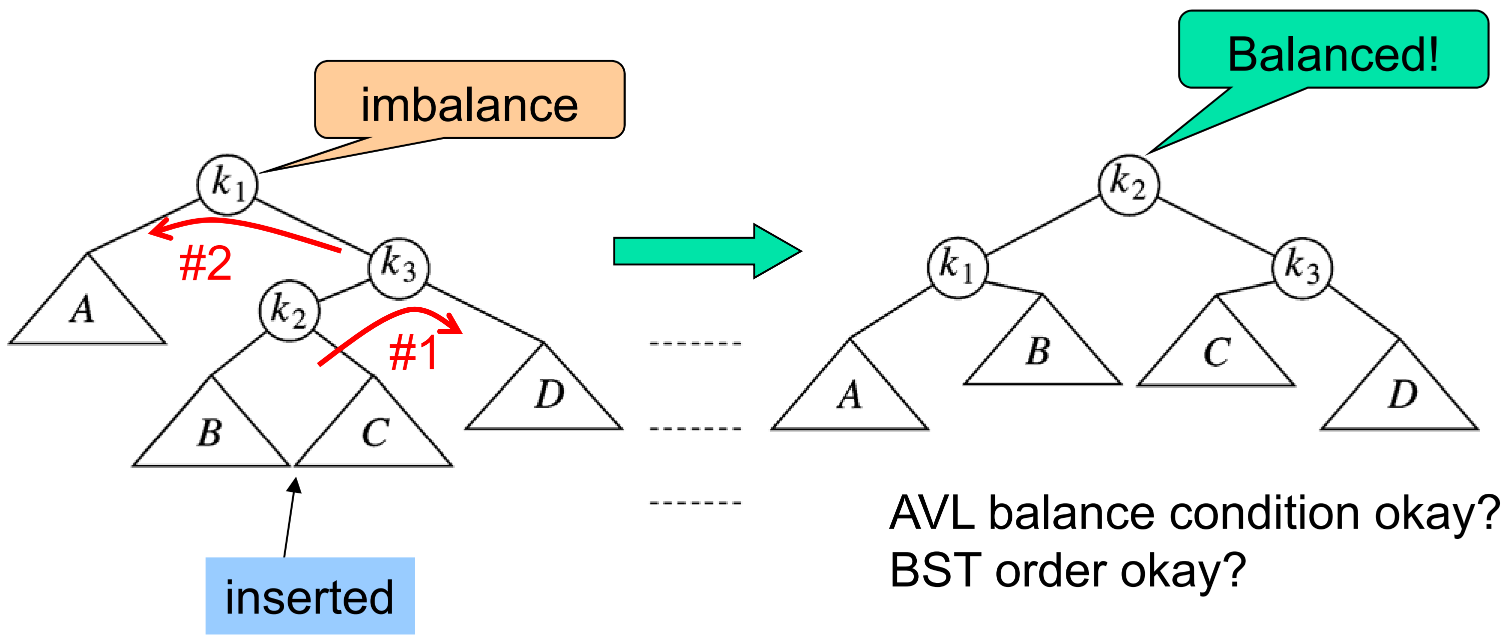
Case 3 for AVL insert



Case 3 == mirror case of Case 2

AVL Insert

- <u>Case 3:</u> Right-left double rotation

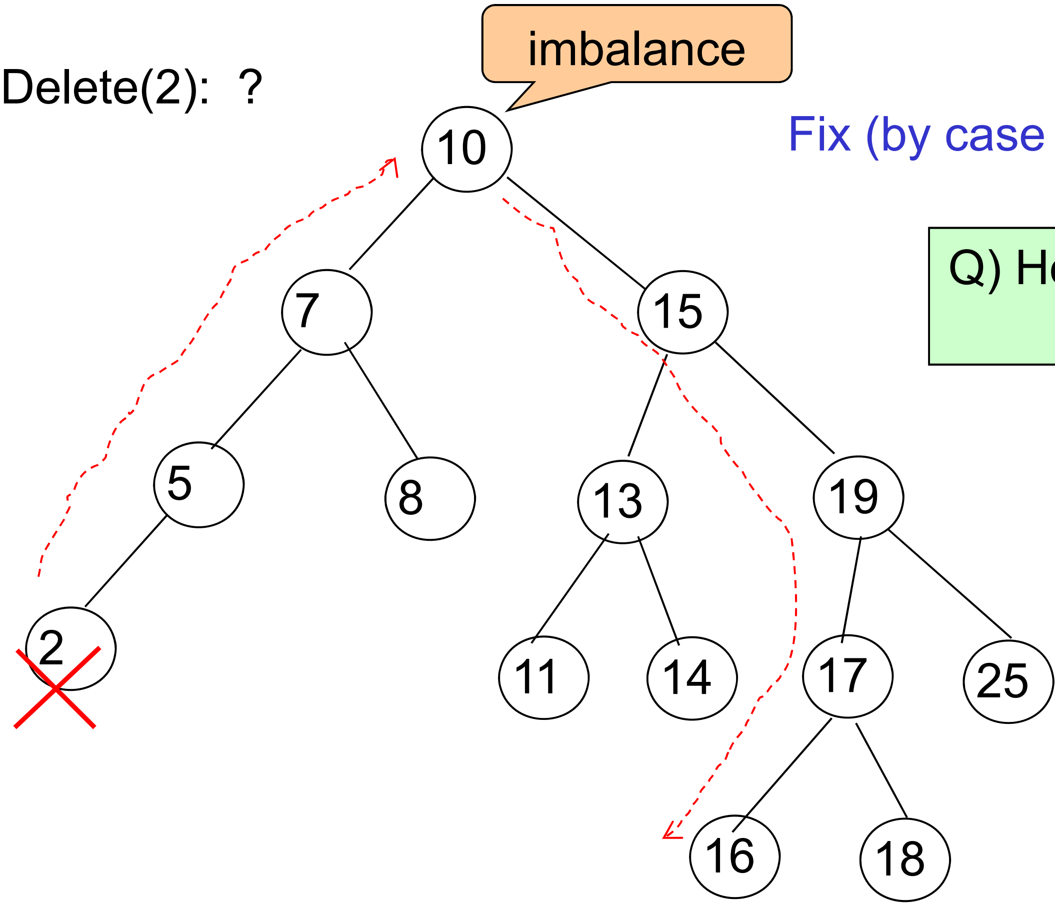


AVL balance condition okay?
BST order okay?

<u>Invariant:</u> A < k_1 < B < k_2 < C < k_3 < D

Exercise for AVL deletion/remove

Delete(2): ?



Fix (by case 4)

Q) How much time will it take to identify the case?

Alternative for AVL Remove (Lazy deletion)

- Assume **`remove`** accomplished using <u>lazy deletion</u>
  - Removed nodes only marked as deleted, but not actually removed from BST until some cutoff is reached
  - Unmarked when same object re-inserted
    - Re-allocation time avoided
  - Does not affect $O(\log_2 N)$ height as long as deleted nodes are not in the majority
  - Does require additional memory per node
- Can accomplish **`remove`** without lazy deletion

AVL Tree Implementation

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & theElement, AvlNode *lt,
                                            AvlNode *rt, int h = 0 )
      : element( theElement ), left( lt ), right( rt ), height( h )
};
```

AVL Tree Implementation

```
/**
 * Return the height of node t or -1 if NULL.
 */
int height( AvlNode *t ) const
{
    return t == NULL ? -1 : t->height;
}
```

Q) Is it guaranteed that the ***deepest*** *node with imbalance is the one that gets fixed?*

A) Yes, recursion will ensure that.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );
    }
    else
        ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Insert first, and then fix

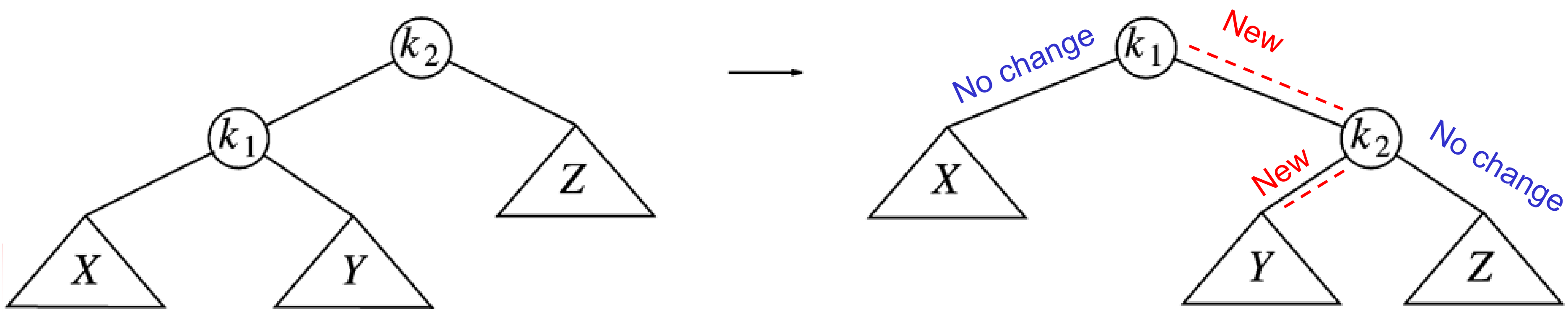
Locate insertion site relative to the imbalanced node (if any)

Case 1

Case 2

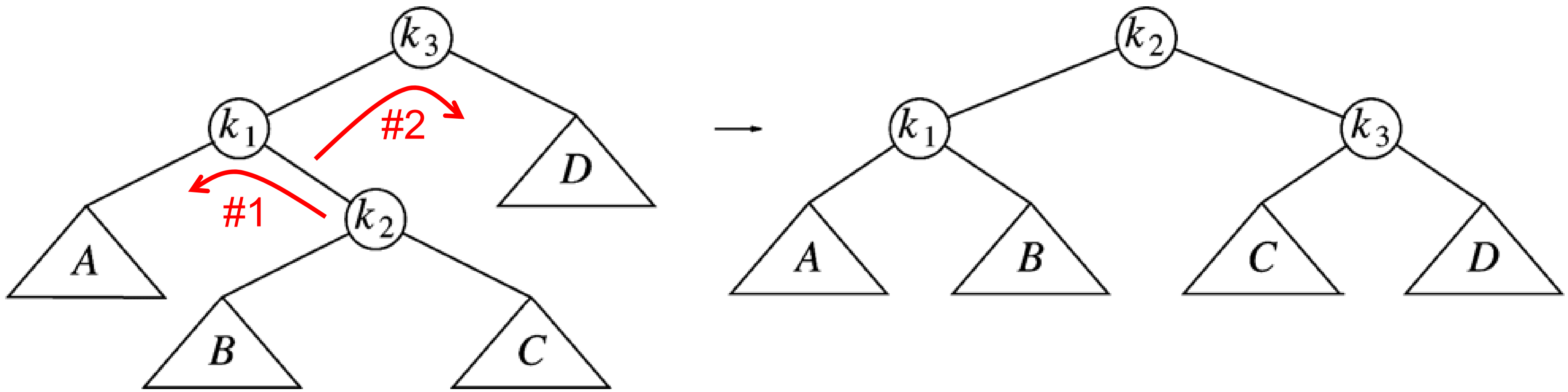
Case 4

Case 3




