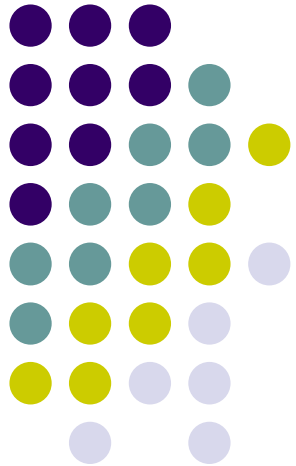
Mastering Data Structures and Algorithms: A Practical Approach

Trees and Graphs

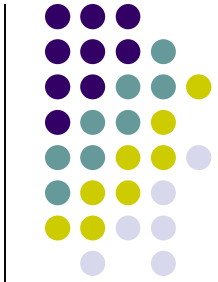


By Dr. Juan C. Gomez

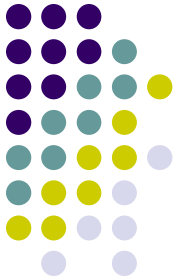
Fall 2018

UC Santa Cruz Extension in Silicon Valley

Overview



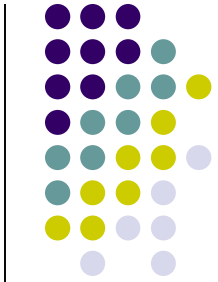
- Trees vs Graphs
- Terminology
- Types of Trees:
  - Generic Tree vs Binary tree
  - Binary Tree vs Binary Search tree
  - Balanced vs Unbalanced tree
  - Complete Binary Tree
  - Full Binary Tree
  - Perfect binary tree
- Tree Traversals
  - In-order
  - Pre-order
  - Post-order



Overview

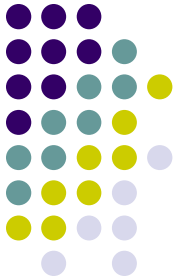
- Tries (Prefix trees)
- Trees is Java
- Trees in C++
- Well-known Tree Data Structures:
  - AVL Trees: O(log(N))
  - Splay Trees: Amortized O(log(N))
  - B-Trees

Overview



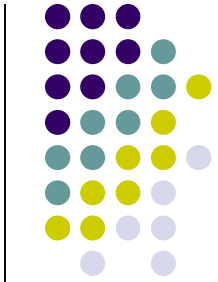
- Graphs
  - Concept
  - Adjacency lists
  - Adjacency Matrix
  - Directed vs Undirected graphs
  - Graphs with weights
  - Graph traversals
    - Depth-First Search
    - Breadth-First Search
    - Bidirectional Search

Overview



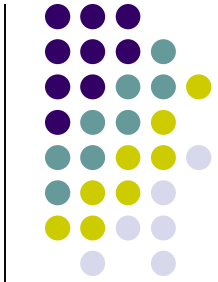
- Well known graph algorithms:
  - Topological Sort
  - Dijkstra's Shortest Path Algorithm

Trees: Trees vs Graphs



- Trees are a Subset of Graphs
  - Acyclic Graphs
- Tree Terminology:
  - Root node
  - Leaf Node
  - Inner Node
  - Tree height
  - Node depth

Trees: Trees vs Graphs



- Tree Terminology:
  - Parent node
  - Child Node
  - Leaf Node
  - Path
  - Ancestor
  - Descendant

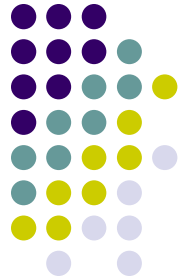
Trees: Implementation

- Generic Implementation

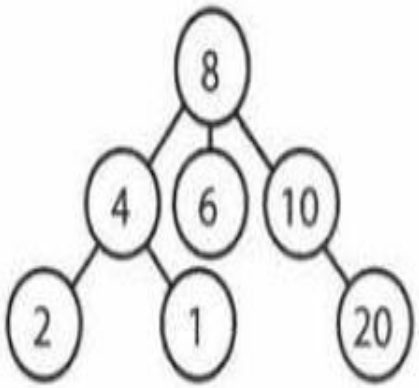
```
1  class Node {
2     public String name;
3     public Node[] children;
4  }
```

```
1  class Tree {
2     public Node root;
3  }
```

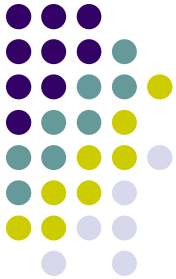
Trees: Generic vs Binary Trees



A binary tree is a tree in which each node has up to two children. Not all trees are binary trees. For example, this tree is not a binary tree. You could call it a ternary tree.

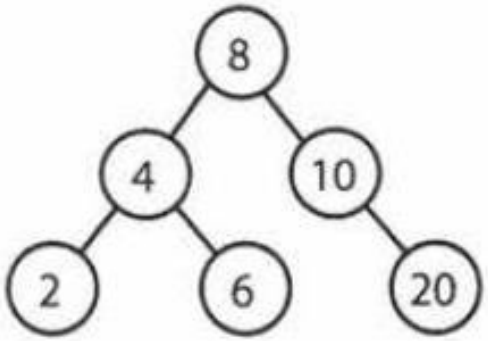


Trees: Binary vs Binary Search Trees

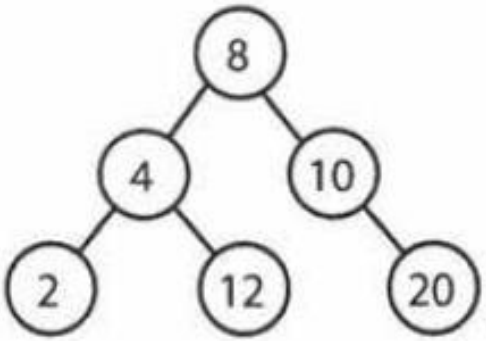


A binary search tree is a binary tree in which every node fits a specific ordering property: `all left descendents <= n < all right descendents`. This must be true for each node n.

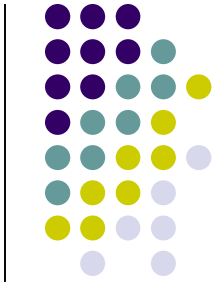
A binary search tree.



Not a binary search tree.



Trees: Balanced vs Unbalanced Trees

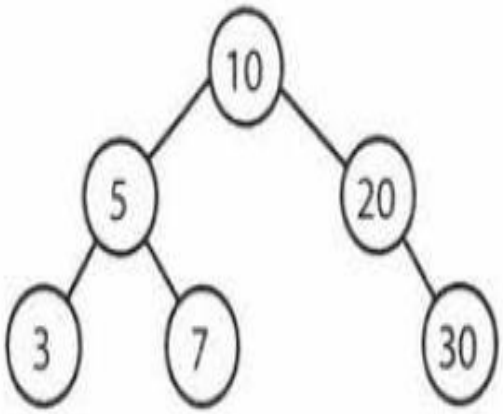


- Balanced Trees:
  - Not perfectly balanced, but balanced enough to ensure O(log(N))
  - Usually differences are in the last level
  - Well known balanced trees:
    - AVL
    - Red-Black

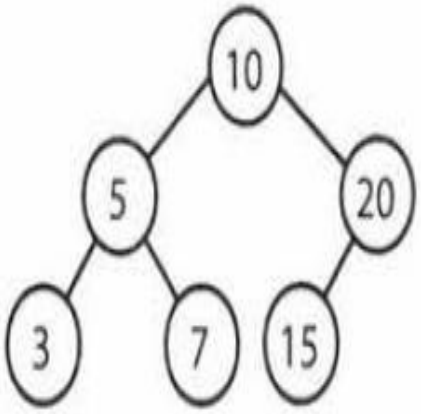
Trees: Complete Binary Tree

A complete binary tree is a binary tree in which every level of the tree is fully filled, except for perhaps the last level. To the extent that the last level is filled, it is filled left to right.

