EECS 560
DATA STRUCTURES

MODULE IV: TREE

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.

-

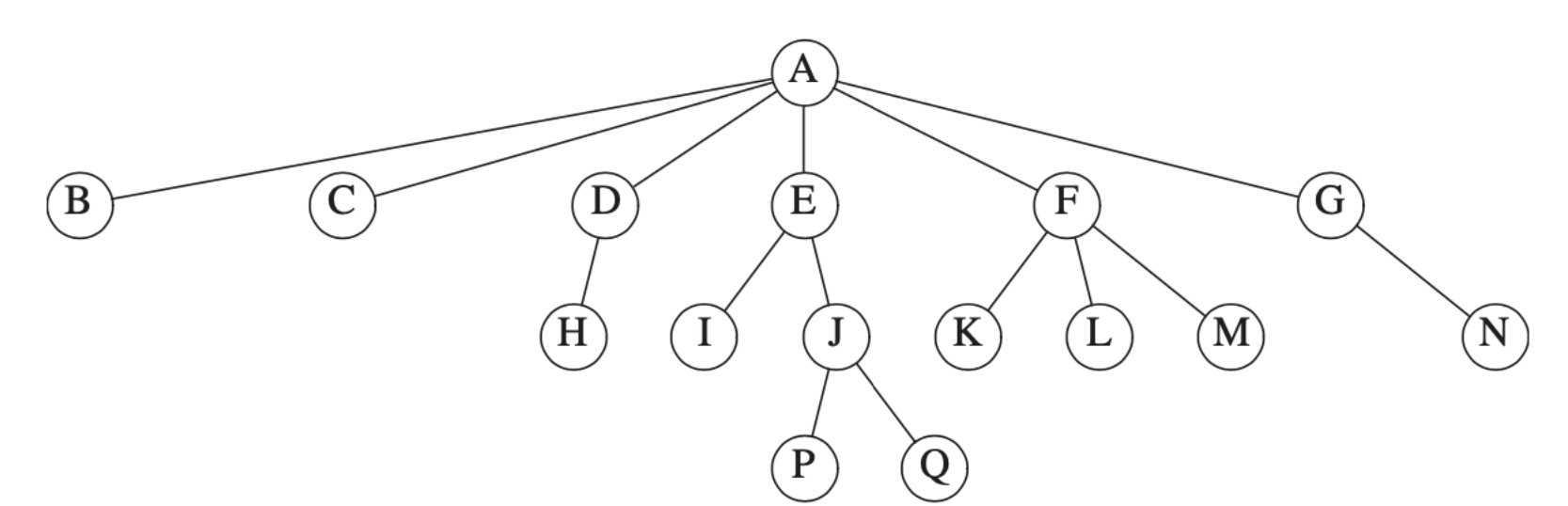
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Data Structures and Algorithm Analysis in C++***, 4th edition, by Mark Allen Weiss.

TREE

- We can think of the tree ADT as an extended linked list:
  - in the linked list, each element can link to at most one previous element and at most one next element
  - in the tree, each element can linked to at most one previous element **but multiple next elements**.



TERMINOLOGY

- In the context of tree ADT, we define the following terminology
  - each element (which stores the actual data) in the linked-list counterpart is formally defined as a tree node, or simply a **node**
  - if a node is not linked to any previous node, the node is called a **root**; if a node is not linked to any next node, the node is called a **leaf**; if a node is linked to both previous and next nodes, it is called an **internal node**
  - the between-node pointers as in the linked-list counterpart is defined as an **edge**
  - a set of ordered and continuous (the end node of the former edge is the starting node of the later edge) edges are called a **path**
  - given a node, the length (number of involved edges) of the path that goes from the root to itself is called its **depth**; the length of the longest path that goes from it to a leaf is called its **height**

TERMINOLOGY

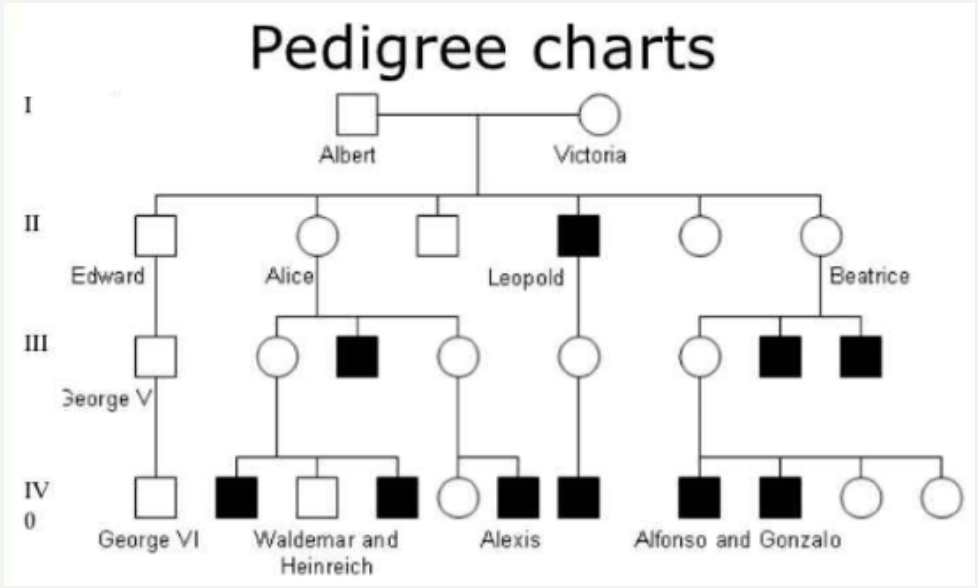
- Definition continued:
  - given a node, its immediate previous node is called its **parent**,
  - given a node, its immediate next nodes are called its **children** (or **child** if we are referring to a singular next node)
  - nodes that have the same parent are called **siblings**
  - the grandparent or grand-grandparent or grand-grand-grandparent… of a node is called its **ancestor** (alternatively, there exists a path that goes from the ancestor to the current node)
  - the grandchild or grand-grandchild or grand-grand-grandchild… of a node is called its **descendant** (alternatively, there exists a path that goes from the current node to the descendant)

DEFINITION

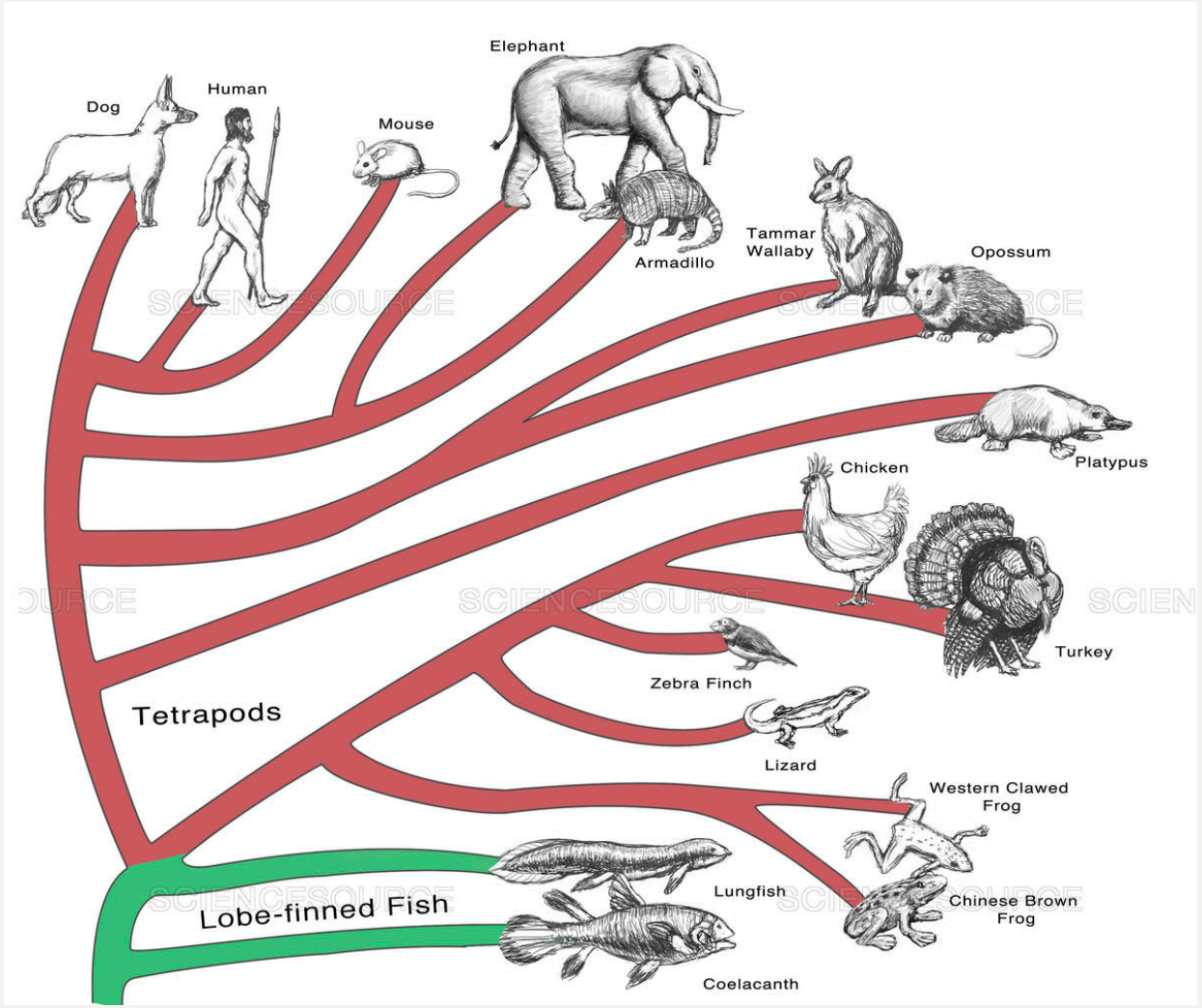
- A recursive definition:
  - A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished root, and zero or more nonempty (sub)trees, each of whose roots are connected by a directed edge from the root.