```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```

Similarly, write *rotateWithRightChild()* for case 4




```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );   // #1
    rotateWithLeftChild( k3 );          // #2
}
```

Splay Tree

Observation:

- Height imbalance is a problem only if & when the nodes in the deeper parts of the tree are accessed

Idea:

- Use a **lazy** strategy to fix height imbalance

Strategy:

- After a node is accessed, push it to the root via AVL rotations
- Guarantees that any M consecutive operations on an empty tree will take at most $O(M \log_2 N)$ time
- Amortized cost per operation is $O(\log_2 N)$
- Still, some operations may take O(N) time
- Does not require maintaining height or balance information

Splay Tree

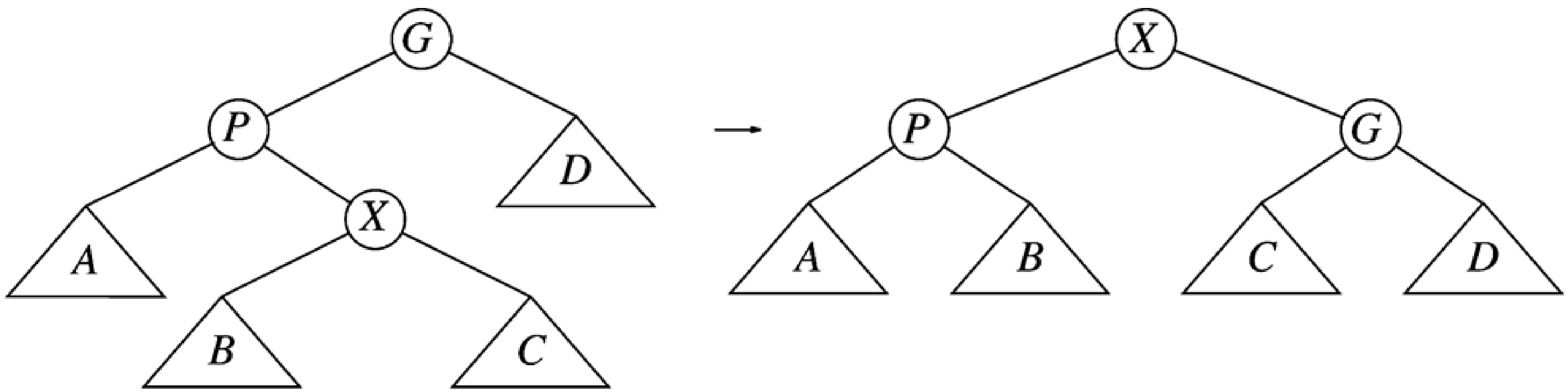
- Solution 1
  - Perform single rotations with accessed/new node and parent until accessed/new node is the root
  - Problem
    - Pushes current root node deep into tree
    - In general, can result in O(M*N) time for M operations
    - E.g., insert 1, 2, 3, …, N

Splay Tree

- Solution 2
  - Still rotate tree on the path from the new/accessed node X to the root
  - But, rotations are more selective based on node, parent and grandparent
  - If X is child of root, then rotate X with root
  - Otherwise, …

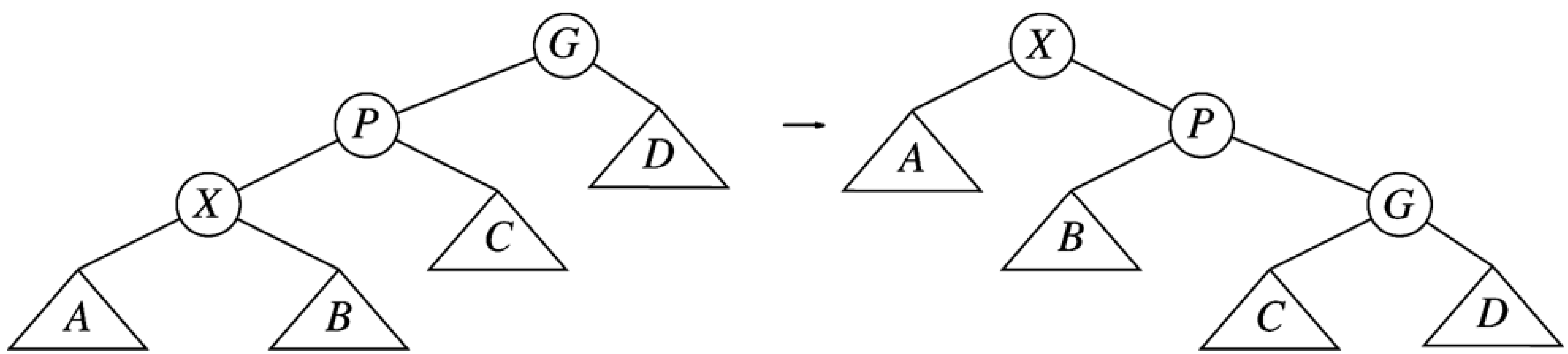
Splaying: Zig-zag

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



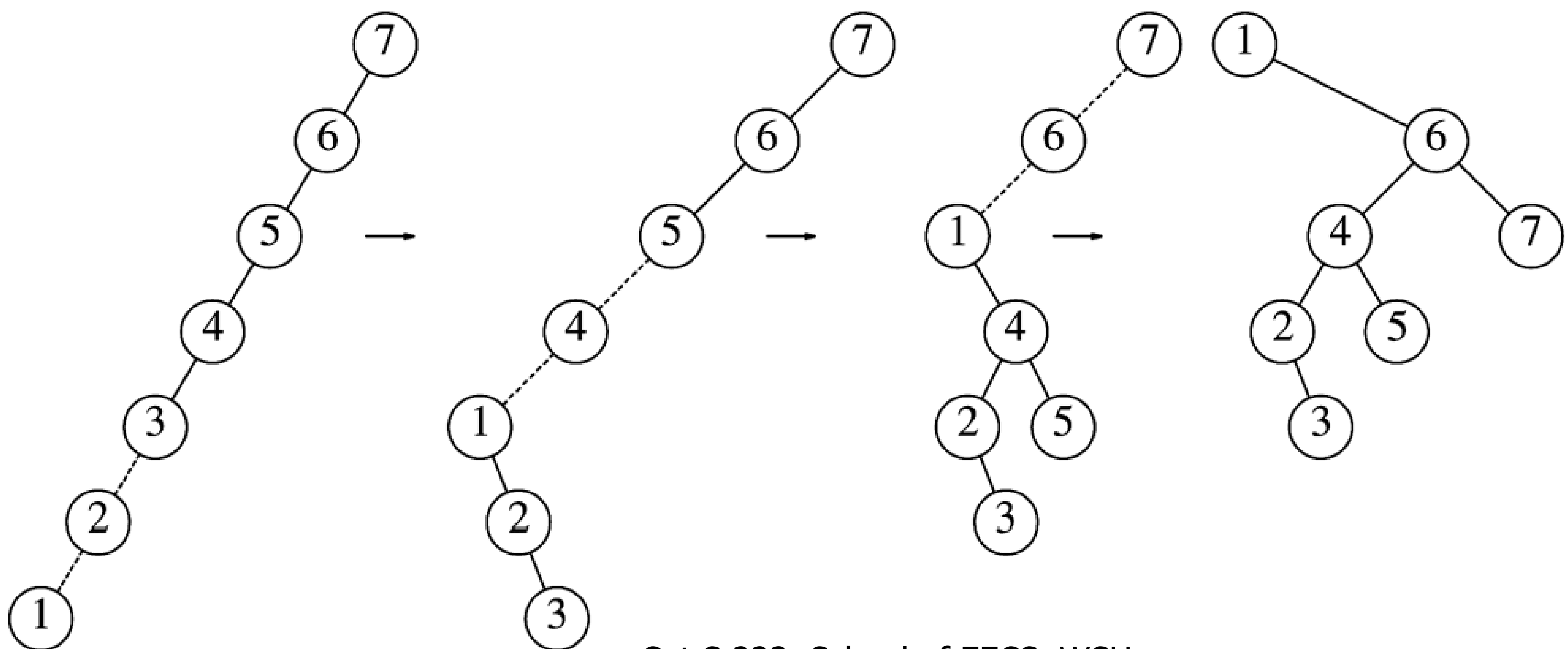
Splaying: Zig-zig

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



Splay Tree

- E.g., consider previous worst-case scenario: insert 1, 2, …, N



Splay Tree: Remove

- Access node to be removed (now at root)
- Remove node leaving two subtrees T_L and T_R
- Access largest element in T_L
  - Now at root; no right child
- Make T_R right child of root of T_L

Balanced BSTs

- AVL trees
  - Guarantees $O(\log_2 N)$ behavior
  - Requires maintaining height information
- Splay trees
  - Guarantees amortized $O(\log_2 N)$ behavior
  - Moves frequently-accessed elements closer to root of tree
- Other self-balancing BSTs:
  - Red-black tree (used in STL)
  - Scapegoat tree
  - Treap
- All these trees assume N-node tree can fit in main memory
  - If not?