not a complete binary tree



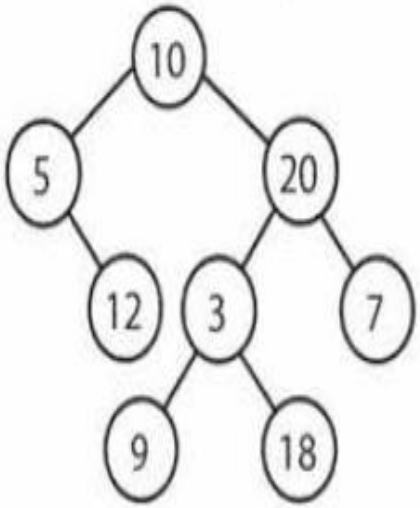
a complete binary tree



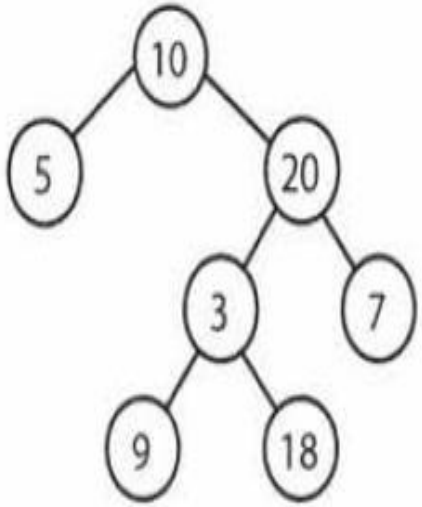
Trees: Full Binary Tree

A full binary tree is a binary tree in which every node has either zero or two children. That is, no nodes have only one child.

not a full binary tree



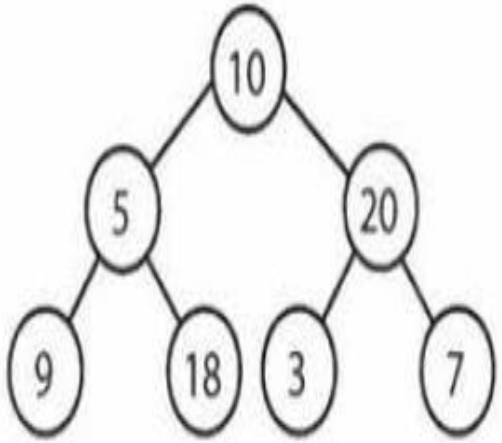
a full binary tree



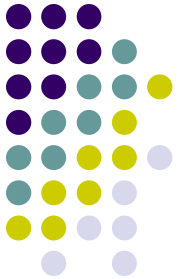
Trees: Perfect Binary Tree

Perfect Binary Trees

A perfect binary tree is one that is both full and complete. All leaf nodes will be at the same level, and this level has the maximum number of nodes.

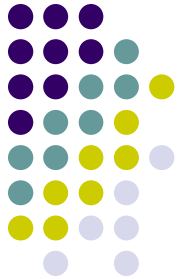


Trees: Tree Traversals



- Depth-first Search:
  - In-order
  - Pre-order
  - Post-Order
- Breadth-first Search:
  - Level-Order

Trees: In-Order Traversal

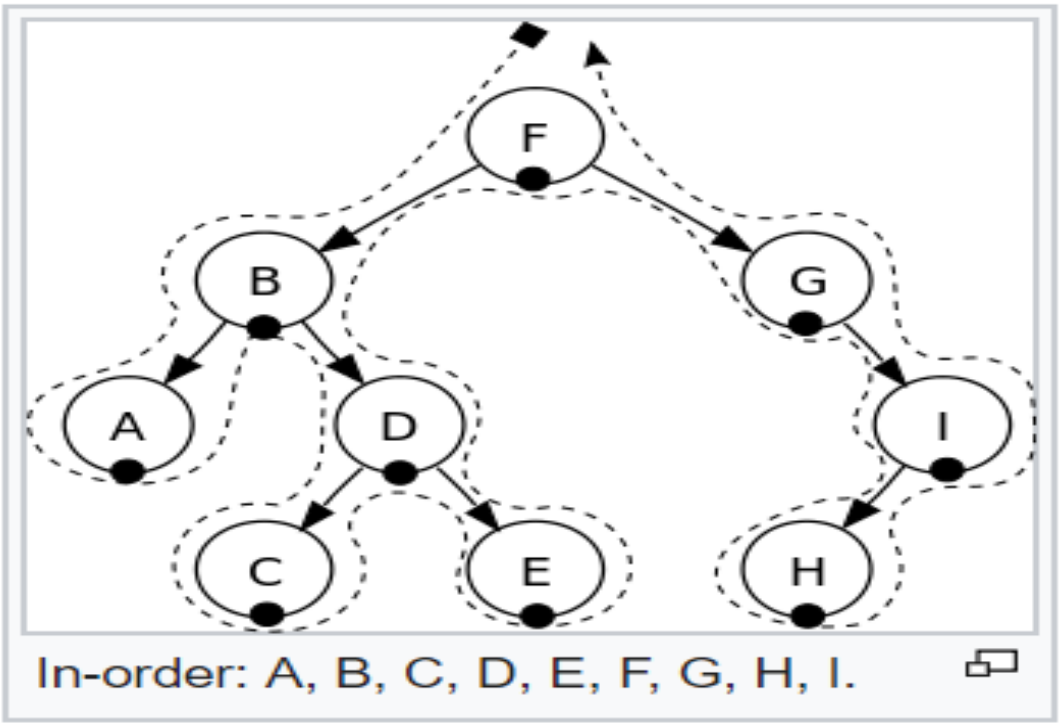


In-order traversal means to "visit" (often, print) the left branch, then the current node, and finally, the right branch.

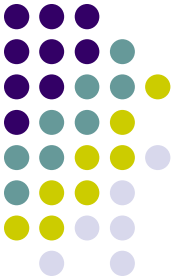
```
1  void inOrderTraversal(TreeNode node) {
2    if (node != null) {
3      inOrderTraversal(node.left);
4      visit(node);
5      inOrderTraversal(node.right);
6    }
7  }
```

When performed on a binary search tree, it visits the nodes in ascending order (hence the name "in-order").

Trees: In-Order Traversal

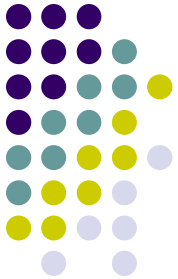


Trees: In-Order Traversal



```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

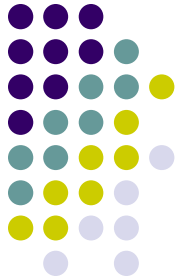
Trees: In-Order Traversal



```
/* Iterative function for inorder tree
   traversal */
void inOrder(struct Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
           curr Node */
        while (curr !=  NULL)
        {
            /* place pointer to a tree node on
               the stack before traversing
              the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
           left subtree.  Now, it's right
           subtree's turn */
        curr = curr->right;