APPLICATIONS

- Many real-world relationships can be modeled as tree

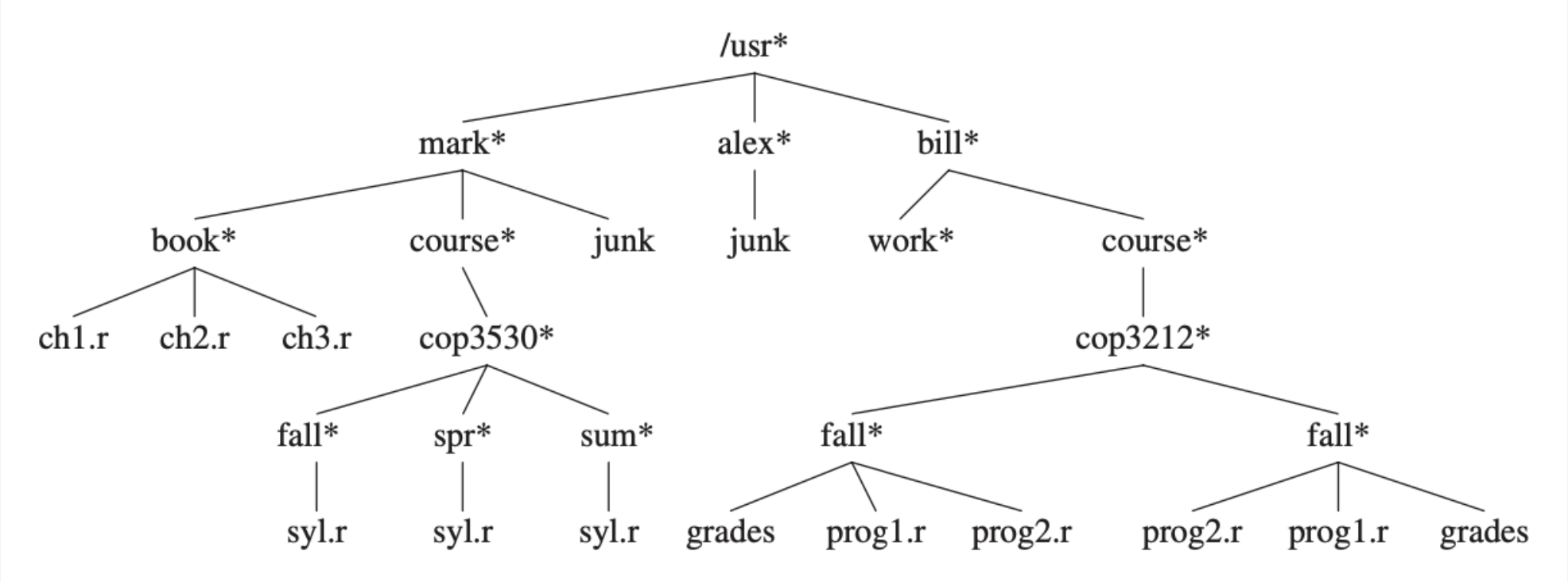


a pedigree tree
https://www.ilovefreesoftware.com/01/featured/free-online-pedigree-chart-maker-websites.html



a phylogenetic tree
https://www.sciencesource.com/archive/Evolutionary-Tree--Animals-SS2737563.html

APPLICATIONS



A Unix file system organization

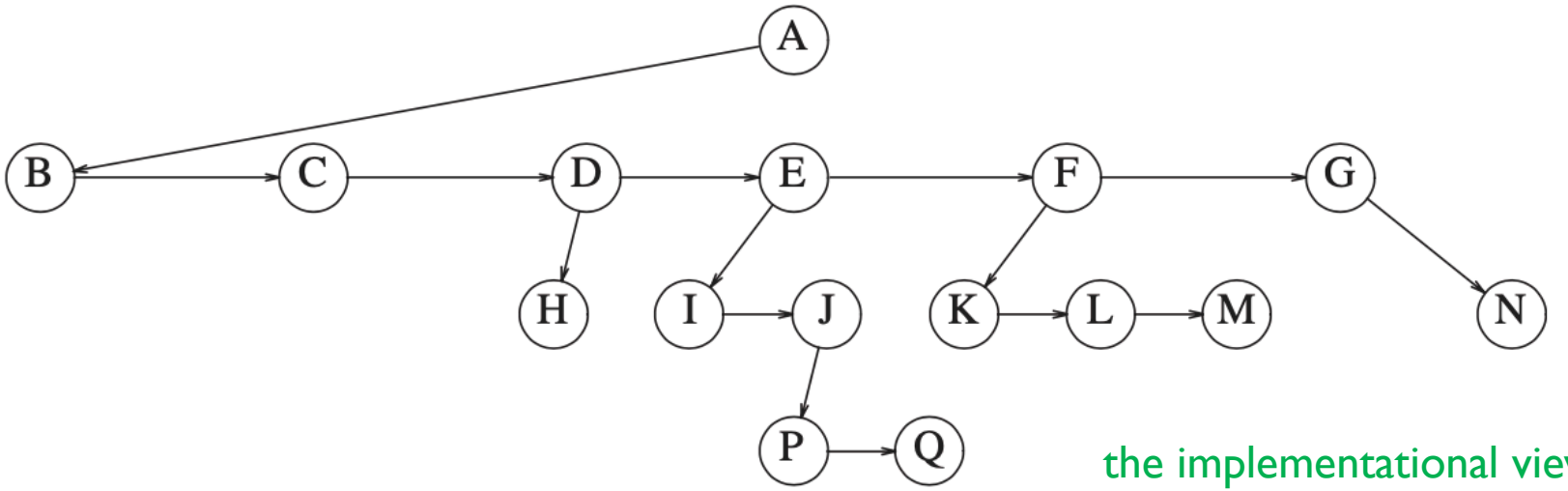
APPLICATIONS

- More importantly, the tree ADT can be used as a data holder that balances the search performance between linked list and hash table
  - for linked list, searching for an element needs to go through the entire linked list and has a worst-case time complexity of O(n)
  - for hash table, searching for an element can be done in O(1); however, it is associated with a large constant because the computation of the hash function is much slower than simply following pointers (as in the linked list ADT)
  - the tree ADT offers an alternative performance: a worst-case time complexity of O(log(n)) with a small constant (only following links plus some simple comparisons)

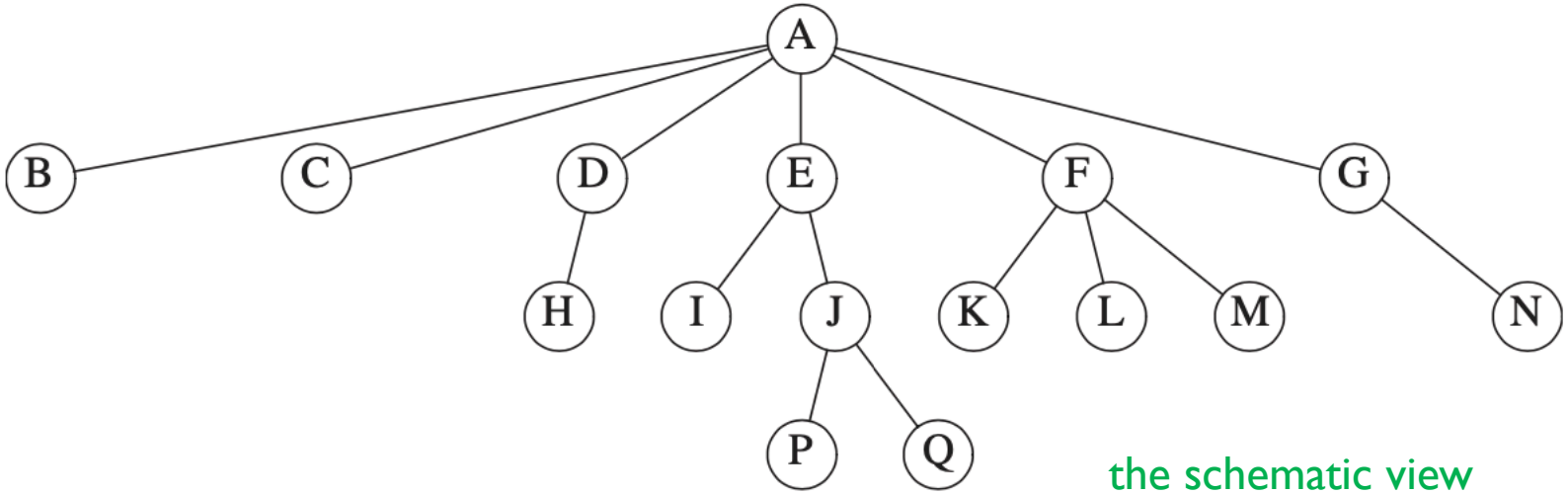
IMPLEMENTATION

```
1   struct TreeNode
2   {
3       Object    element;
4       TreeNode *firstChild;    // pointer that points to its leftmost child
5       TreeNode *nextSibling;   // pointer that points to its next sibling
6   };                           // we could add another pointer to the parent
```