Balanced Binary Search Trees in STL: `set` and `map`

- **`vector`** and **`list`** STL classes inefficient for search

- STL **`set`** and **`map`** classes guarantee logarithmic insert, delete and search

STL `set` Class

- STL **`set`** class is an ordered container that does not allow duplicates
- Like lists and vectors, sets provide iterators and related methods: **`begin`**, **`end`**, **`empty`** and **`size`**
- Sets also support **`insert`**, **`erase`** and **`find`**

Set Insertion

- **`insert`** adds an item to the set and returns an iterator to it

- Because a *set* does not allow duplicates, **`insert`** may fail
  - In this case, **`insert`** returns an iterator to the item causing the failure
  - (if you want duplicates, use *multiset*)

- To distinguish between success and failure, **`insert`** actually returns a pair of results
  - This ***`pair`*** structure consists of an iterator and a Boolean indicating success

```
pair<iterator,bool> insert (const Object & x);
```

Sidebar: STL **pair** Class

- **pair<Type1,Type2>**
- Methods: **first**, **second**, **first_type**, **second_type**

```
#include <utility>

pair<iterator,bool> insert (const Object & x)
{
  iterator itr;
  bool found;
  …
  return pair<itr,found>;
}
```

Example code for set insert

```
set<int> s;
//insert
for (int i = 0; i < 1000; i++){
  s.insert(i);
}

//print
iterator<set<int>> it=s.begin();
for(it=s.begin(); it!=s.end();it++) {
      cout << *it << " " << endl;
}
```

What order will the elements get printed?

Sorted order (iterator does an in-order traversal)

Example code for set insert

Write another code to test the return condition of each insert:

```
set<int> s;
pair<iterator<set<int>>,bool> ret;
for (int i = 0; i < 1000000; i++){
  ret = s.insert(i);
  … ?

}
```

Set Insertion

- Giving **`insert`** a hint

```
pair<iterator,bool> insert (iterator hint, const Object & x);
```

- For good hints, **`insert`** is O(1)
- Otherwise, reverts to one-parameter **`insert`**
- E.g.,

```
set<int> s;
for (int i = 0; i < 1000000; i++)
  s.insert (s.end(), i);
```

Set Deletion

- `int erase (const Object & x);`
  - Remove x, if found
  - Return number of items deleted (0 or 1)
- `iterator erase (iterator itr);`
  - Remove object at position given by iterator
  - Return iterator for object after deleted object
- `iterator erase (iterator start, iterator end);`
  - Remove objects from start up to (but not including) end
  - Returns iterator for object after last deleted object
  - **`Again, iterator advances from start to end using in-order traversal`**

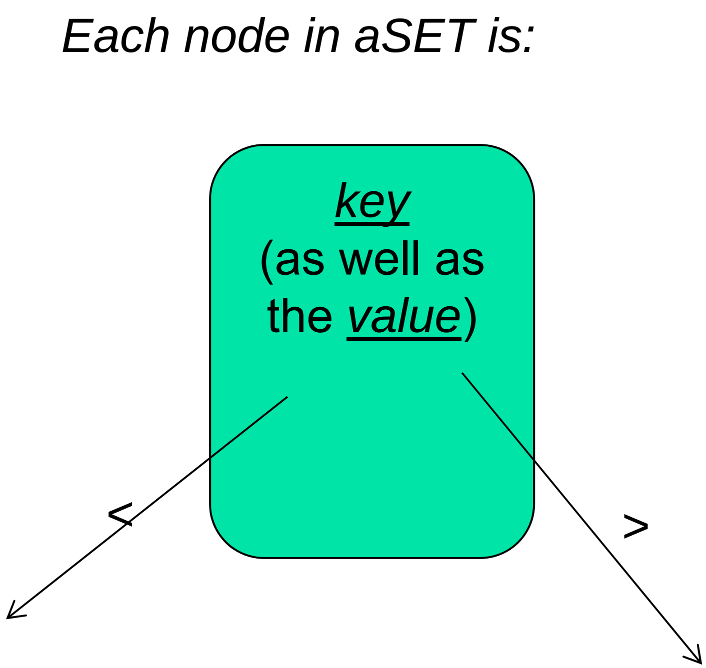
Set Search

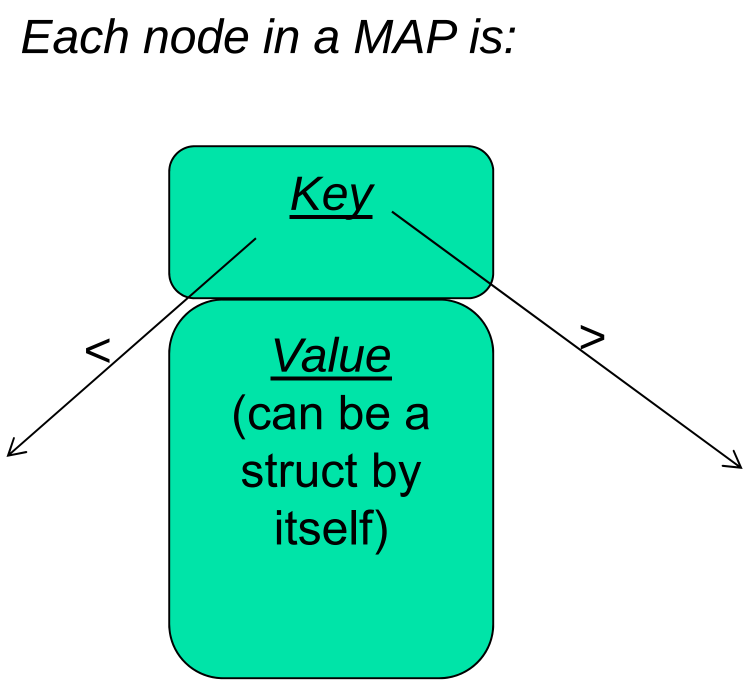
- **`iterator find (const Object & x) const;`**
  - Returns iterator to object (or end() if not found)
  - Unlike **`contains`**, which returns Boolean
- **`find`** runs in logarithmic time

STL `map` Class

- Associative container
  - Each item is 2-tuple: [Key, Value]
- STL **`map`** class stores items *sorted by Key*
- set vs. map:
  - The **`set`** class ≡ **`map`** where key is the whole record
- Keys must be unique (no duplicates)
  - If you want duplicates, use mulitmap
- Different keys can map to the same value
- Key type and Value type can be totally different

STL set and map classes





STL `map` Class

- Methods
  - **`begin, end, size, empty`**
  - **`insert, erase, find`**
- Iterators reference items of type **`pair<KeyType,ValueType>`**
- Inserted elements are also of type **`pair<KeyType,ValueType>`**

Syntax: MapObject[key] returns value

STL `map` Class

- Main benefit: overloaded **`operator[]`**

```
ValueType & operator[] (const KeyType & key);
```

- If key is present in map
  - Returns reference to corresponding value
- If key is not present in map
  - Key is inserted into map with a default value
  - Reference to default value is returned

```
map<string,double> salaries;
salaries["Pat"] = 75000.0;
```

Example

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  map<const char*, int, ltstr> months;
  months["january"] = 31;
  months["february"] = 28;
  months["march"] = 31;
  months["april"] = 30;
  ...
```

Key type

Value type

Comparator if Key type not primitive

key

value

- You really don't have to call insert() explicitly.
- This syntax will do it for you.
- If element already exists, then value will be updated.

Example (cont.)

```
  ...
  months["may"] = 31;
  months["june"] = 30;
  ...
  months["december"] = 31;

  cout << “february -> " << months[“february"] << endl;
  map<const char*, int, ltstr>::iterator cur = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next; --prev;
  cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
  cout << "Next (in alphabetical order) is " << (*next).first << endl;