    } /* end of while */
}
```

```
/* A binary tree Node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
    Node (int data)
    {
        this->data = data;
        left = right = NULL;
    }
};
```

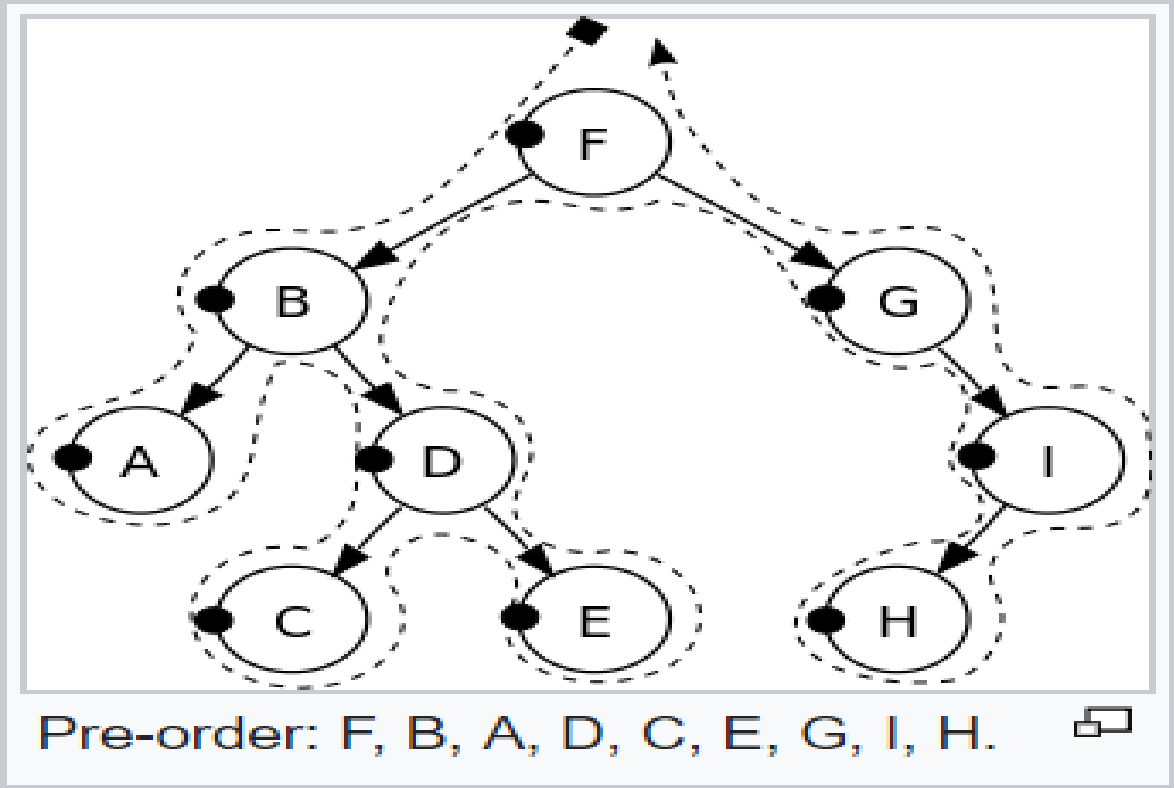
Trees: Pre-order Traversal



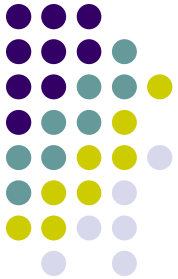
Pre-order traversal visits the current node before its child nodes (hence the name "pre-order").

```
1  void preOrderTraversal(TreeNode node) {
2     if (node != null) {
3        visit(node);
4        preOrderTraversal(node.left);
5        preOrderTraversal(node.right);
6     }
7  }
```

Trees: Pre-order Traversal



Trees: Pre-order Traversal



1) Create an empty stack *nodeStack* and push root node to stack.

2) Do following while *nodeStack* is not empty.

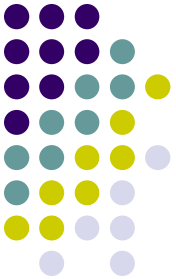
....**a)** Pop an item from stack and print it.

....**b)** Push right child of popped item to stack

....**c)** Push left child of popped item to stack

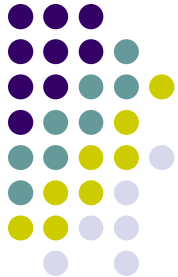
Right child is pushed before left child to make sure that left subtree is processed first.

Trees: Pre-order Traversal



```
// An iterative process to print preorder traversal of Binary tree
void iterativePreorder(node *root)
{
    // Base Case
    if (root == NULL)
       return;

    // Create an empty stack and push root to it
    stack<node *> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one. Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
    Note that right child is pushed first so that left is processed first */
    while (nodeStack.empty() == false)
    {
        // Pop the top item from stack and print it
        struct node *node = nodeStack.top();
        printf ("%d ", node->data);
        nodeStack.pop();