IMPLEMENTATION



the implementational view



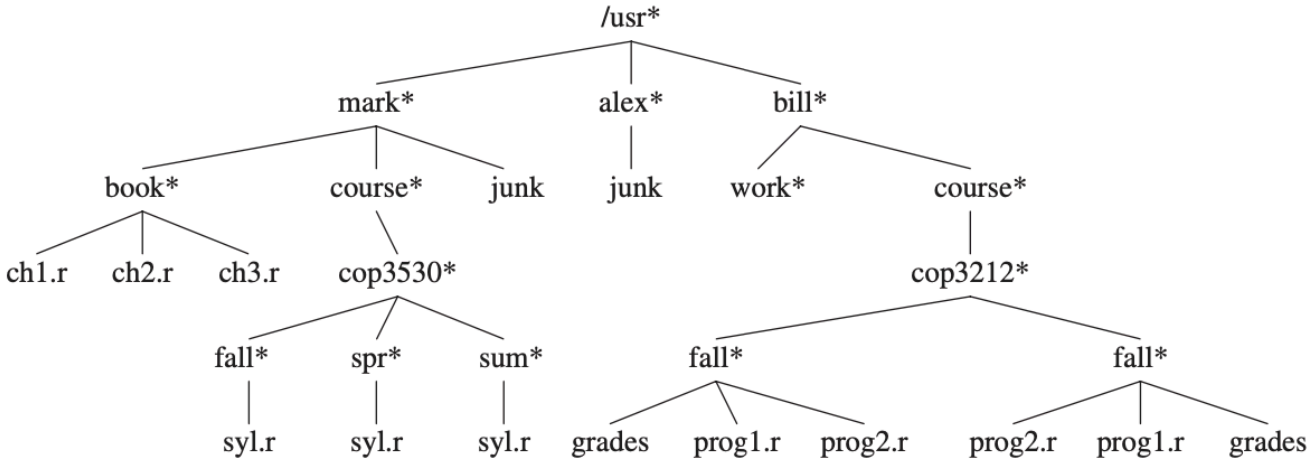
the schematic view

TREE TRAVERSAL

- Tree traversal: output all elements stored in the tree
- Tree traversal can be done in multiple ways (defined recursively)
  - preorder (current, left child,…, right child)
    - the current node is visited before the children, that's why it is named "pre"
  - postorder (left child,…, right child, current)
    - the current node is visited after the children, that's why it is called "post"

TREE TRAVERSAL

- Preorder application: listing the file directory path

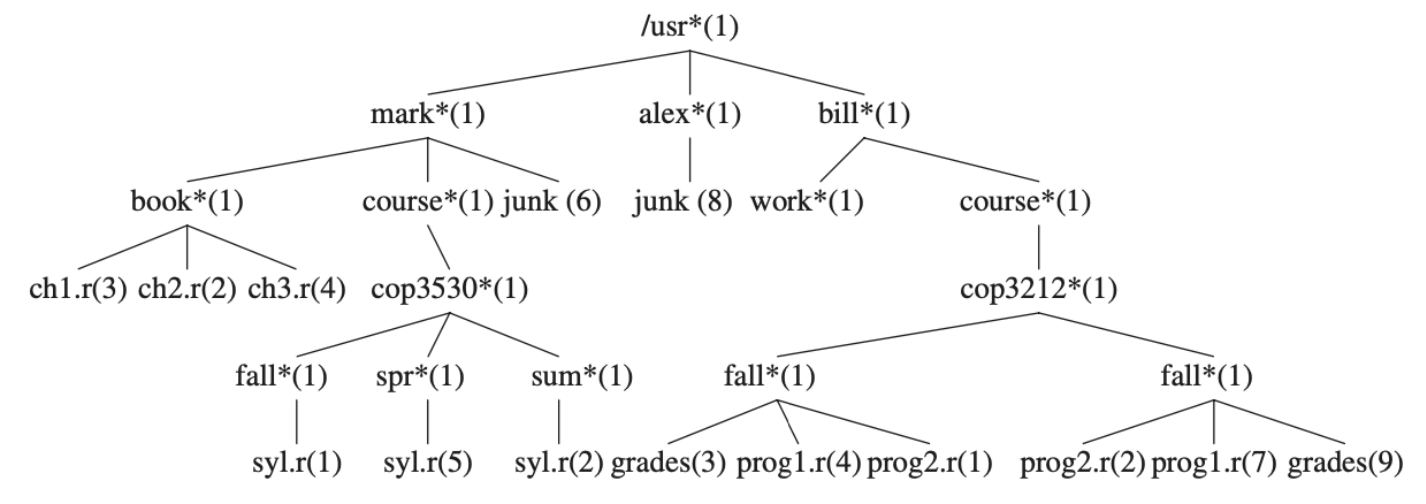



```
    void FileSystem::listAll( int depth = 0 ) const
    {
1       printName( depth );  // Print the name of the object
2       if( isDirectory( ) )
3           for each file c in this directory (for each child)
4               c.listAll( depth + 1 );
    }
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

TREE TRAVERSAL

- Postorder application: calculating the disk usage under each directory




```
int FileSystem::size( ) const
{
    int totalSize = sizeOfThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

     return totalSize;
}
```

```
            ch1.r            3
            ch2.r            2
            ch3.r            4
        book                10
                syl.r        1
            fall             2
                syl.r        5
            spr              6
                syl.r        2
            sum              3
        cop3530             12
    course                  13
    junk                     6
mark                        30
    junk                     8
alex                        9
    work                     1
                grades       3
                prog1.r      4
                prog2.r      1
            fall             9
                prog2.r      2
                prog1.r      7
                grades       9
            fall            19
        cop3212             29
    course                  30
bill                        32
/usr                        72
```

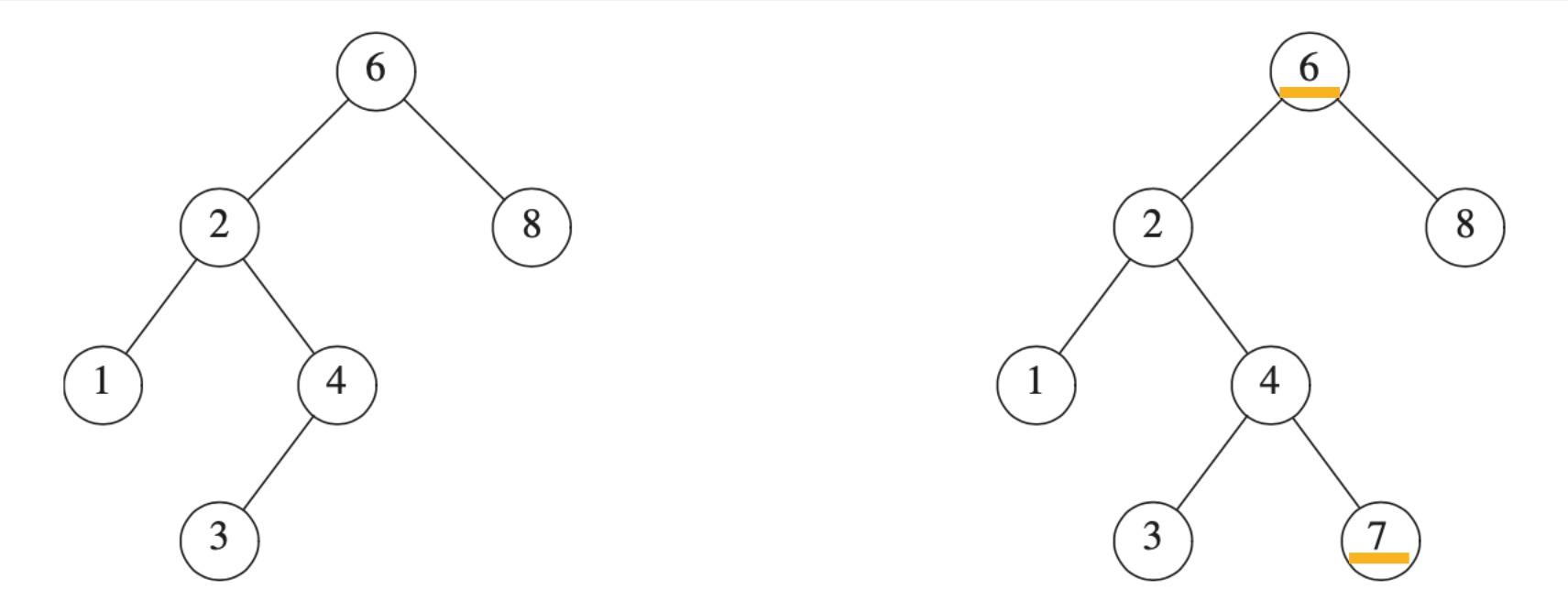
BINARY TREE

- A binary tree is a tree that each of its node can contain no more than two children

- An important application of the binary tree is to facilitate information search
  - suppose we have a list of ordered element
  - for each element, we will ensure that all elements that are smaller than it must be placed in its left subtree, and all elements that are larger than it must be placed in its right subtree (this property defines a **binary search tree**)
  - when we perform the search, we will start from the root, and recursively compare the search key with the current element; if the key is smaller, we will go deeper into the left subtree and vice versa

BINARY TREE

- The left tree is a binary search tree, while the right one is not. Note that in the right tree, 7 is larger than 6, but it is placed in the left subtree of 6.



BINARY SEARCH TREE: OPERATIONS

- The binary search tree (BST) ADT has three fundamental operations:
  - findMin: returns the smallest element in the BST
  - finxMax: returns the largest element in the BST
  - contains: searches against the BST to determine whether an element exists
  - insert: insert a new element into the BST
  - remove: remove an existing element from the BST

BINARY SEARCH TREE: OPERATIONS

- definition