  months["february"] = 29;
  cout << “february -> " << months[“february"] << endl;
}
```

What will this code do?

Implementation of `set` and `map`

- Support insertion, deletion and search in worst-case logarithmic time
- Use balanced binary search tree (a red-black tree)
- Support for iterator
  - Tree node points to its predecessor and successor
    - Which traversal order?

When to use *set* and when to use *map*?

- set
  - Whenever your entire record structure to be used as the Key
  - E.g., to maintain a searchable set of numbers

- map
  - Whenever your record structure has fields other than Key
  - E.g., employee record (search Key: ID, Value: all other info such as name, salary, etc.)

B-Trees

A Tree Data Structure for Disks

Top 10 Largest Databases

Organization	Database Size
WDCC	6,000 TBs
NERSC	2,800 TBs
AT&T	323 TBs
Google	33 trillion rows (91 million insertions per day)
Sprint	3 trillion rows (100 million insertions per day)
ChoicePoint	250 TBs
Yahoo!	100 TBs
YouTube	45 TBs
Amazon	42 TBs
Library of Congress	20 TBs

Source: www.businessintelligencelowdown.com, 2007.

How to count the bytes?

- Kilo ≈ x 10^{3}
- Mega ≈ x 10^{6}
- Giga ≈ x 10^{9}
- Tera ≈ x 10^{12}

Current limit for single node storage

- Peta ≈ x 10^{15}
- Exa ≈ x 10^{18}
- Zeta ≈ x 10^{21}
- …

Needs more sophisticated disk/IO machine

Primary storage vs. Disks

	Primary Storage	Secondary Storage
Hardware	RAM (main memory), cache	Disk (ie., I/O)
Storage capacity	>100 MB to 2-4GB	Giga (10^9) to Terabytes (10^{12}) to..