        // Push right and left children of the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);
    }
}
```

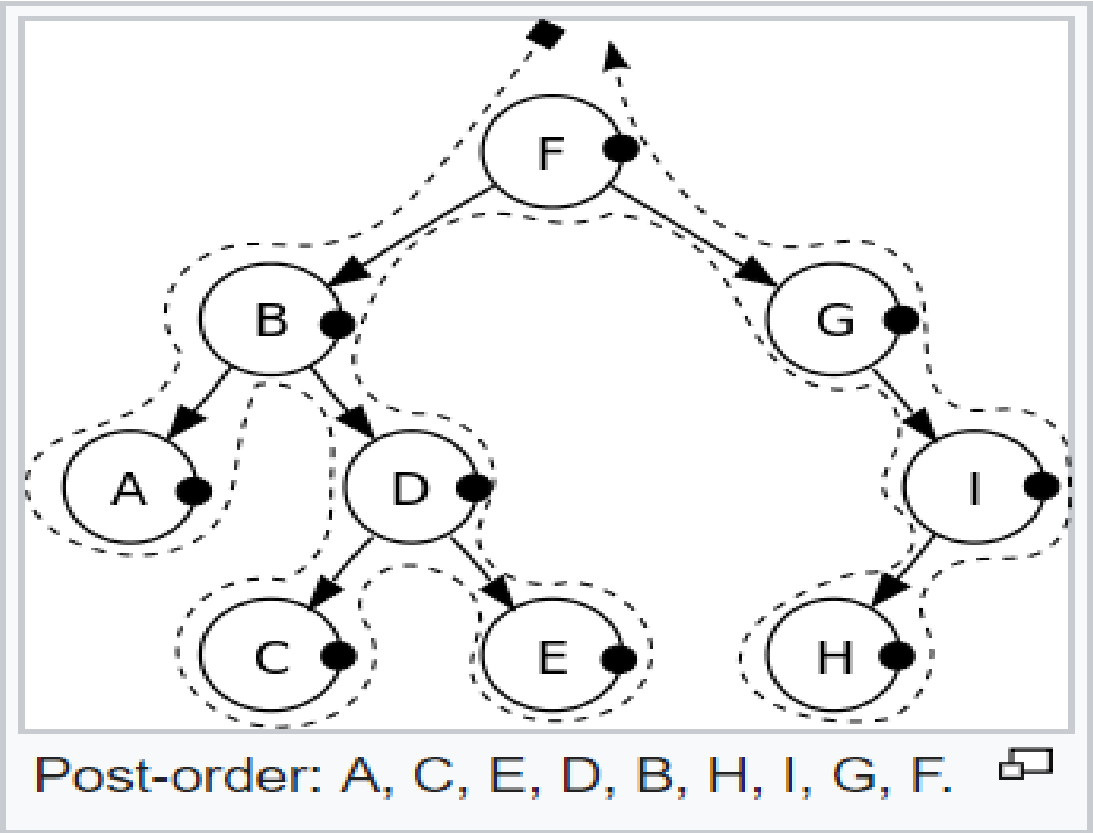
Trees: Post-order Traversal



Post-order traversal visits the current node after its child nodes (hence the name "post-order").

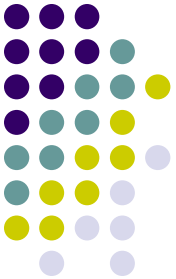
```
1  void postOrderTraversal(TreeNode node) {
2    if (node != null) {
3      postOrderTraversal(node.left);
4      postOrderTraversal(node.right);
5      visit(node);
6    }
7  }
```

Trees: Post-order Traversal



UC Santa Cruz Extension in Silicon Valley

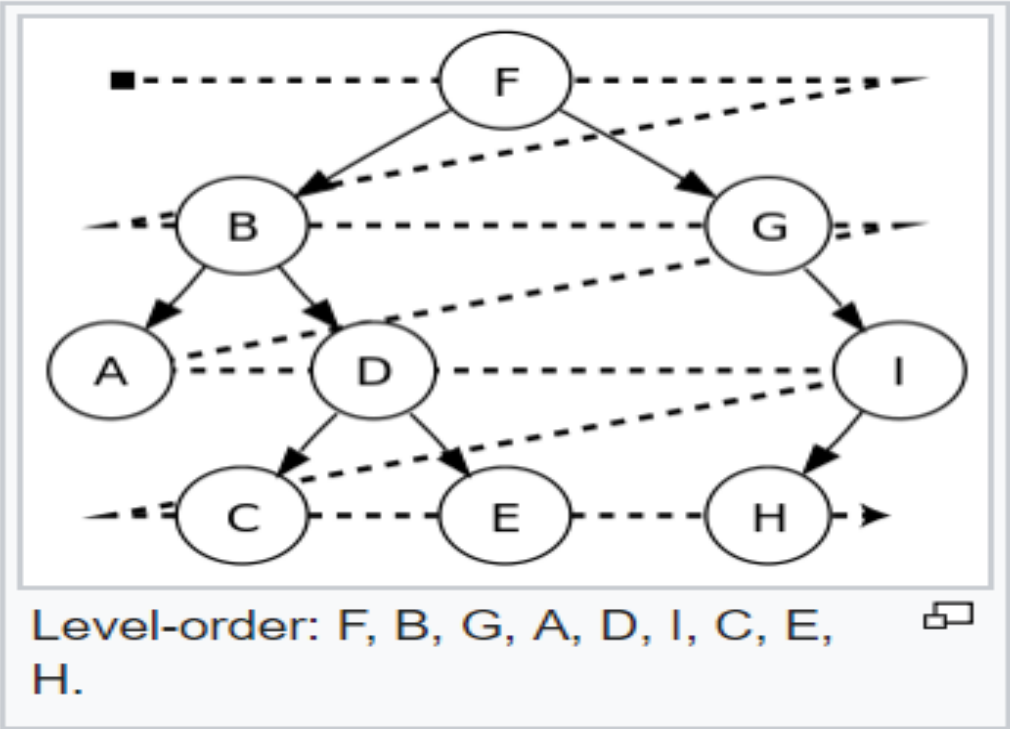
Trees: Post-order Traversal



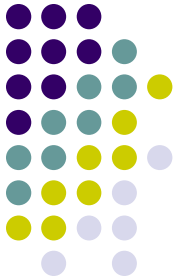
```
iterativePostorder(node)
  s ← empty stack
  lastNodeVisited ← null
  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      peekNode ← s.peek()
      // if right child exists and traversing node
      // from left child, then move right
      if (peekNode.right ≠ null and lastNodeVisited ≠
peekNode.right)
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← s.pop()
```

Trees: Level-Order

- Breadth-First Search

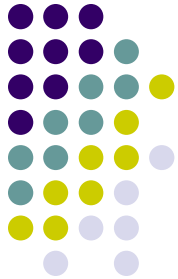


Trees: Level-Order