```
struct BinaryNode
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

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

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

BINARY SEARCH TREE: OPERATIONS

- findMin() and findMax(): recursively find the lestmost or rightmost leaf node

```
1   /**
2    * Internal method to find the smallest item in a subtree t.
3    * Return node containing the smallest item.
4    */
5   BinaryNode * findMin( BinaryNode *t ) const
6   {
7       if( t == nullptr )
8           return nullptr;
9       if( t->left == nullptr )
10          return t;
11      return findMin( t->left );
12  }
```

```
1   /**
2    * Internal method to find the largest item in a subtree t.
3    * Return node containing the largest item.
4    */
5   BinaryNode * findMax( BinaryNode *t ) const
6   {
7       if( t != nullptr )
8           while( t->right != nullptr )
9               t = t->right;
10      return t;
11  }
```

BINARY SEARCH TREE: OPERATIONS

- contains(): this is essentially a binary search

```
1   /**
2    * Internal method to test if an item is in a subtree.
3    * x is item to search for.
4    * t is the node that roots the subtree.
5    */
6   bool contains( const Comparable & x, BinaryNode *t ) const
7   {
8       if( t == nullptr )
9           return false;
10      else if( x < t->element )
11          return contains( x, t->left );    // a recursive call
12      else if( t->element < x )
13          return contains( x, t->right );   // a recursive call
14      else
15          return true;    // Match
16  }
```

BINARY SEARCH TREE: OPERATIONS

- insert(): find the correct location to insert in a way similar to contains()

```
1   /**
2    * Internal method to insert into a subtree.
3    * x is the item to insert.
4    * t is the node that roots the subtree.
5    * Set the new root of the subtree.
6    */
7   void insert( const Comparable & x, BinaryNode * & t )
8   {
9       if( t == nullptr )
10          t = new BinaryNode{ x, nullptr, nullptr };
11      else if( x < t->element )
12          insert( x, t->left );       // a recursive call
13      else if( t->element < x )
14          insert( x, t->right );      // a recursive call
15      else
16          ;  // Duplicate; do nothing
17  }
```

BINARY SEARCH TREE: OPERATIONS

- remove():

```
1   /**
2    * Internal method to remove from a subtree.
3    * x is the item to remove.
4    * t is the node that roots the subtree.
5    * Set the new root of the subtree.
6    */
7   void remove( const Comparable & x, BinaryNode * & t )
8   {
9       if( t == nullptr )
10          return;   // Item not found; do nothing
11      if( x < t->element )
12          remove( x, t->left );      // a recursive call
13      else if( t->element < x )
14          remove( x, t->right );     // a recursive call
15      else if( t->left != nullptr && t->right != nullptr ) // Two children
16      {
17          t->element = findMin( t->right )->element;
18          remove( t->element, t->right );
19      }
20      else
21      {
22          BinaryNode *oldNode = t;
23          t = ( t->left != nullptr ) ? t->left : t->right;
24          delete oldNode;
25      }
26  }
```

// if the node to be removed has two children, use the smallest element in the right subtree to replace the current node

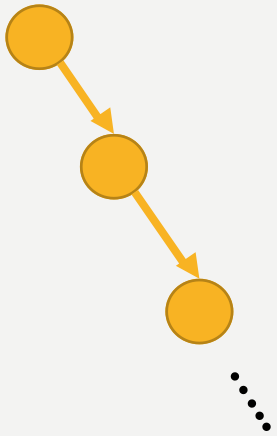
// if the node to be removed has only one child, directly link the node's parent to its child

BINARY SEARCH TREE: OPERATIONS

- Expected time complexity: how many steps do we expect to take before hitting the correct location of the tree (the traveled path length)?
  - if we pick a random element as the root, we can expect that half of the elements are going to be smaller than it and half of the elements are going to be larger than it. In this case, we can expect that the left subtree has the same size as the right subtree
  - we can make this assumption recursively for each internal node, so we get a tree that is as "wide" as possible, where the number of nodes in each layer doubles as we go deeper
  - in this case, we can expect of tree depth of O(log(n))
  - more formal analysis will be introduced later in the class

BINARY SEARCH TREE: THE WORST-CASE SCENARIO

- The tree can be very deep and degenerates to a linked list if we are adding in sorted elements
  - e.g., the second element is larger than the first element, so we add the second element as the right child of the first element
  - the third element is larger than the second element, so we add the third element as the right child of the second element
  - …..



BINARY SEARCH TREE: THE WORST-CASE SCENARIO

- In this case, the expected O(log(n)) search/insert/remove time becomes O(n)

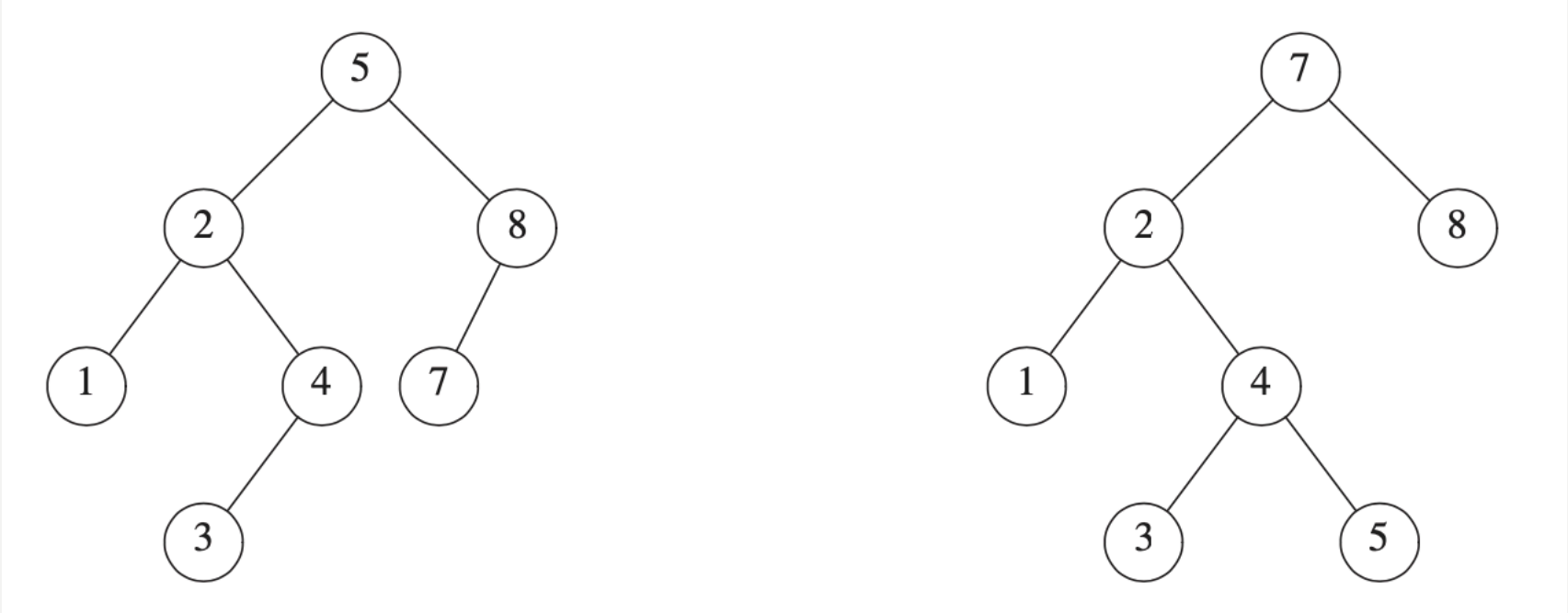
- To avoid the worst-case scenario, we can perform additional operations (during tree construction) to alter the tree topology, and make the tree as "wide" and as "flat" as possible (we call such a tree **balanced**)

BINARY SEARCH TREE: THE AVL TREE

- AVL is due to Adelson, Velskii, and Landis
- They proposed a balance property for binary search tree (and ways to maintain this property)
  - the AVL property: for each node, the height of its left subtree and the height of its right subtree cannot differ by more than 1
- A binary search tree that satisfies the AVL property is called an AVL tree
- An AVL tree guarantees O(log(n)) search/insert/remove time

BINARY SEARCH TREE: THE AVL TREE

- The left tree is an AVL tree, the right one is not.
  - consider node 7 (the root), its left tree depth is 3, while its right tree depth is 1