Data persistence	Transient (erased after process terminates)	Persistent (permanently stored)
Data access speeds	~ a few clock cycles (ie., x 10^{-9} seconds)	milliseconds (10^{-3} sec) = Data seek time + read time

could be ***million times slower*** than main memory read

Use a balanced BST?

- Google: 33 trillion items
- Indexed by ?
  - IP, HTML page content

- Estimated access time (if we use a simple balanced BST):
  - h = O($\log_2 33 \times 10^{12}$) $\cong$44.9 disk accesses
  - Assume 120 disk accesses per second

    ==> Each search takes 0.37 seconds

What happens if you do a million searches?

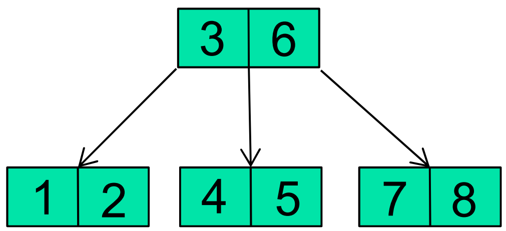
- 1 disk access == > 10^6 CPU instructions

Main idea: ***Height reduction***

- Why ?
  - BST, AVL trees at best have heights O(lg n)
    - $N=10^6$ ➔ lg 10^6 is roughly 20
    - 20 disk seeks for each level would be too much!
  - So reduce the height !
  - How?
    - Increase the log base beyond 2
    - Eg., $\log_5 10^6$ is < 9
    - Instead of binary (2-ary) trees, use m-ary search trees s.t. m>2

How to store an m-way tree?

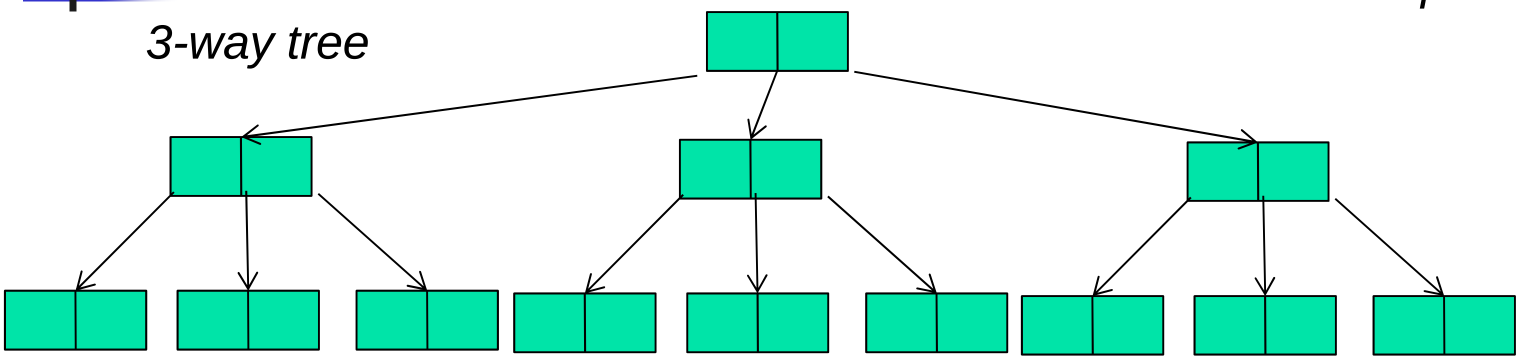
- Example: 3-way search tree
- Each node stores:
  - ≤ 2 keys
  - ≤ 3 children



- Height of a balanced 3-way search tree?

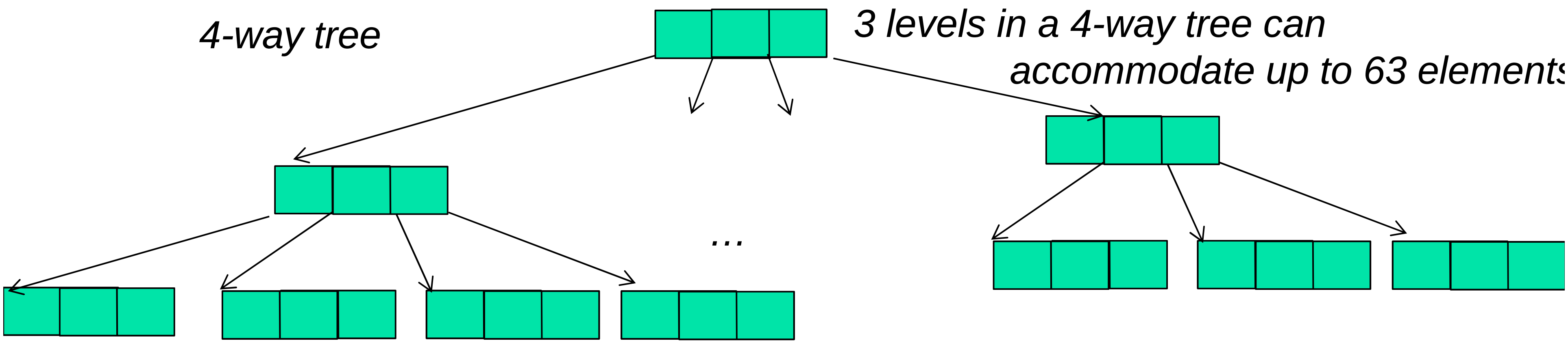
3 levels in a 3-way tree can
accommodate up to 26 elements

3-way tree



4-way tree

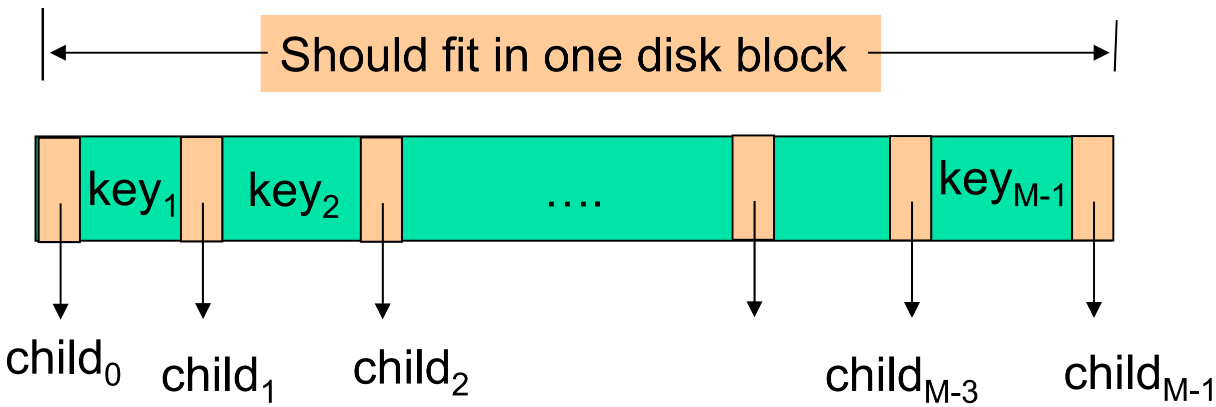
3 levels in a 4-way tree can
accommodate up to 63 elements



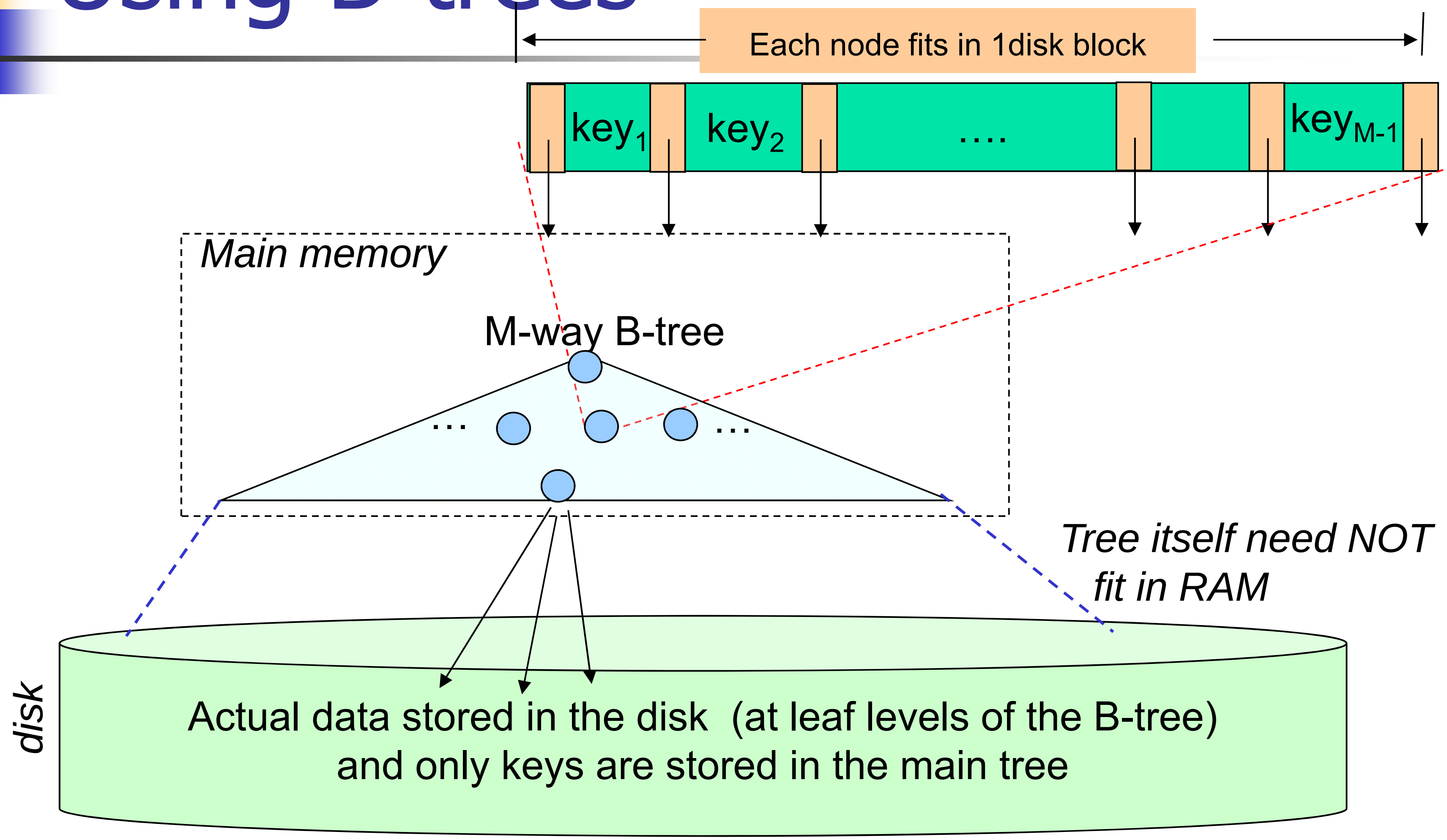
Bigger Idea