```
levelorder(root)
   q ← empty queue
   q.enqueue(root)
   while (not q.isEmpty())
      node ← q.dequeue()
      visit(node)
      if (node.left ≠ null)
        q.enqueue(node.left)
      if (node.right ≠ null)
        q.enqueue(node.right)
```

Trees: Tries (Prefix trees)



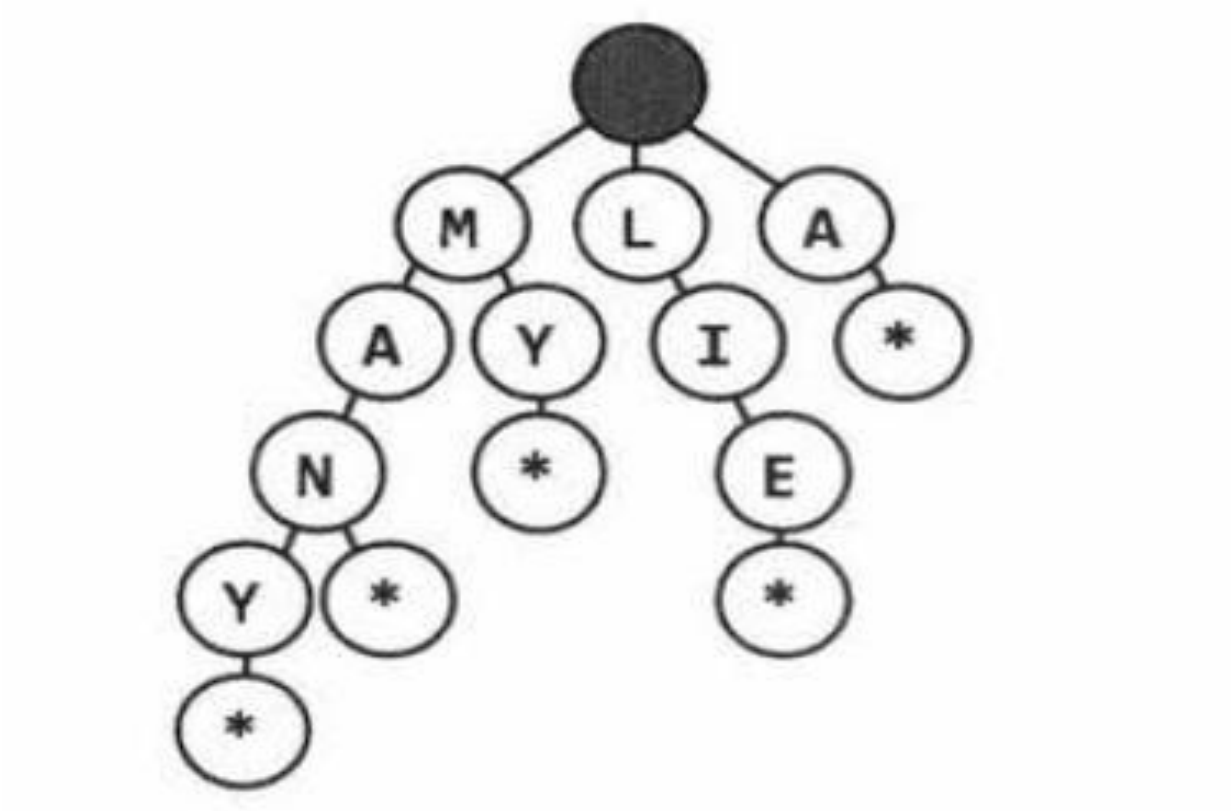
A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word.

The * nodes (sometimes called "null nodes") are often used to indicate complete words. For example, the fact that there is a * node under MANY indicates that MANY is a complete word. The existence of the MA path indicates there are words that start with MA.

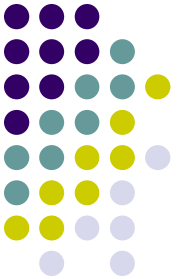
The actual implementation of these * nodes might be a special type of child (such as a `TerminatingTrieNode`, which inherits from `TrieNode`). Or, we could use just a boolean flag `terminates` within the "parent" node.

A node in a trie could have anywhere from 1 through `ALPHABET_SIZE + 1` children (or, 0 through `ALPHABET_SIZE` if a boolean flag is used instead of a * node).

Trees: Tries (Prefix trees)



Trees: Tries (Prefix trees)

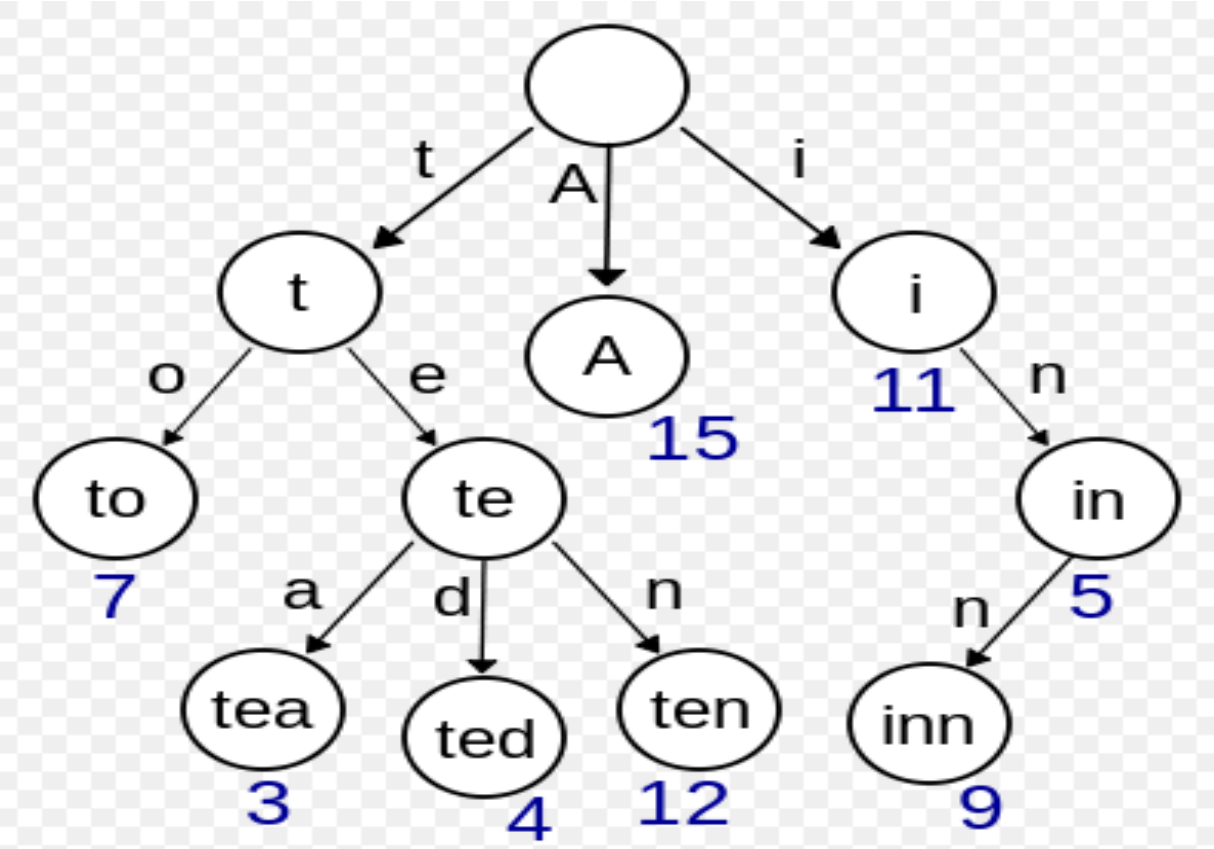


- Applications:
  - Networking prefix routing
  - Word auto-completion
  - Checking valid word:
    - O(k), where K is the number of characters in the string.

Trees: Tries (Prefix trees)

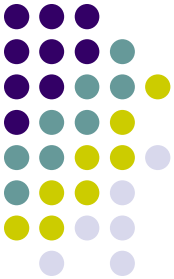
- Implementation:
  - Node:
    - Value: char
    - Pointer to children nodes indexed by char
      - Hash char -> Node
      - Array of all chars -> Node

Trees: Tries (Prefix trees)



Santa Cruz
UC Extension
in Silicon Valley

Trees: Tries (Prefix trees)

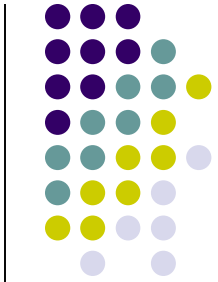


```
class Node():
  def __init__(self):
      # Note that using dictionary for children (as in this implementation) would not allow lexicographic sorting mentioned in the next section (Sorting),
      # because ordinary dictionary would not preserve the order of the keys
      self.children = {}  # mapping from character ==> Node
      self.value = None
```

```
def find(node, key):
    for char in key:
        if char in node.children:
            node = node.children[char]
        else:
            return None
    return node.value
```

```
def insert(root, string, value):
    node = root
    index_last_char = None
    for index_char, char in enumerate(string):
        if char in node.children:
            node = node.children[char]
        else:
            index_last_char = index_char
            break

    # append new nodes for the remaining characters, if any
    if index_last_char is not None:
        for char in string[index_last_char:]:
            node.children[char] = Node()
            node = node.children[char]

    # store value in the terminal node
    node.value = value
```

Trees: Java Interface

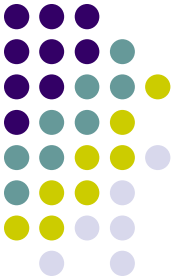


- TreeSet, TreeMap: O(log(n)) basic operations
  - Add()
  - Remove()
  - Contains()

Trees: Java Interface

```
1     public static void printHighChangeables( Map<String,List<String>> adjWords,
2                                                                      int minWords )
3     {
4         for( Map.Entry<String,List<String>> entry : adjWords.entrySet( ) )
5         {
6             List<String> words = entry.getValue( );
7
8             if( words.size( ) >= minWords )
9             {
10                System.out.print( entry.getKey( ) + " (" );
11                System.out.print( words.size( ) + "):" );
12                for( String w : words )
13                    System.out.print(  " " + w );
14                System.out.println( );
15            }
16        }
17    }
```

Trees: C++ Interface



- Set, map: O(log(n)) basic operations
  - insert()
  - find()
  - Contains()