BINARY SEARCH TREE: THE AVL TREE

- We will need to maintain the AVL property when we alter the topology of the tree
  - insertion and deletion will modify the topology
  - we will use insertion cases as examples to illustrate how to maintain the AVL property

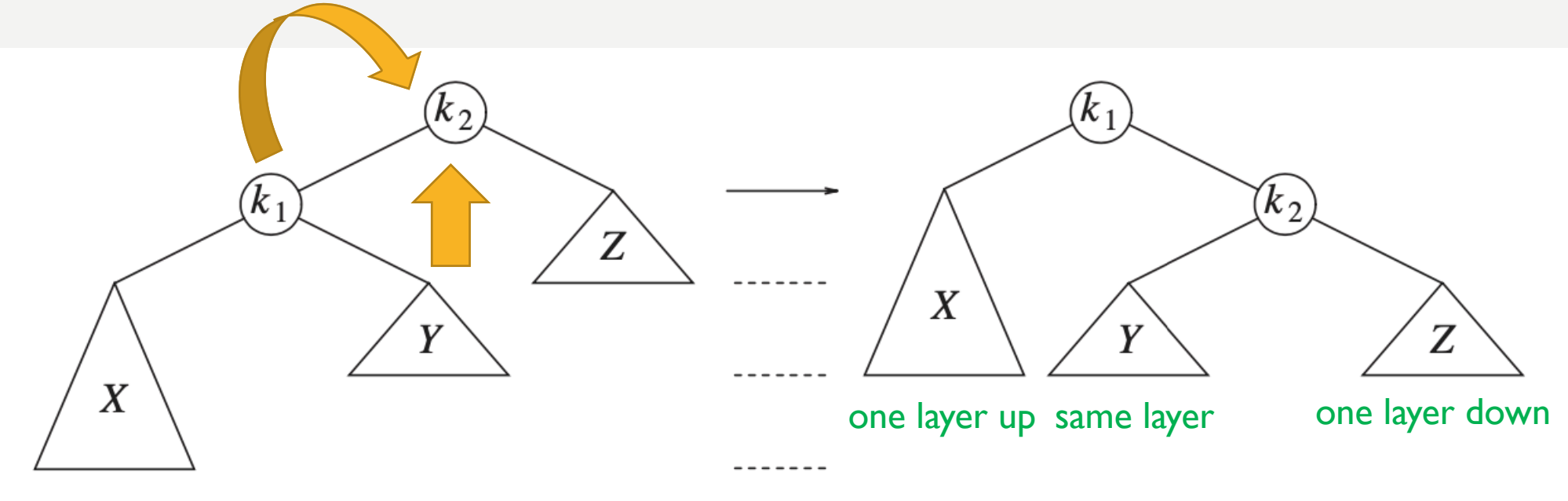
- Given a node α, There are 4 insertion cases we need to consider:
  - we are inserting to the left subtree of the left child of α
  - we are inserting to the right subtree of the left child of α
  - we are inserting to the left subtree of the right child of α
  - we are inserting to the right subtree of the right child of α
  - the first and fourth cases are essentially the same (symmetric), and the second and the third cases are essentially the same as well

BINARY SEARCH TREE: THE AVL TREE

- For the first and fourth cases, we will use a simpler "**single rotation**" approach
- For the second and third cases, we will use a slightly more sophisticated "**double rotation**" approach

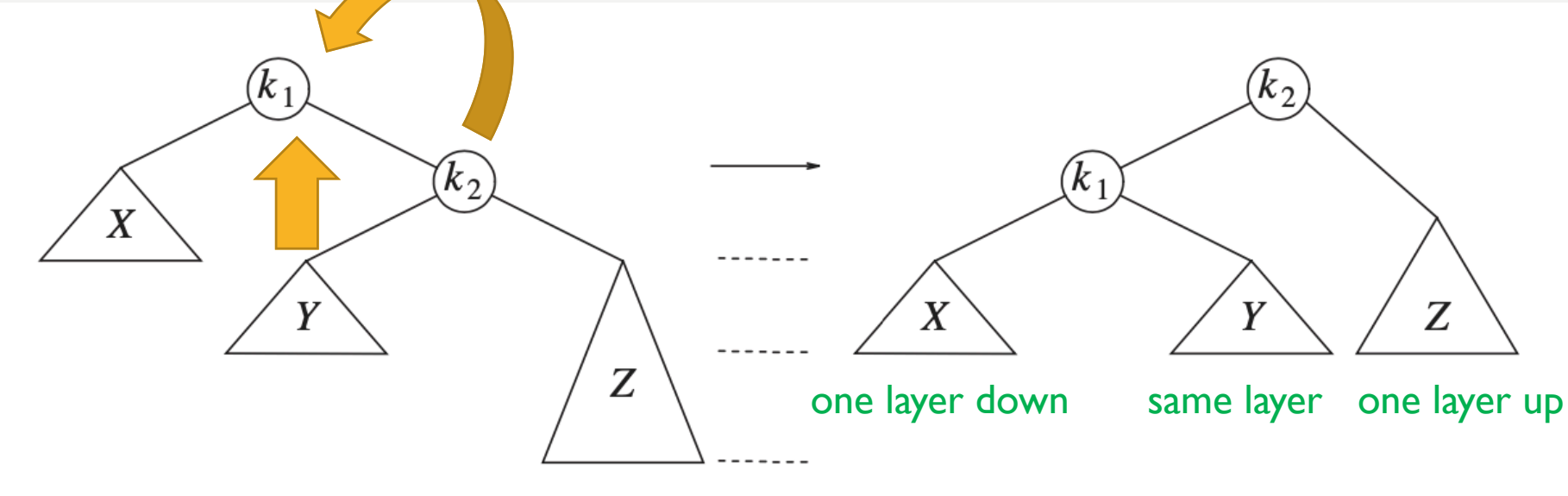
AVL TREE: SINGLE ROTATION

- Consider case 1, where we are inserting an element into the subtree X; it increases the height of X by 1 and results in the violation of the AVL property at the node k2
- We will rotate k1 and k2 clockwise, and detach the subtree Y (if non-empty) from k1 and reattach it to k2



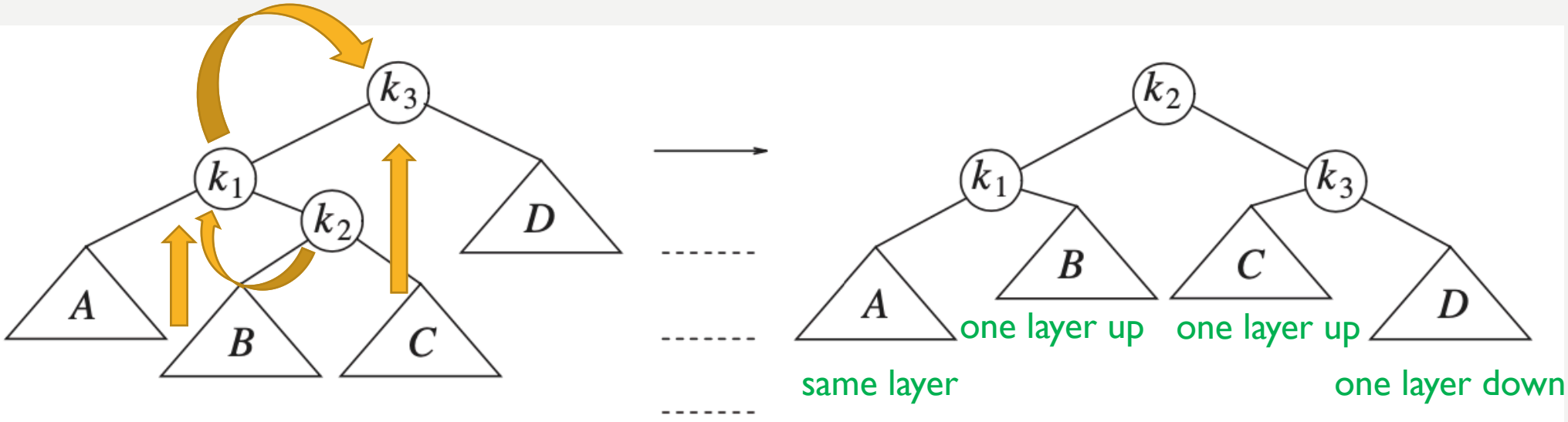
AVL TREE: SINGLE ROTATION

- Consider case 4, where we are inserting an element into the subtree Z; it increases the height of Z by 1 and results in the violation of the AVL property at the node k1
- We will rotate k1 and k2 anti-clockwise, and detach the subtree Y (if non-empty) from k2 and reattach it to k1