- Use an M-way search tree
- Each node access brings in M-1 keys an M child pointers
- Choose M so node size = 1 disk block size
- Height of tree = $\Theta(\log_M N)$

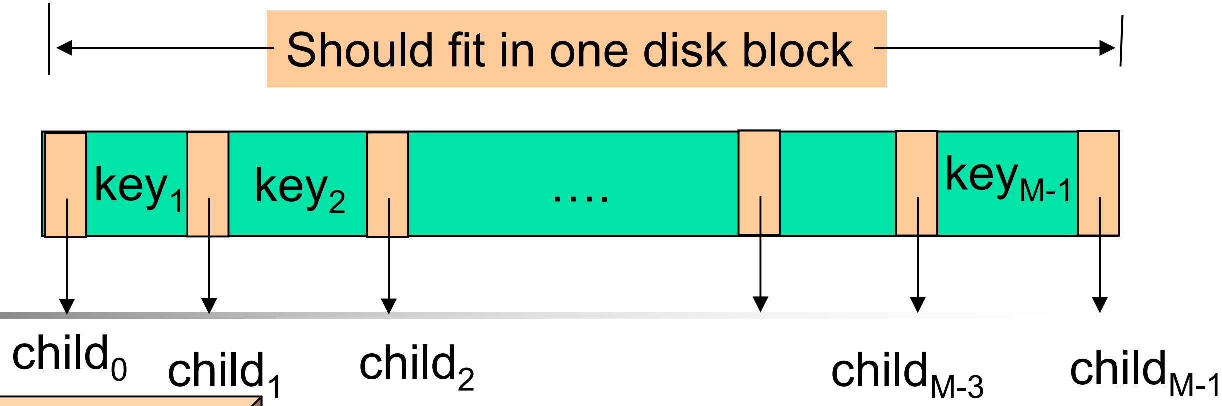
Tree node structure:



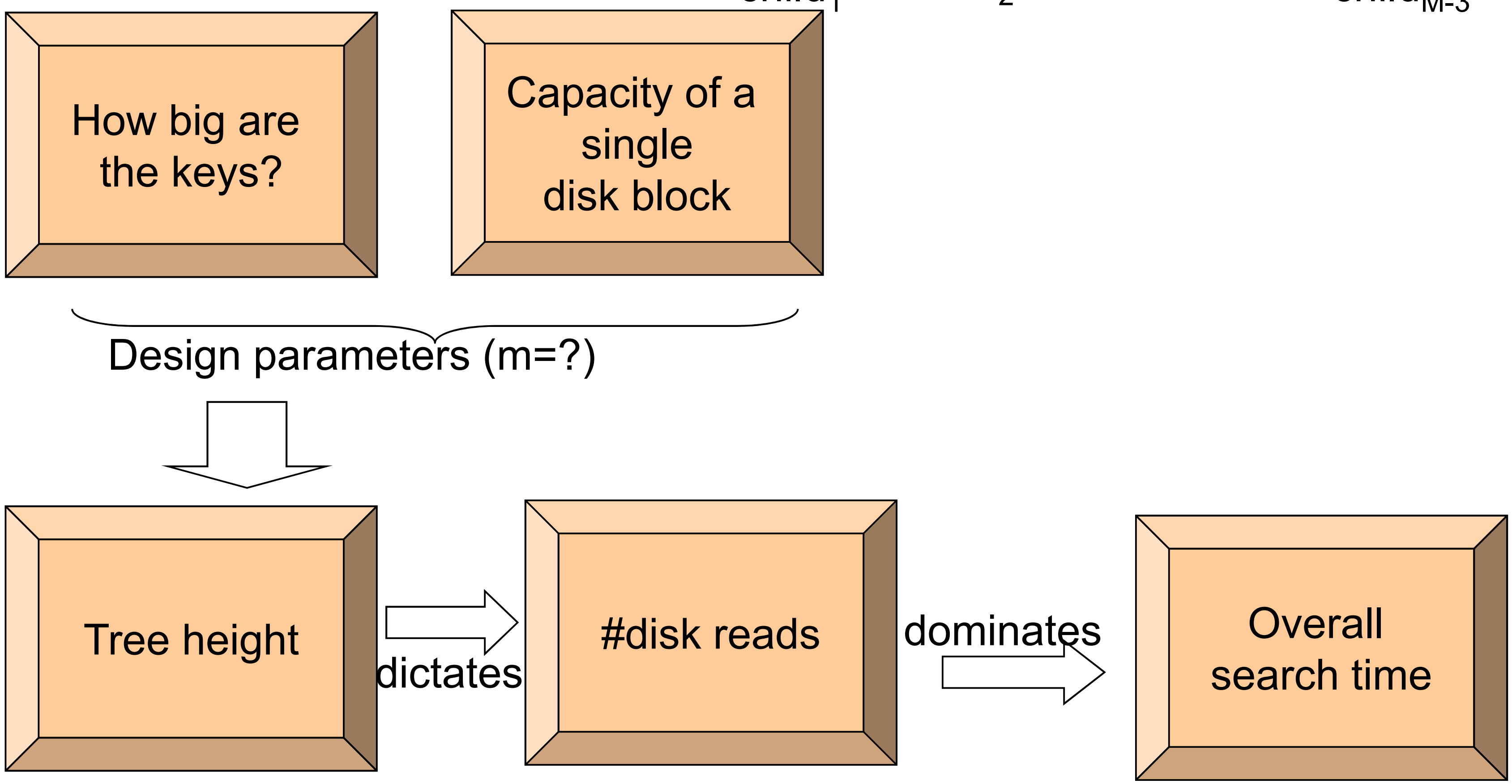
Using B-trees



Factors

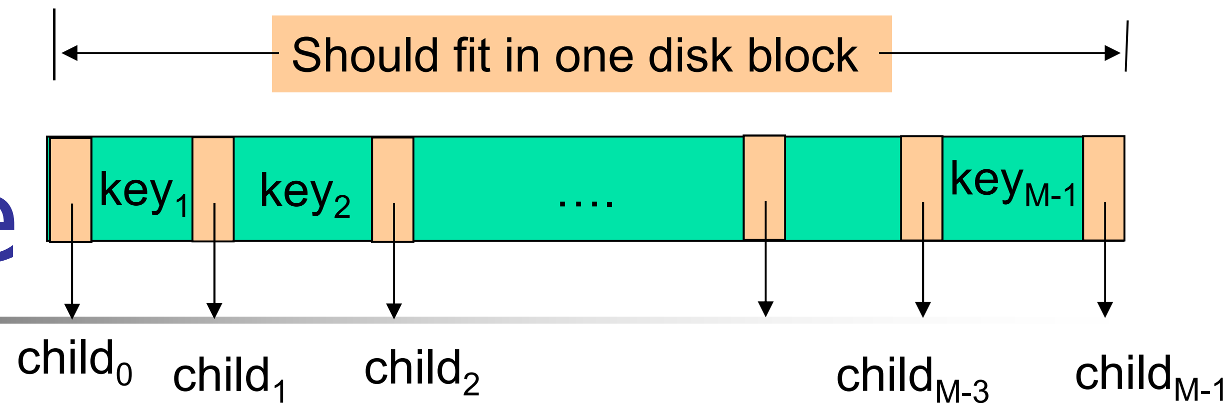


Should fit in one disk block
key$_1$
key$_2$
….
key$_{M-1}$
child$_0$
child$_1$
child$_2$
child$_{M-3}$
child$_{M-1}$



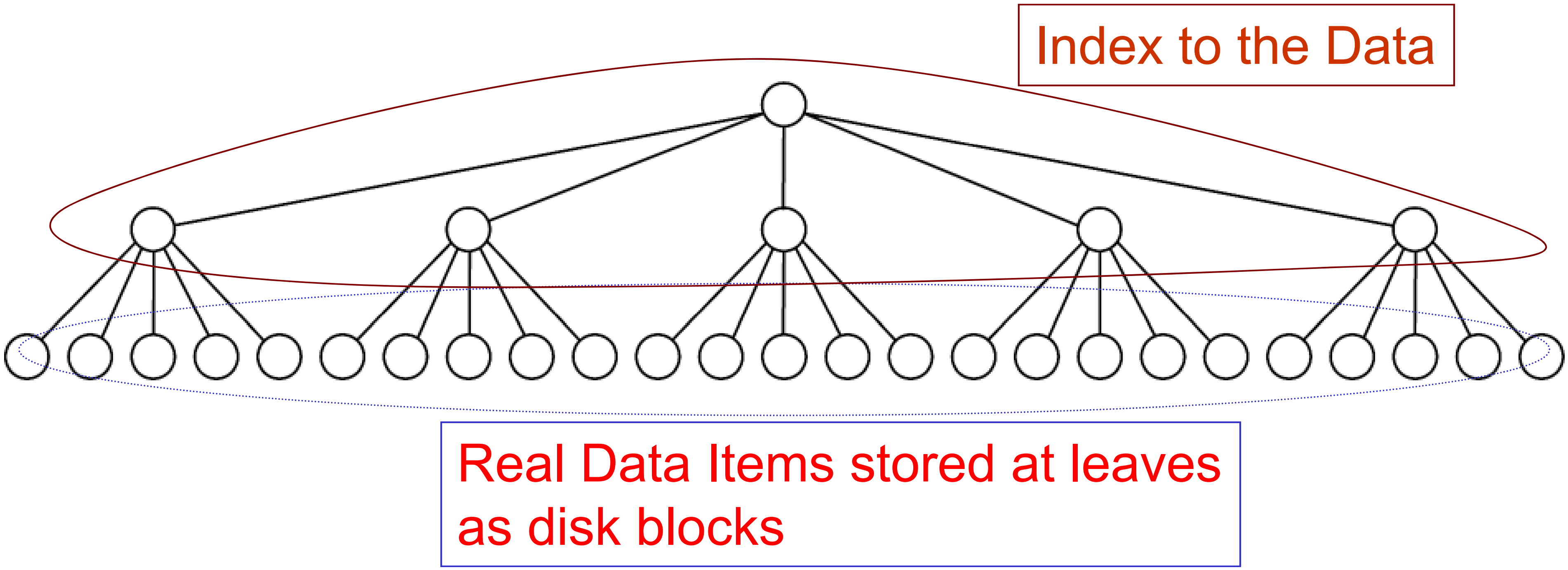
How big are the keys?
Capacity of a single disk block
Design parameters (m=?)
Tree height
dictates
#disk reads
dominates
Overall search time

Example



- Standard disk block size = 8192 bytes
- Assume keys use 32 bytes, pointers use 4 bytes
  - Keys uniquely identify data elements
- Space per node = 32*(M-1) + 4*M = 8192
- M = 228
- $\log_{228} 33 \times 10^{12} = 5.7$ (disk accesses)
- Each search takes 0.047 seconds

5-way tree of 31 nodes has only 3 levels



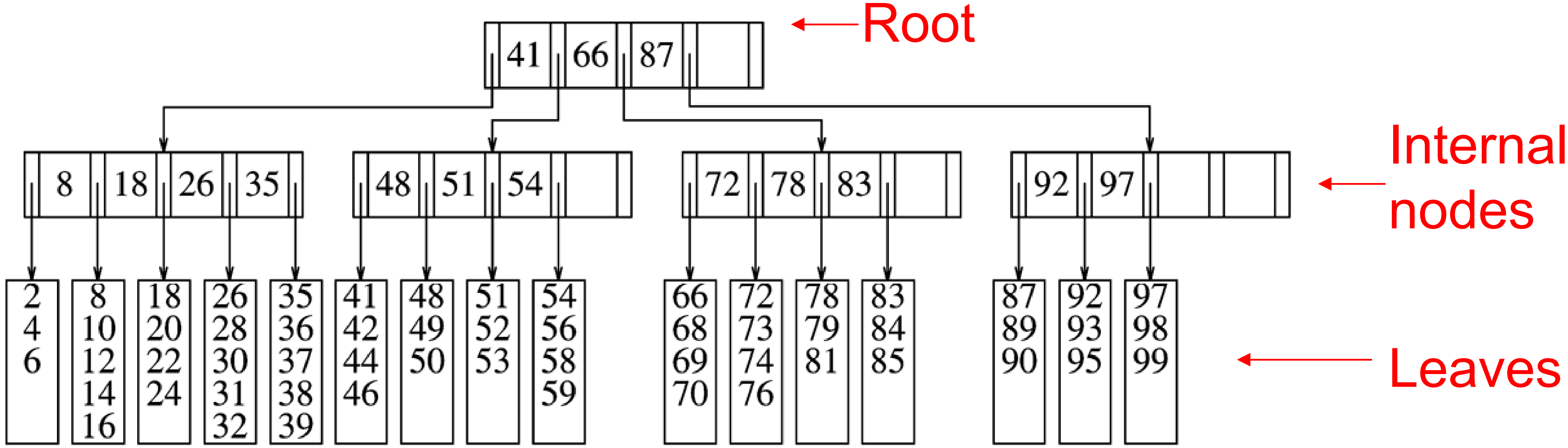
B+ trees: Definition

A *B+ tree* of order *M* is an *M-way tree* with all the following properties:

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

Parameters: N, M, L

B+ tree of order 5

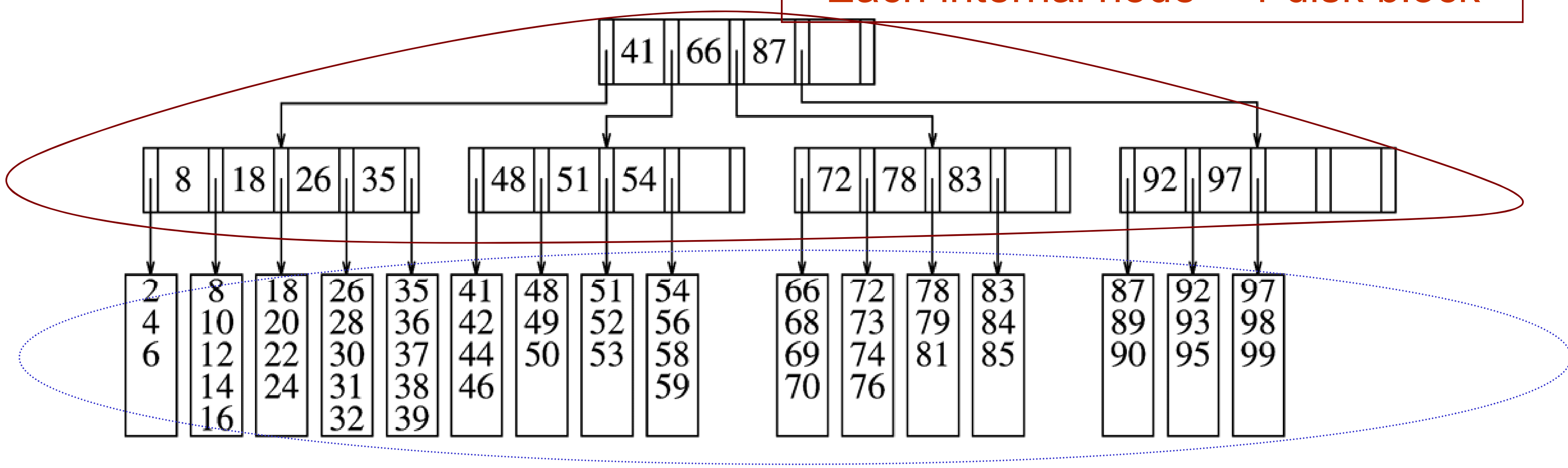


- M=5 (order of the B+ tree)
- L=5 (#data items bound for leaves)

- Each int. node (except root) has to have at least 3 children
- Each leaf has to have at least 3 data items

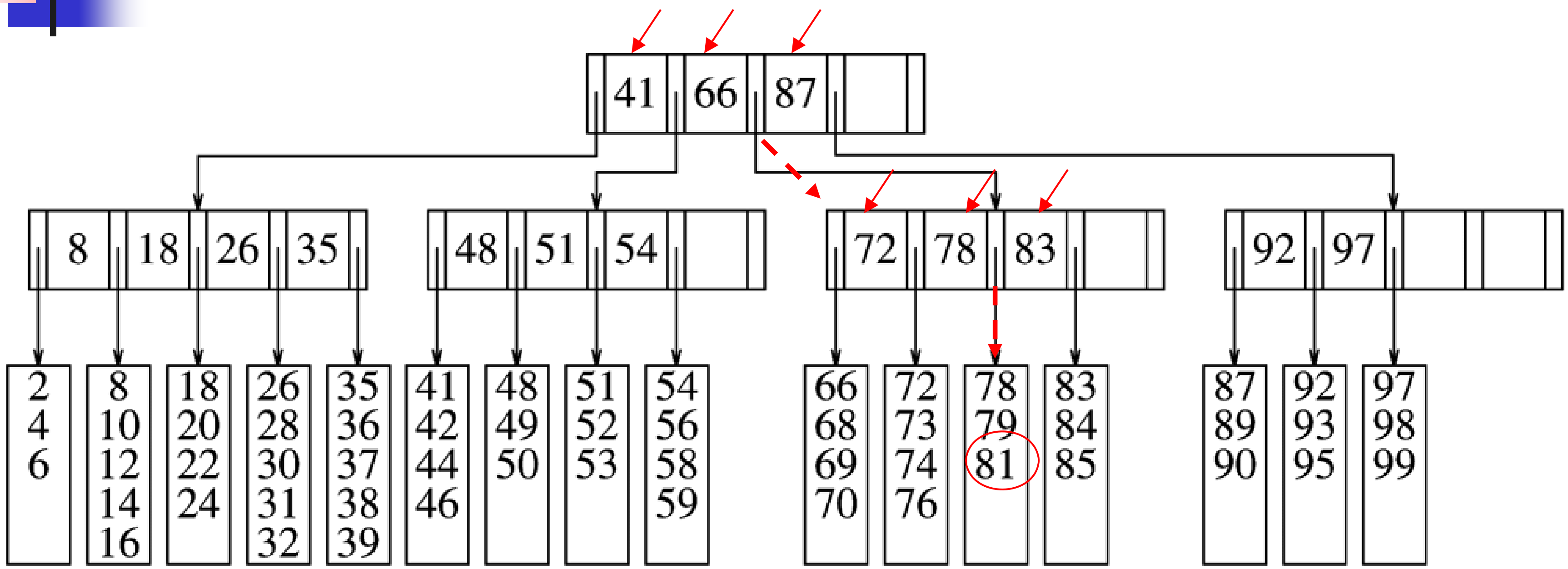
B+ tree of order 5

- Index to the Data (store only keys)