```
set<string,CaseInsensitiveCompare> s;
s.insert( "Hello" ); s.insert( "HeLLo" );
cout << "The size is: " << s.size( ) << endl;
```

Trees: AVL Trees

- AVL Tree: height of left and right subtree differ by 1 at most.

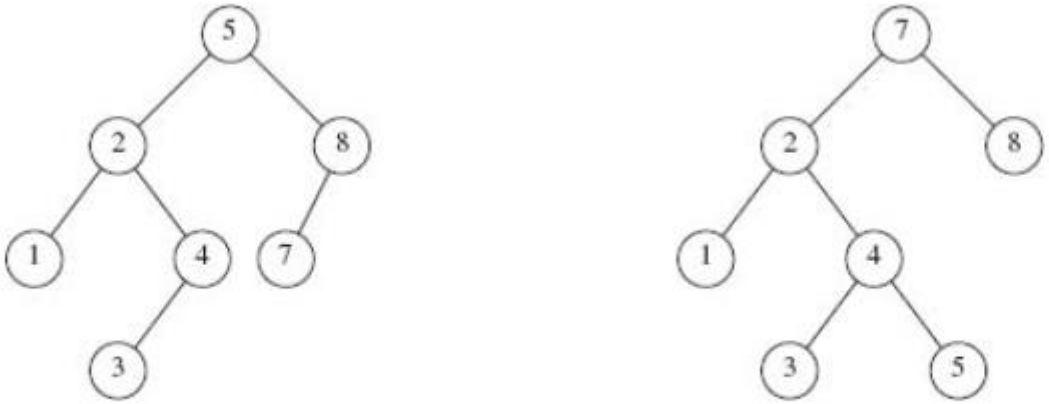
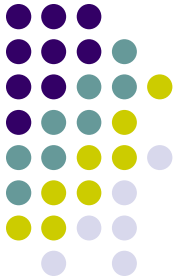


Figure 4.29 Two binary search trees. Only the left tree is AVL

Trees: AVL Trees



- AVL Tree property violated on insertion.

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

Trees: AVL Trees

- AVL Tree property violated on insertion.

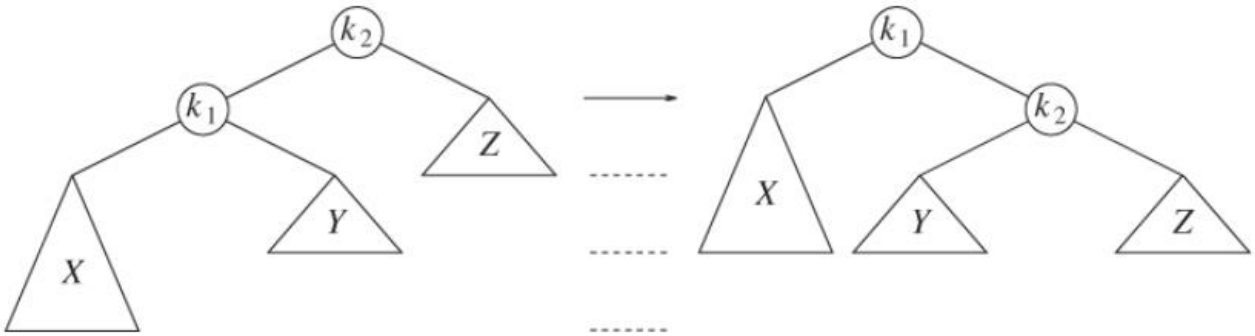


Figure 4.31 Single rotation to fix case 1

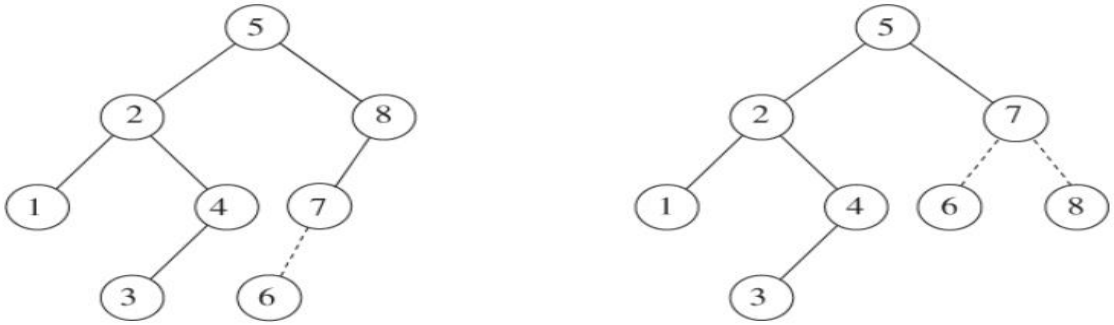


Figure 4.32 AVL property destroyed by insertion of 6, then fixed by a single rotation

UC Santa Cruz Extension in Silicon Valley

Trees: AVL Trees

- AVL Tree property violated on insertion.

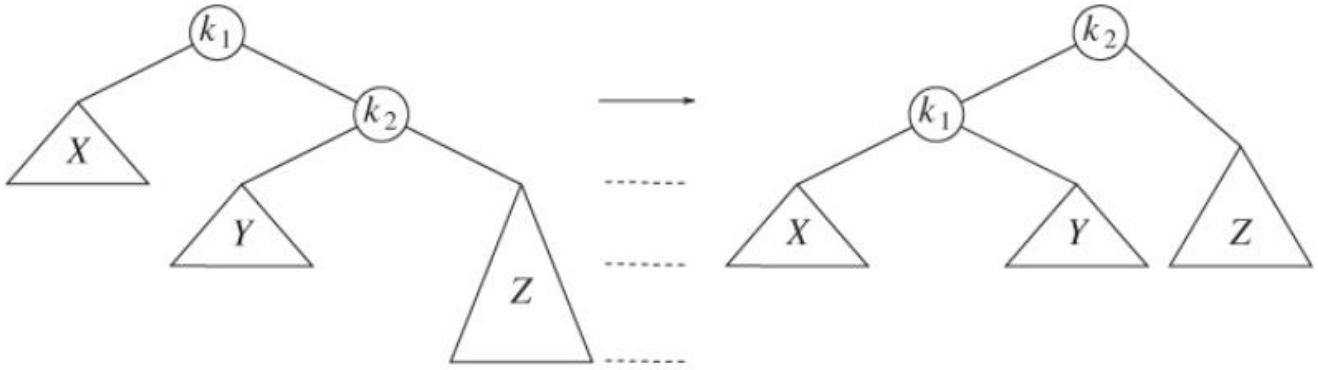
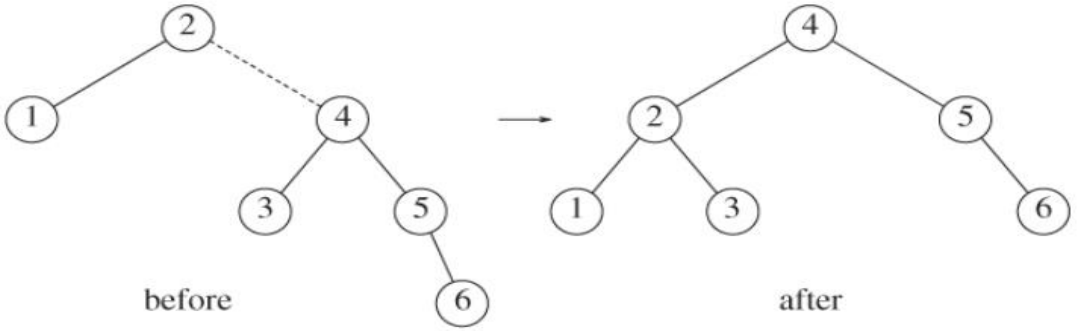


Figure 4.33 Single rotation fixes case 4



UC Santa Cruz Extension in Silicon Valley

Trees: AVL Trees

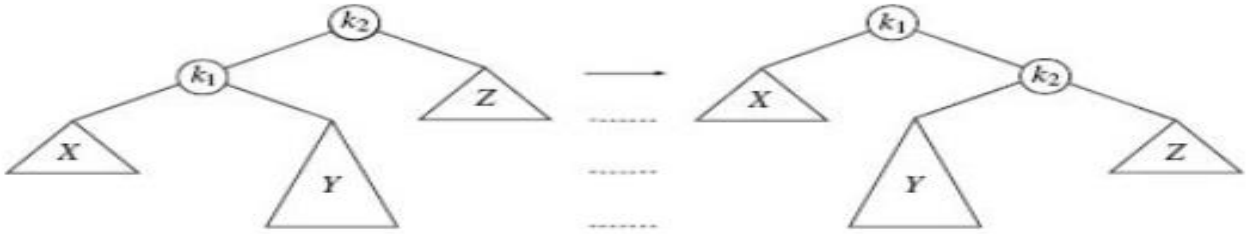


Figure 4.34 Single rotation fails to fix case 2

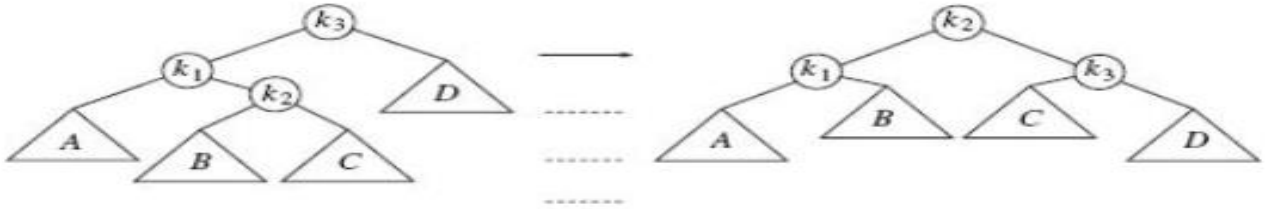


Figure 4.35 Left-right double rotation to fix case 2

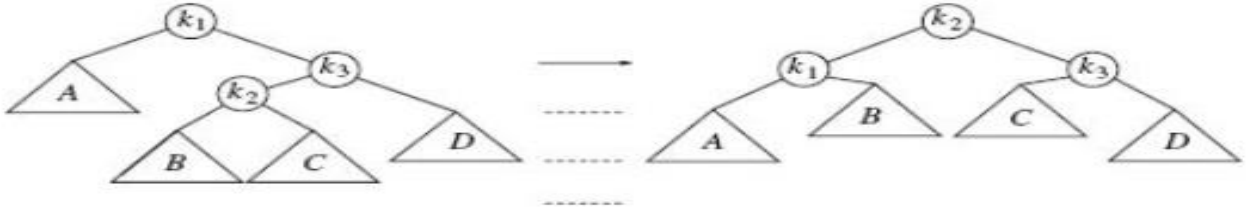


Figure 4.36 Right-left double rotation to fix case 3

Trees: Splay Trees

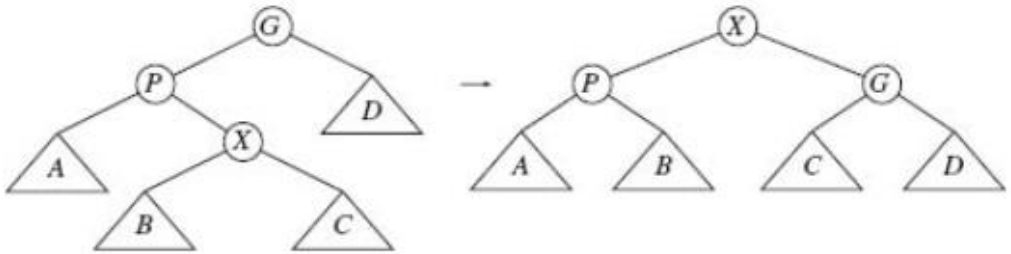


Figure 4.45 Zig-zag

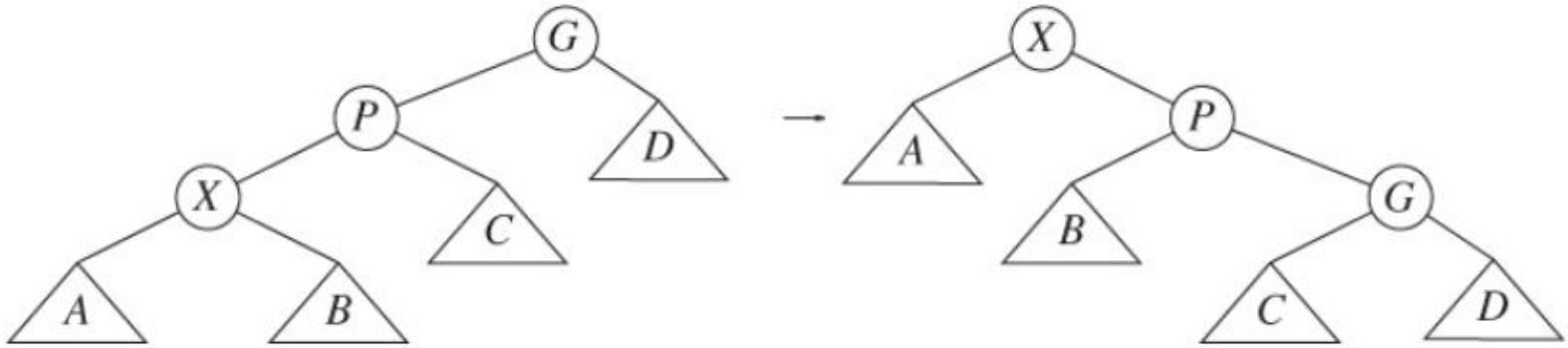
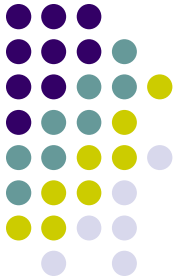


Figure 4.46 Zig-zig

Trees: B-Trees



A B-tree of order *M* is an *M*-ary tree with the following properties:[5]

[5] Rules 3 and 5 must be relaxed for the first L insertions.

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M - 1$ keys to guide the searching; key *i* represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between two and *M* children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and *M* children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and *L* data items, for some *L* (the determination of *L* is described shortly).

Trees: B-Trees

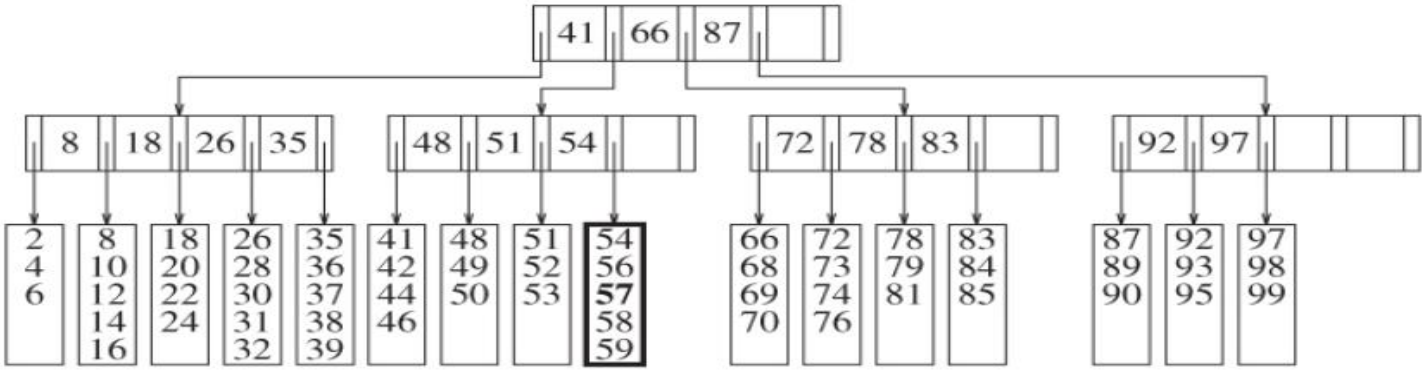


Figure 4.61 B-tree after insertion of 57 into the tree in Figure 4.60

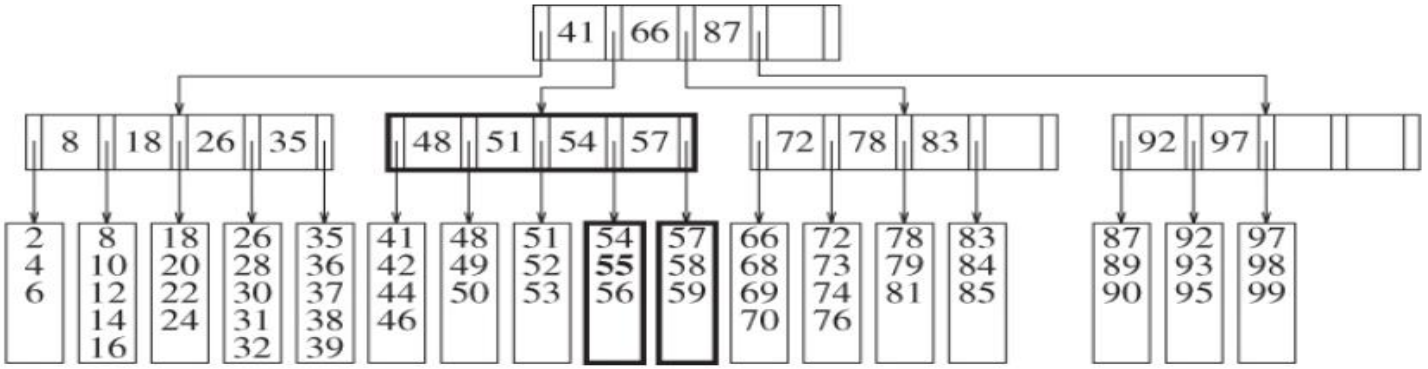
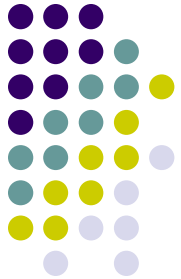


Figure 4.62 Insertion of 55 into the B-tree in Figure 4.61 causes a split into two leaves

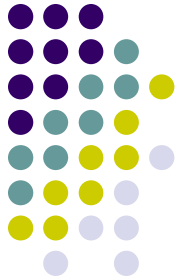
UC Santa Cruz Extension in Silicon Valley

Graphs: Concept



- Collection of nodes and edges between them.
- Tree is a special type of graph
  - Hence algorithms that apply to graphs apply to trees.

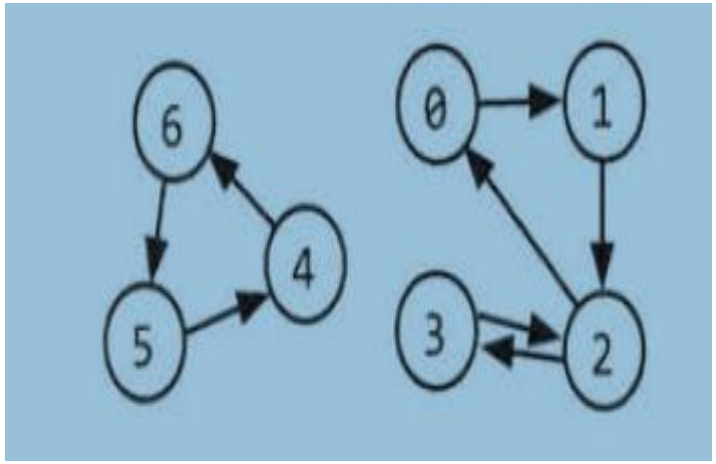
Graphs: Representation



- Adjacency List
  - Array of nodes
  - Each node has a linked list or array of children nodes.
  - Good for sparse graphs