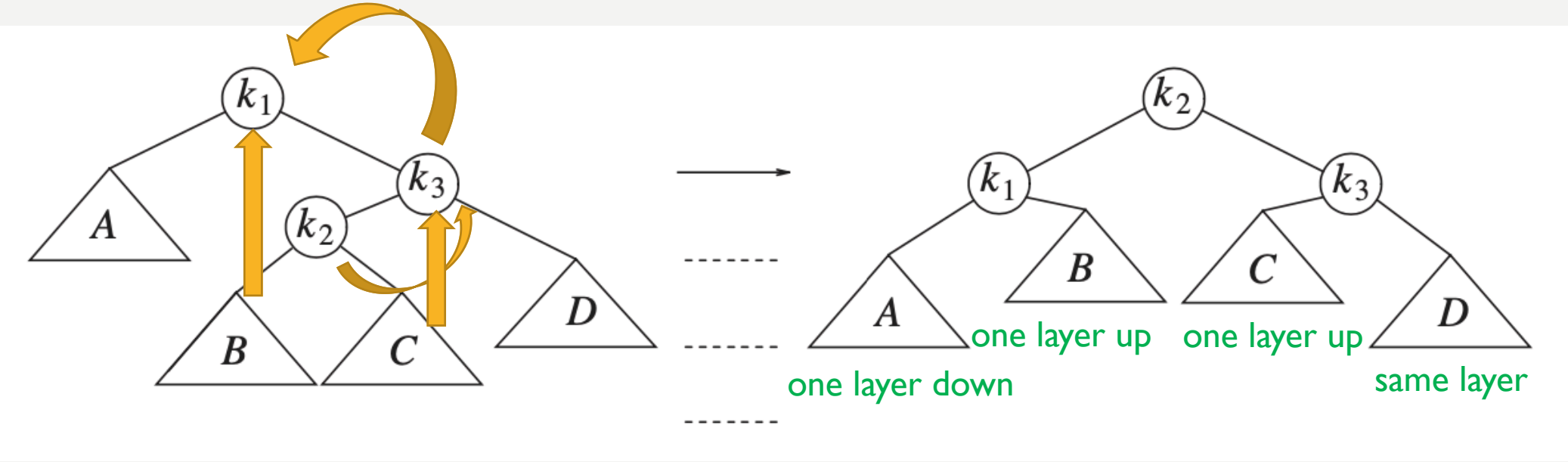
AVL TREE: DOUBLE ROTATION

- Consider case 2, where we are inserting an element into the subtree B or C; it increases either of their heights by 1 and results in the violation of the AVL property at the node k3
- We will rotate k1 and k2 clockwise, and then rotate k2 and k3 clockwise (hence double rotation). We will also detach the subtrees B and C (if non-empty) from k2, reattach B to k1 and C to K3.



AVL TREE: DOUBLE ROTATION

- Consider case 3, where we are inserting an element into the subtree B or C; it increases either of their heights by 1 and results in the violation of the AVL property at the node k1
- We will rotate k1 and k2 clockwise, and then rotate k2 and k3 clockwise (hence double rotation). We will also detach the subtrees B and C (if non-empty) from k2, reattach B to k1 and C to K3.



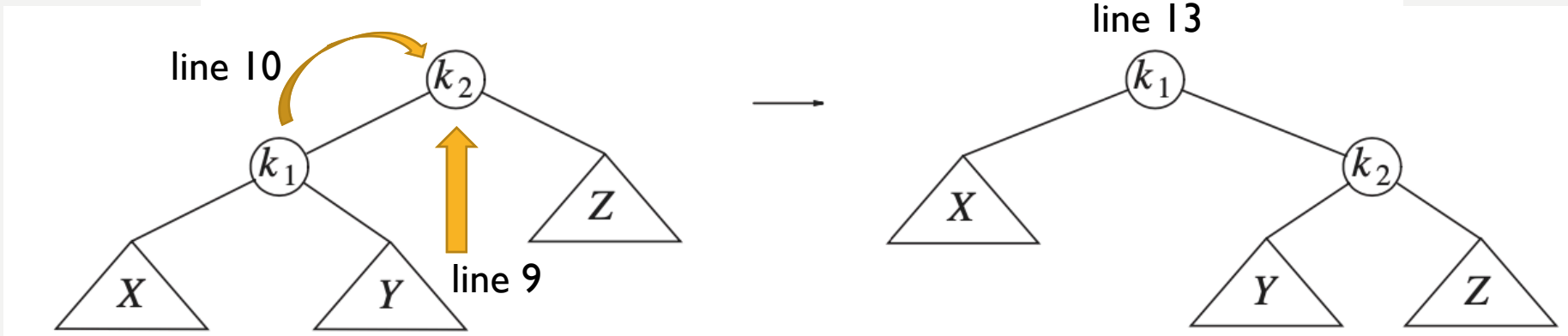
AVL TREE: IMPLEMENTATION

- We will be recording the height of each node for AVL tree, while the other components stay the same

```
1   struct AvlNode
2   {
3       Comparable element;
4       AvlNode   *left;
5       AvlNode   *right;
6       int       height;    // now adding subtree height
7
8       AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
9         : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10
11      AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12        : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13  };
```

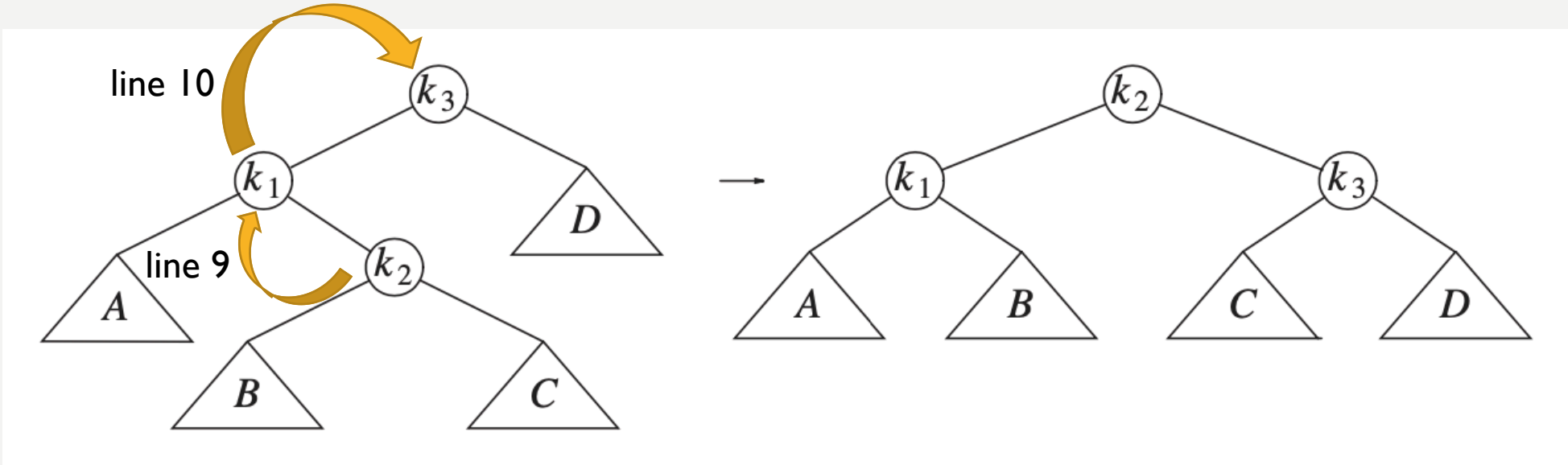
AVL TREE: IMPLEMENTATION

```
1   /**
2    * Rotate binary tree node with left child.
3    * For AVL trees, this is a single rotation for case 1.
4    * Update heights, then set new root.
5    */
6   void rotateWithLeftChild( AvlNode * & k2 )
7   {
8       AvlNode *k1 = k2->left;
9       k2->left = k1->right;
10      k1->right = k2;
11      k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12      k1->height = max( height( k1->left ), k2->height ) + 1;
13      k2 = k1;
14  }
```



AVL TREE: IMPLEMENTATION

```
 7   void doubleWithLeftChild( AvlNode * & k3 )
 8   {
 9       rotateWithRightChild( k3->left );
10       rotateWithLeftChild( k3 );
11   }
```



AVL TREE: IMPLEMENTATION

- balance the subtree rooted at t

```
21  // Assume t is balanced or within one of being balanced
22  void balance( AvlNode * & t )
23  {
24      if( t == nullptr )
25          return;
26
27      if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )   // if the AVL property is violated
28          if( height( t->left->left ) >= height( t->left->right ) )      // case 1
29              rotateWithLeftChild( t );
30          else                                                           // case 2
31              doubleWithLeftChild( t );
32      else
33      if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )   // if the AVL property is violated
34          if( height( t->right->right ) >= height( t->right->left ) )    // case 4
35              rotateWithRightChild( t );
36          else
37              doubleWithRightChild( t );                                 // case 3
38
39      t->height = max( height( t->left ), height( t->right ) ) + 1;
40  }
```

AVL TREE: IMPLEMENTATION

```
 7   void insert( const Comparable & x, AvlNode * & t )
 8   {
 9       if( t == nullptr )                                  // locate where to insert the node
10           t = new AvlNode{ x, nullptr, nullptr };
11       else if( x < t->element )
12           insert( x, t->left );
13       else if( t->element < x )
14           insert( x, t->right );
15
16       balance( t );                                       // re-balance the tree if necessary
17   }
```

AVL TREE: IMPLEMENTATION

```
7   void remove( const Comparable & x, AvlNode * & t )
8   {
9       if( t == nullptr )
10          return;   // Item not found; do nothing
11
12      if( x < t->element )
13          remove( x, t->left );
14      else if( t->element < x )
15          remove( x, t->right );
16      else if( t->left != nullptr && t->right != nullptr )  // Two children
17      {
18          t->element = findMin( t->right  )->element;
19          remove( t->element, t->right );
20      }
21      else
22      {
23          AvlNode *oldNode = t;
24          t = ( t->left != nullptr ) ? t->left : t->right;
25          delete oldNode;
26      }
27
28       balance( t );      // add rebalancing
29  }
```

// deleting a node could make a subtree shorter, which is similar to make its sibling higher; so we can apply rebalancing in a similar way if necessary

B-TREE: MOTIVATION

- Now we have established that the advantage of using the tree ADT for information storage and retrieval will allow O(log(n)) time
  - and we can achieve this purely through following the edges, without spending time to compute the hash function as in the hash table ADT

- More precisely, if we are using a BST, the search/insert/delete time complexity is O(log2(n))

B-TREE: MOTIVATION

- In practice, information in the tree is not stored in the main memory, it is stored in the hard disk (because we do not have that much memory to hold all information).
- Accessing hard disk is much slower than accessing the main memory. To facilitate faster access, related information is often put into adjacent disk blocks and are loaded into the main memory altogether in a single access (called prefetch). The underlying rationale is assuming related information is often accessed altogether.
- For the tree ADT, data stored in the same node can be considered as related information and can be retrieved in a single hard disk access.