- Each internal node = 1 disk block



- Data items stored at leaves
- Each leaf = 1 disk block

Example: Find (81) ?



- $O(\log_M$ #leaves) disk block reads
- Within the leaf: O(L)
  - or even better, O(log L) if data items are kept sorted

How to design a B+ tree?

- How to find the #children per node?
  i.e., M=?
- How to find the #data items per leaf?
  i.e., L=?

Node Data Structures

- Root & internal nodes
  - M child pointers
    - 4 x M bytes
  - M-1 key entries
    - (M-1) x K bytes

- Leaf node
  - Let L be the max number of data items per leaf
  - Storage needed per leaf:
    - L x D bytes

- D denotes the size of each data item
- K denotes the size of a key (ie., K <= D)

How to choose M and L ?

- M & L are chosen based on:
  1. Disk block size (B)
  2. Data element size (D)
  3. Key size (K)

Calculating M: threshold for internal node capacity

- Each internal node needs
  - 4 x M + (M-1) x K bytes
- Each internal node has to fit inside a disk block
  - ==> B = 4M + (M-1)K
- Solving the above:
  - **M = floor[(B+K) / (4+K)]**

- Example: For K=4, B=8 KB:
  - M = 1,024

Calculating L: threshold for leaf capacity

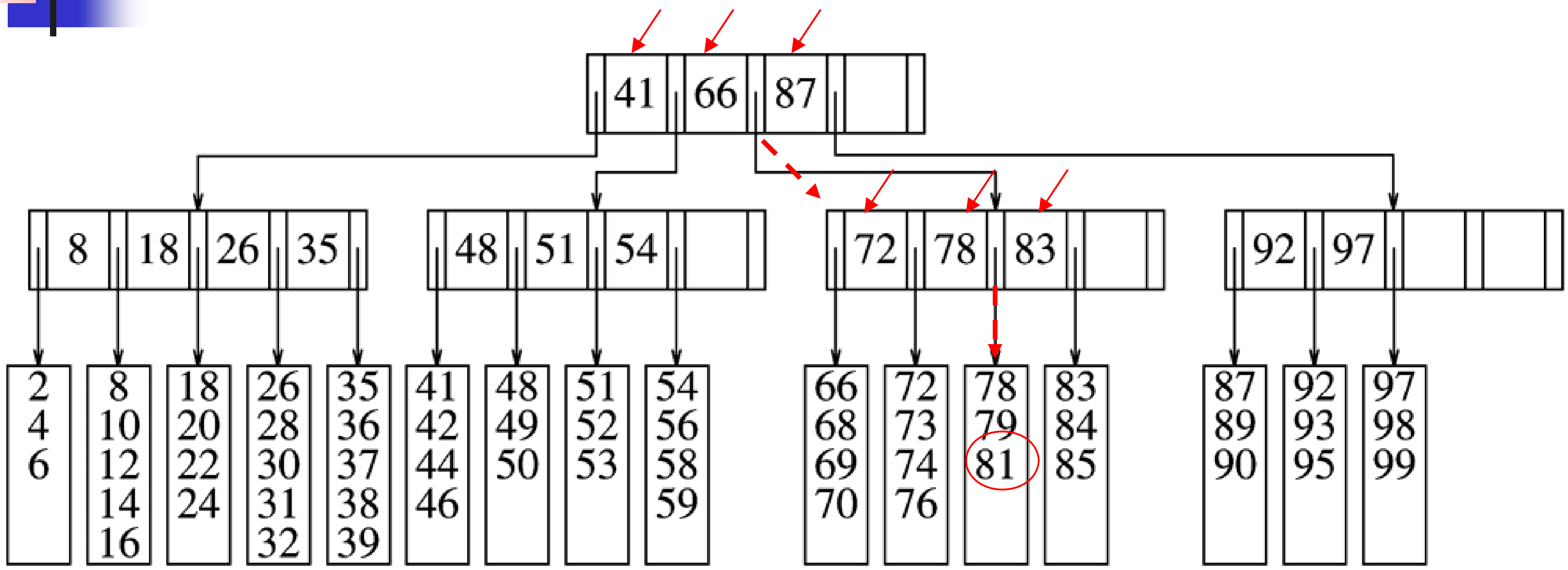
- **L = floor[B / D]**

- Example: For D=4, B = 8 KB:
  - L = 2,048
  - ie., each leaf has to store 1,024 to 2,048 data items

How to use a B+ tree?

- Find
- Insert
- Delete

Example: Find (81) ?



- $O(\log_M$ #leaves) disk block reads
- Within each internal node:
  - O(lg M) assuming binary search
- Within the leaf:
  - O(lg L) assuming binary search & data kept sorted

B+ trees: Other Counters

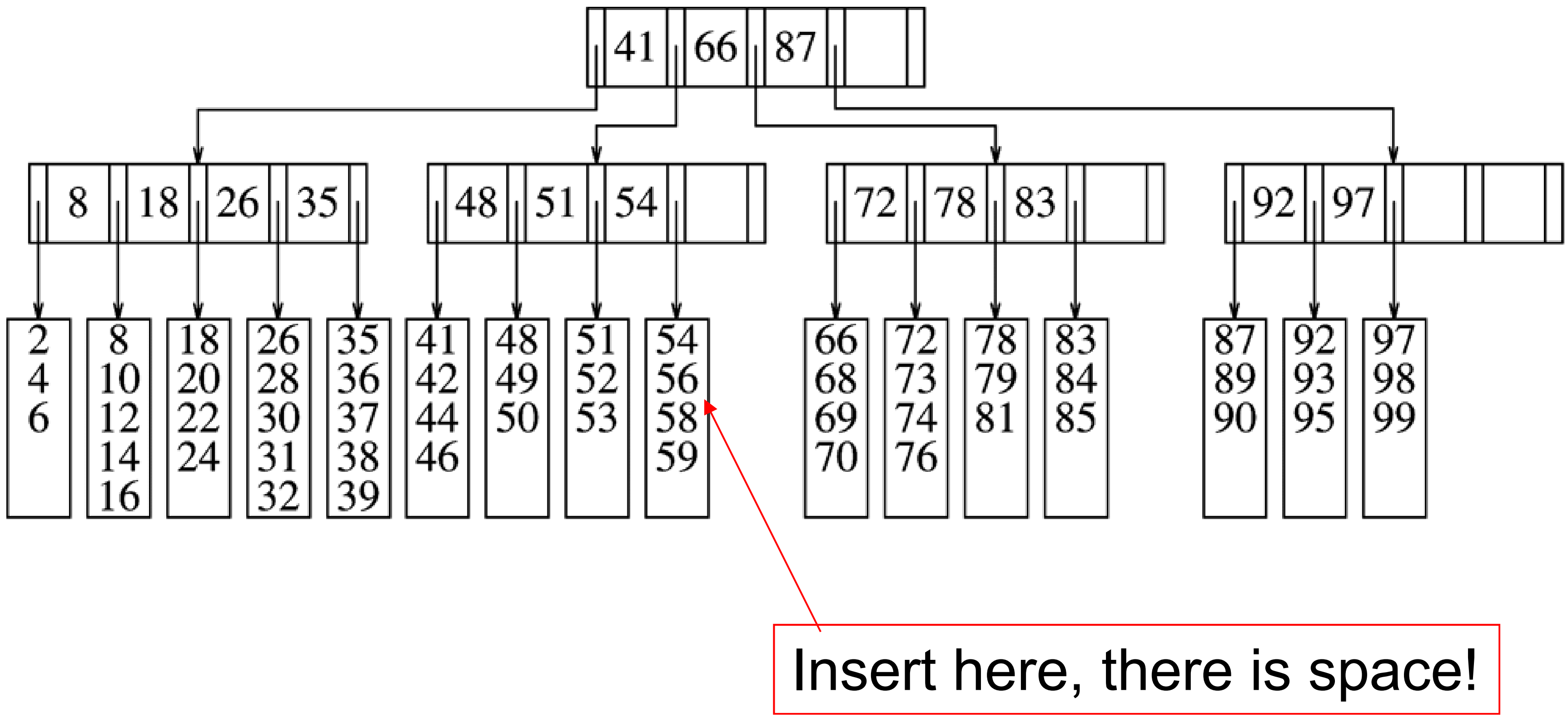
- Let N be the total number of data items
- How many leaves in the tree? how
  - = between ceil [N / L] and ceil [2N / L]

- What is the tree height? how
  - = O ($\log_M$ #leaves)

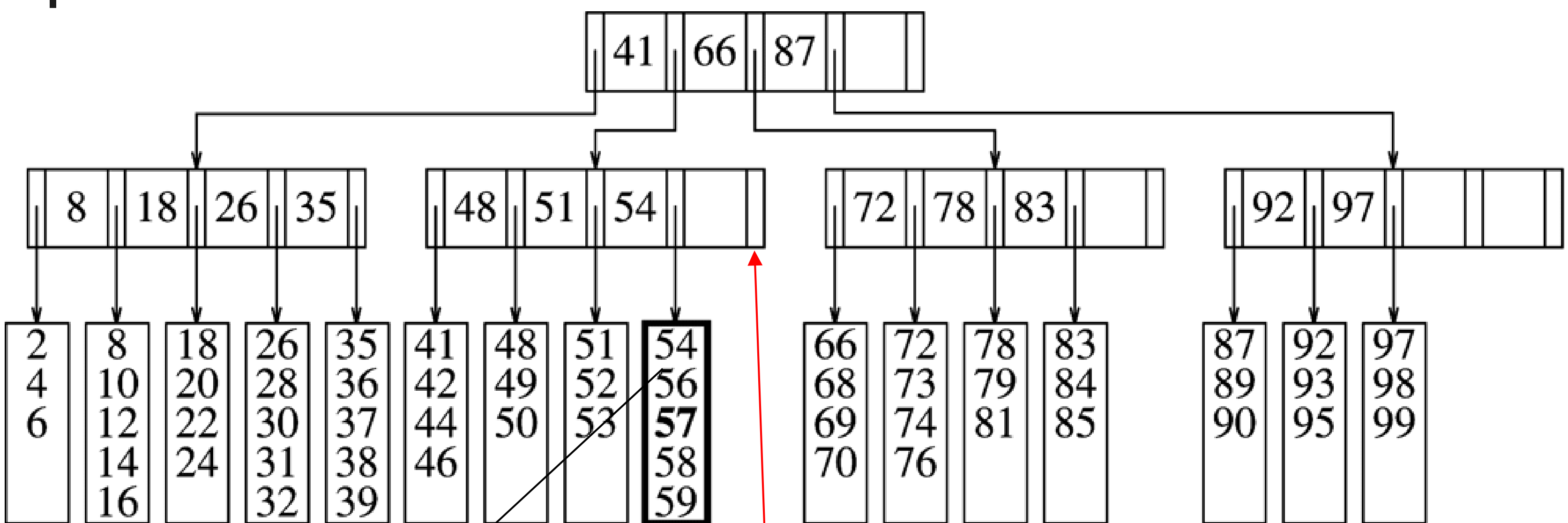
B+ tree: Insertion

- Ends up maintaining all leaves at the same level before and after insertion
- This could mean increasing the height of the tree

Example: Insert (57) before



Example: Insert (57) after



No more room

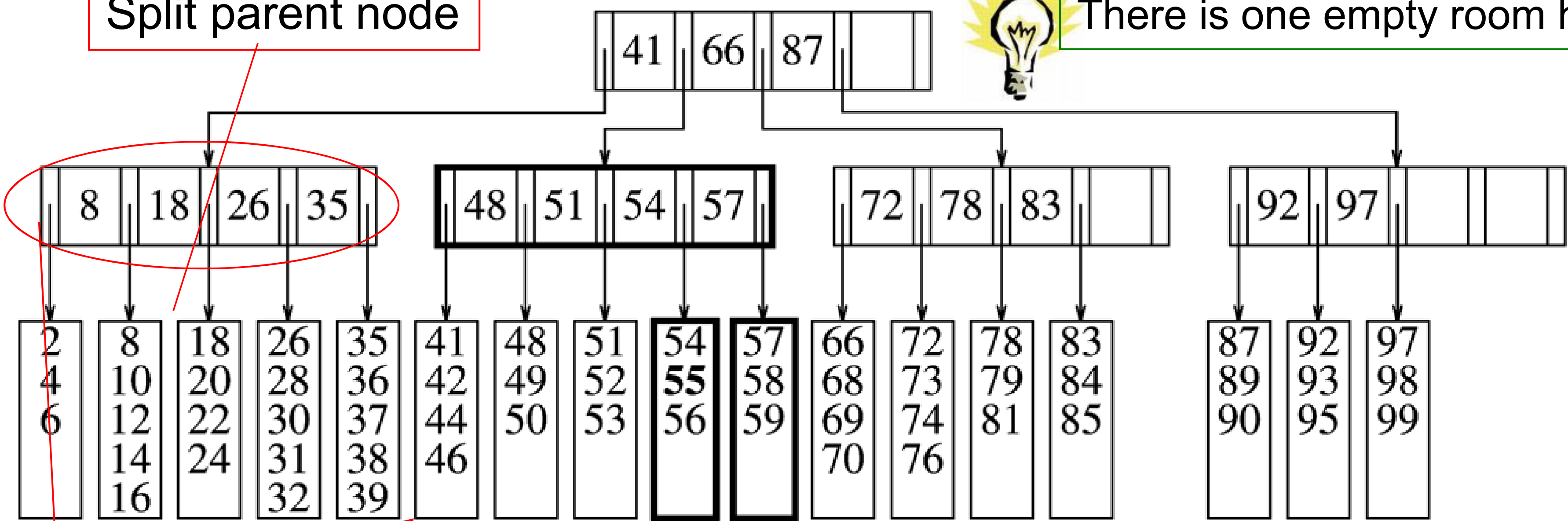
Next: Insert(55)

Empty now
So, split the previous leaf into 2 parts

Example.. Insert (55) after

Split parent node

There is one empty room here

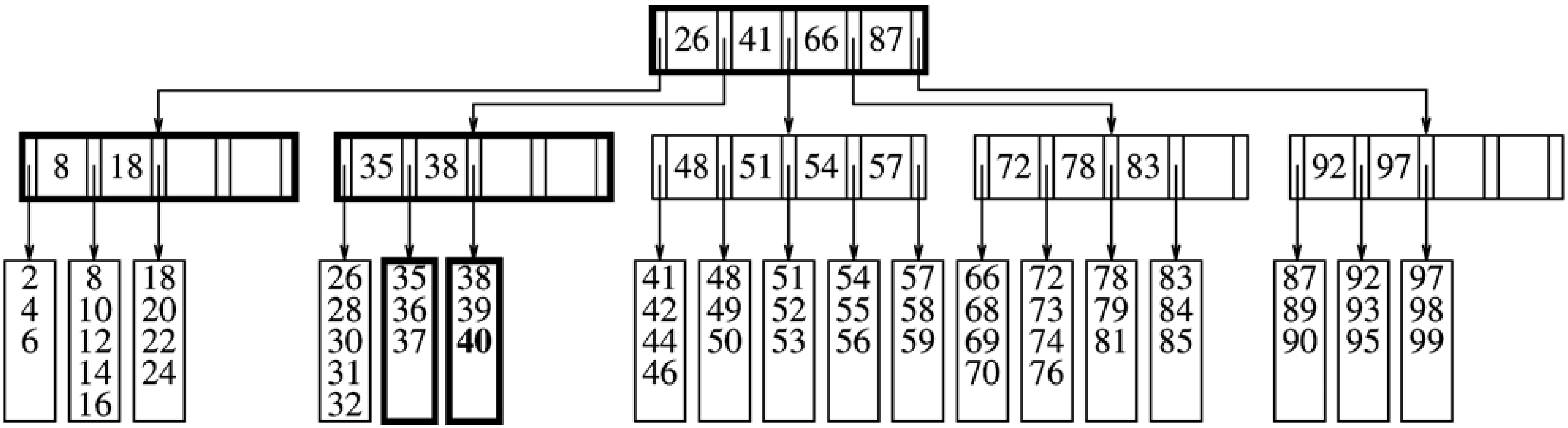


Next: Insert (40)

Hmm.. Leaf already full,
and no empty neighbors!

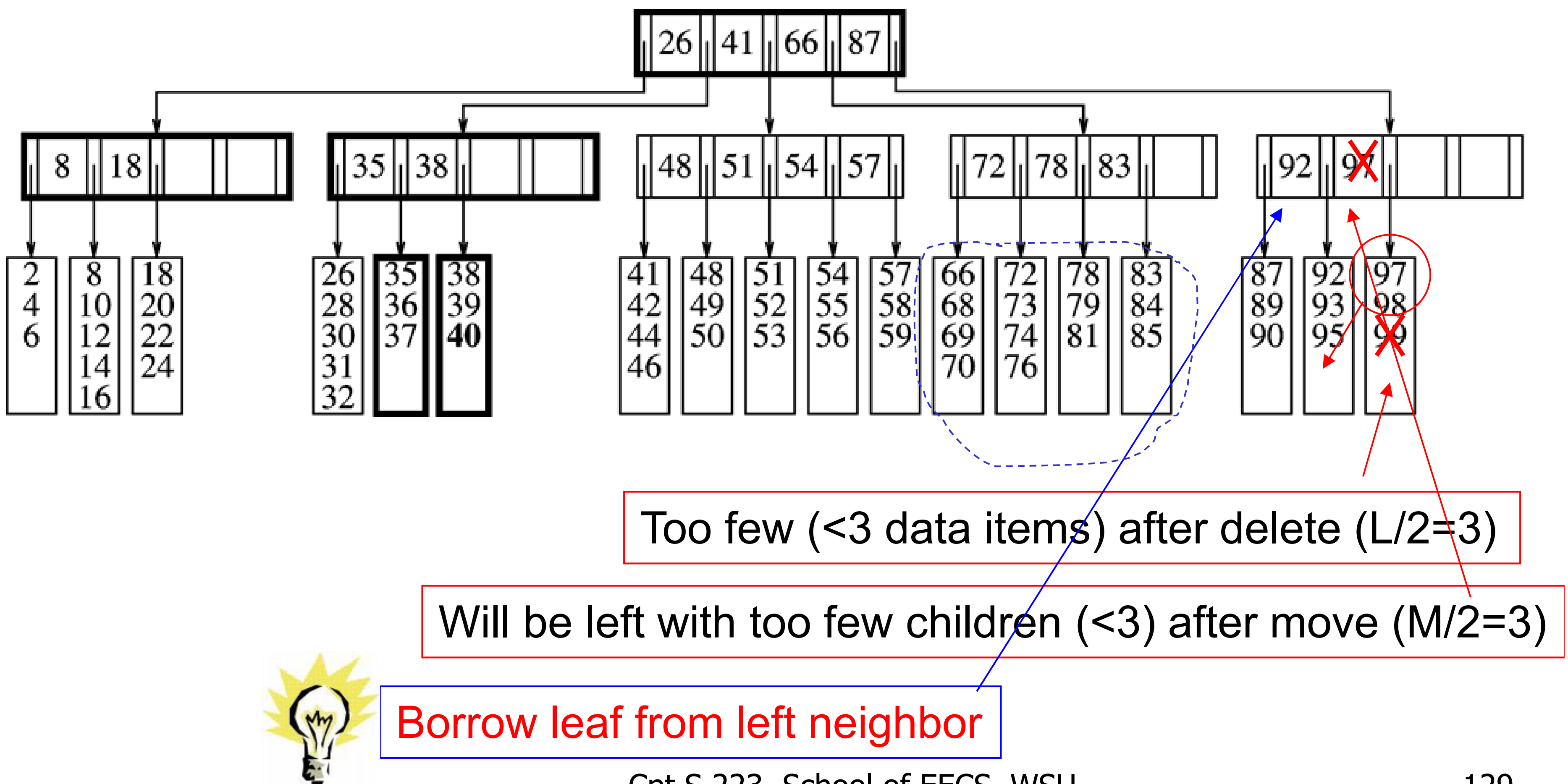
No space
here too

Example.. Insert (40) after

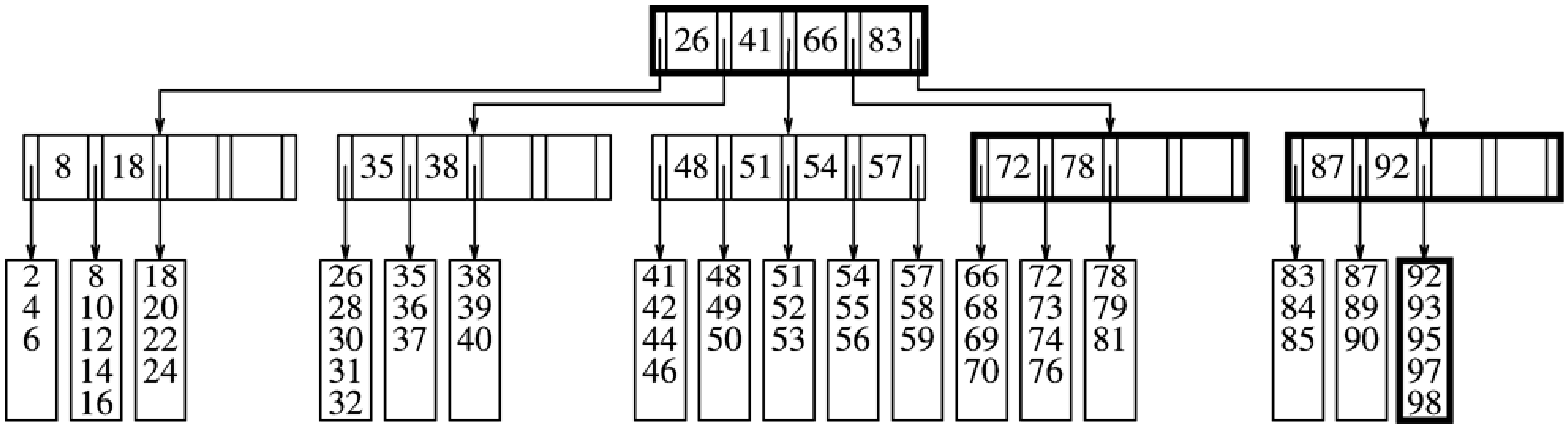


Note: Splitting the root itself would mean we are increasing the height by 1

Example.. Delete (99) before



Example.. Delete (99) after



Summary: Trees

- Trees are ubiquitous in software
- Search trees important for fast search
  - Support logarithmic searches
  - Must be kept balanced (AVL, Splay, B-tree)
- STL **`set`** and **`map`** classes use balanced trees to support logarithmic insert, delete and search
  - Implementation uses top-down red-black trees (not AVL) – Chapter 12 in the book
- Search tree for Disks
  - B+ tree