```
1  class Graph {
2    public Node[] nodes;
3  }
4
5  class Node {
6    public String name;
7    public Node[] children;
8  }
```

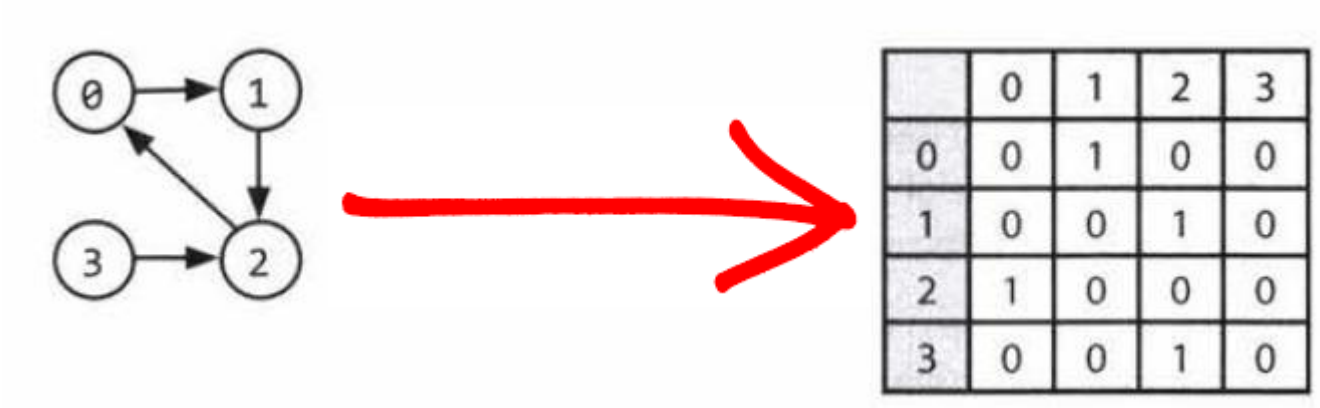
Graphs: Representation