B-TREE: MOTIVATION

- As a result, each time we follow an edge to locate the tree element for search/insertion/deletion, we will incur a hard disk access. The expected number of hard disk access is O(log2(n)).

- An intuitive way to reduce the number of hard disk access is to increase the number of children each node can have. For example, if we allow M children per node, we can expect the number of hard disk access being reduced to O(logM(n)).

- Indeed, finding the correct edge to follow among M edges is more challenging than finding the correct one between 2 edges. However, we are willing to pay the extra computation as the benefit we get from reducing hard disk access is worthy.

B-TREE: DEFINITION

- Such a M-ary search tree allows M-way branching. We ensure the following properties to make it reasonably balanced, which ensures the expected O(logM(n)) search/insert/delete time.

- The properties we will ensure:
  - the data items are stored in leaves
  - the non-leaf nodes store up to M-1 keys to guide the searching; key i represents the smallest key in subtree $i + 1$
  - the root is either a leaf or has between 2 and M children
  - all non-leaf nodes (except the root) have between ceiling(M/2) to M children
  - all leaf nodes are at the same depth and have between ceiling(L/2) and L data items

- An M-ary search tree that satisfies the above properties is called a **B-tree**.

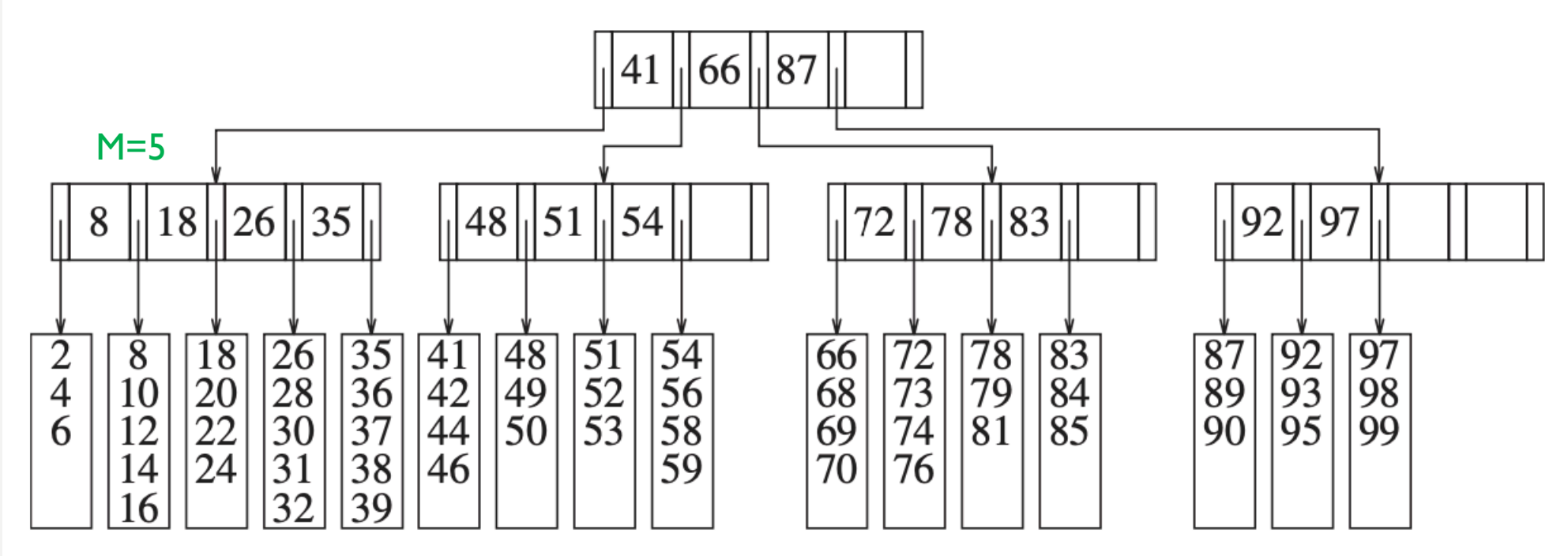
B-TREE: DEFINITION

- While examine the B-tree properties, we found:
  - the tree is only a bit worse than half-full (by requiring all internal nodes to have at least ceiling(M/2) children and all leaf nodes to have at least ceiling(L/2) data items)
  - the tree is balanced (by requiring all leaf nodes to have the same depth)

- We can therefore expect the O(logM(n)) search/insert/delete time.

B-TREE: DEFINITION

- A B-tree example with M=5 and L=5:



B-TREE: OPERATIONS

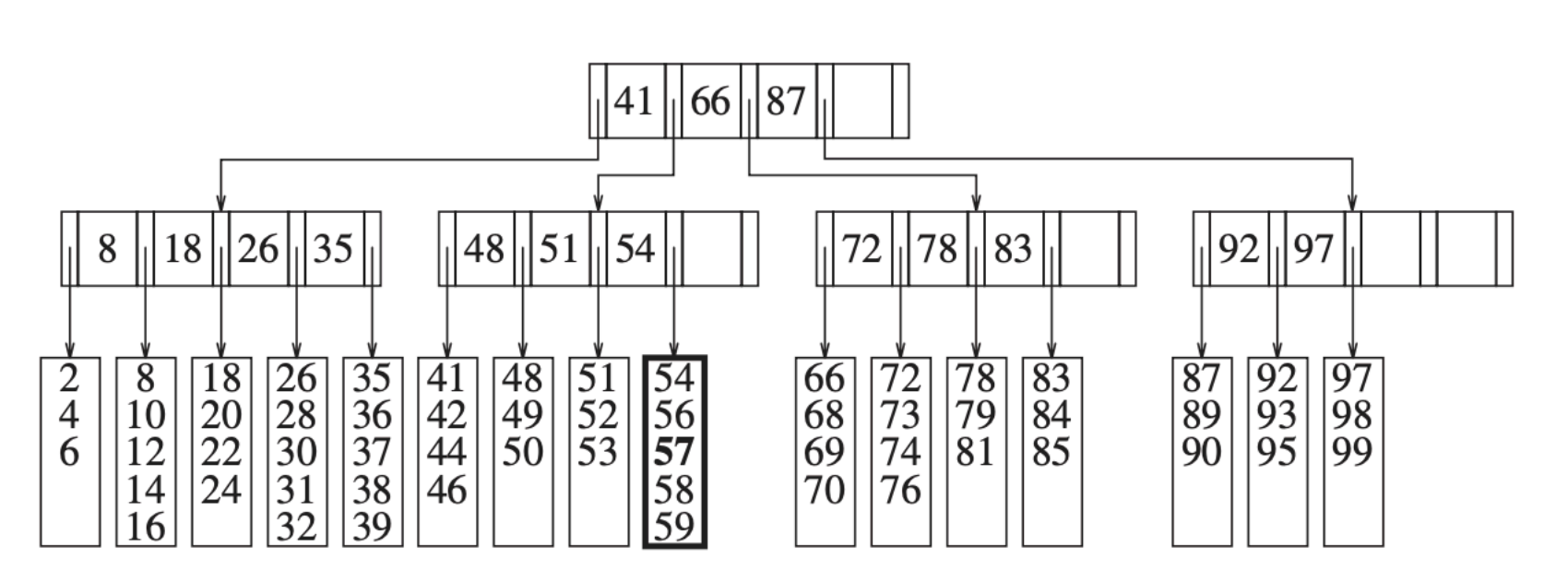
- Search
  - traverse down the B-tree in a similar way as the BST traversal
  - except that we need to compare they search key with each of the M values to determine the correct edge to follow
  - this is OK because all of the M values are stored in an adjacent location of the hard disk and can be retrieved with only one access
  - once we get to the leaf node, compare the key with each data item to complete the search

B-TREE: OPERATIONS

- insert
  - use the search procedure to find the data block to insert
  - however the insertion may change the tree topology, and we need to maintain all B-tree properties
    - case 1: no split needed
    - case 2: one-level split
    - case 3: recursive split

B-TREE: OPERATIONS

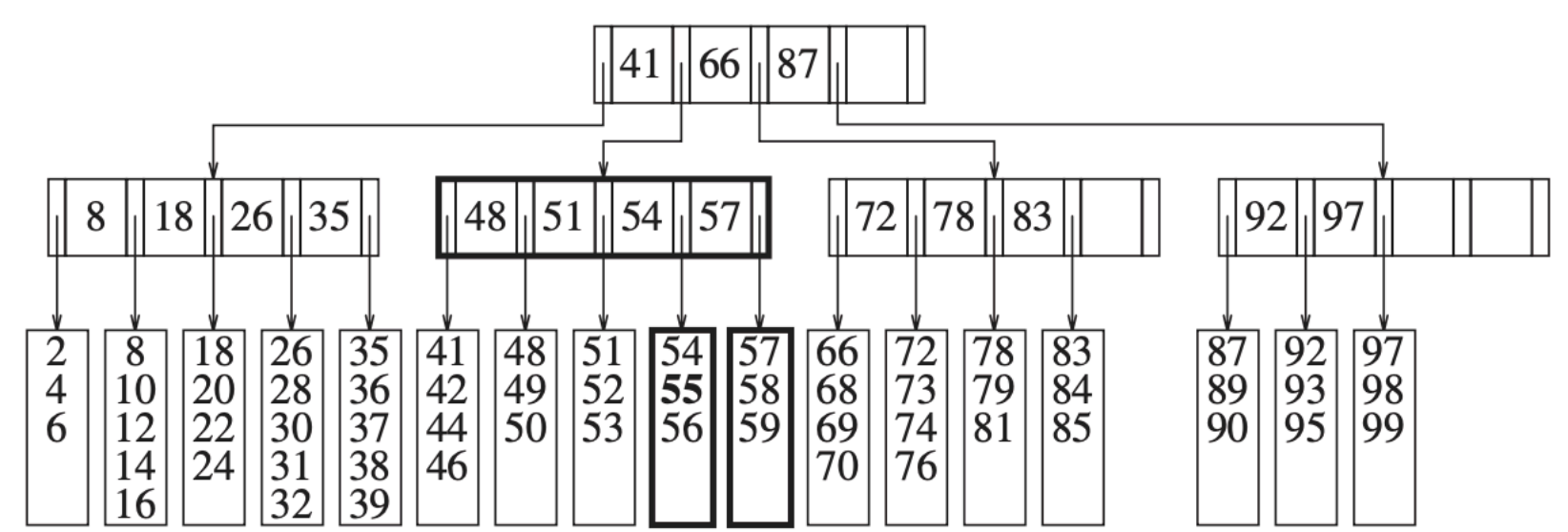
- insertion: if the current data block still has space, no node split is needed



insertion of 57

B-TREE: OPERATIONS

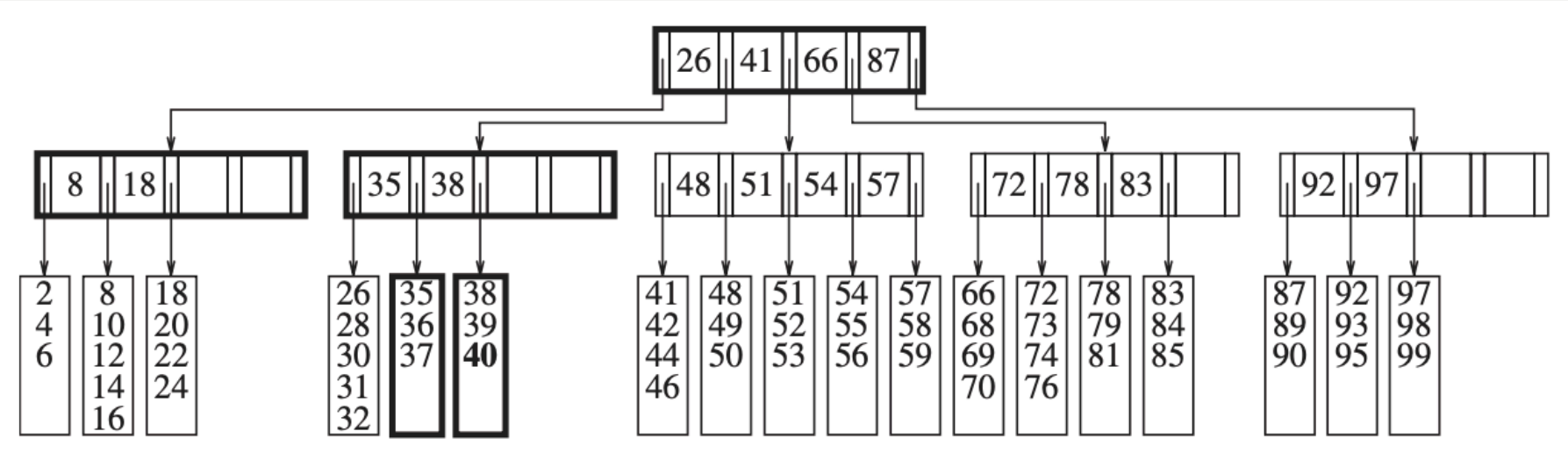
- insertion: if the current data block is full, we will split the data block and add a new child to its parent



1: Insertion of 55, the block becomes full.
2: The block is split into 2, the new block is attached to the parent.
3: The two split blocks each contains ceiling(L/2) data items.
4: The parent contains no more than M items.

B-TREE: OPERATIONS

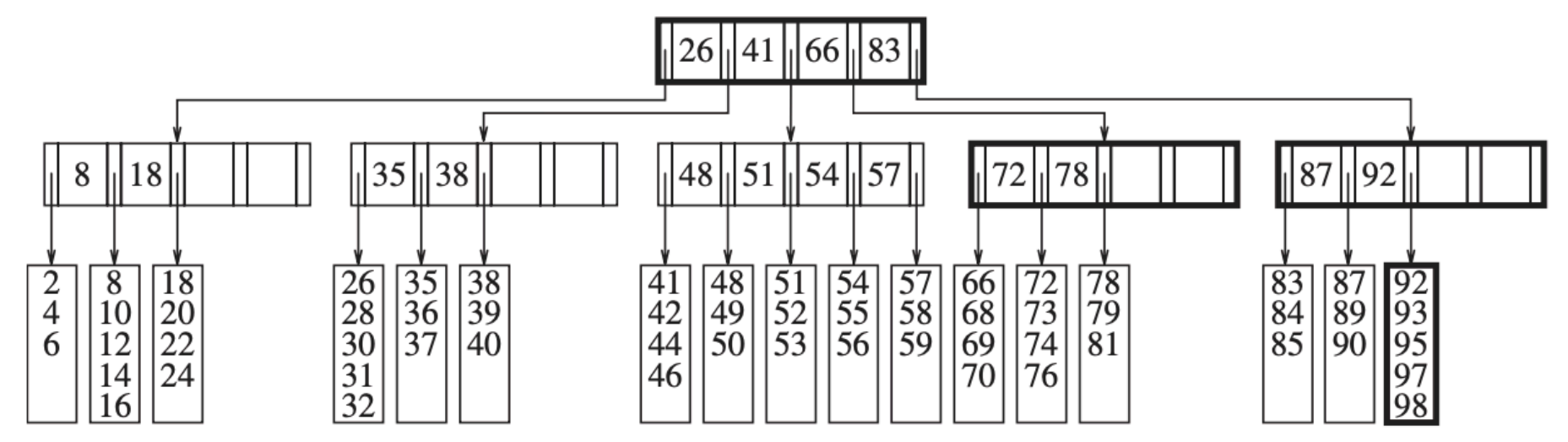
- insertion: if the parent is also full, we will split parent block and do it recursively



1: Insertion of 40, the block becomes full.
2: The block is split into 2, the new block is attached to the parent.
3: The parent is also full, the parent is split into 2.
4: The newly-split parents are further attached to the grandparent.
5: If we need to split the root, we will create a new root as the parent of the two split root. (Note that the root can contain as few as 2 children.)

B-TREE: OPERATIONS

- deletion: the idea is similar to insertion; we will perform (recursive) merge instead of split if necessary



1: Deletion of 99, the block contains less than ceiling(L/2) items.
2: Merge it with its previous data block.
3: The parent contains less than ceiling(M/2) data items.
4: Borrow a data block from its sibling.

C++ STL

- **std::map** and **std::set** implement the search tree ADT.
- std::map stores key-value pairs (similar to std::unordered_map)
  - faster traversal, slower search/insert/delete in general as compared to std::unordered_map
- std::set stores single values (similar to std::unordered_set)
  - faster traversal, slower search/insert/delete in general as compared to std::unordered_set

SUMMARY

- The tree ADT is an important data structure for modeling many natural relationships.
- It further supports O(log(n)) time search/insert/delete with a small associated constant.
  - linked list requires O(n) time with a small constant
  - hash table requires O(1) time with a large constant
- The property is ensured if the tree is balanced
  - AVL trees: binary search tree that satisfies the AVL property. We use rotation to ensure the AVL property in cases of insertion and deletion.
  - B-tree: hard disk-friendly data structure. Essentially an M-ary search tree. We use split and merge to ensure the B-tree property in cases of insertion and deletion.
  - Splay tree (not discussed in-class but available in the textbook): bring elements close to the root such that the subsequent access of them can be made faster (like bubbling up the nodes up the tree). Study if you are interested; will not test it in the exams.