```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```

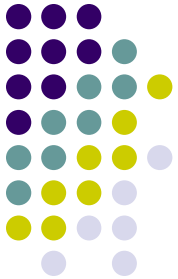
Graphs: Representation

- Adjacency Matrix
  - Matrix[i][j] indicates if there is an edge between node i and node j.



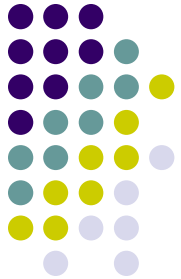
	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

Graphs: Directed vs Undirected



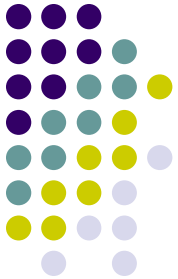
- Directed graph:
  - Edge (i, j) is different from Edge (j, i)
  - Sample use: class prerequisite relationship
- Undirected graph:
  - Edge (i, j) implies Edge(j, i)
  - Adjacency Matrix is symmetric
    - Need only $n^2/2$ entries
  - Sample use: network interconnect.

Graphs: Weighted graphs



- Weight or value associated with each Edge, in addition to its presence or not.
- Sample use:
  - Network interconnect with preferred links.

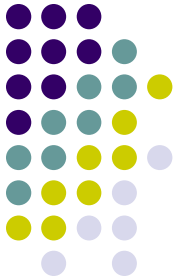
Graphs: Traversals



- Depth-first Search:

In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name *depth-first search*) before we go wide.

Graphs: Traversals



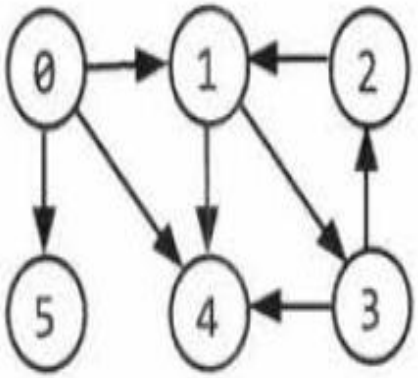
- Breadth-first Search:

In breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence *breadth*-first search) before we go deep.

Graphs: Traversals

- Breadth vs Depth-first Search:

Graph



Depth-First Search

```
1  Node 0
2    Node 1
3      Node 3
4        Node 2
5        Node 4
6  Node 5
```

Breadth-First Search

```
1  Node 0
2  Node 1
3  Node 4
4  Node 5
5  Node 3
6  Node 2
```

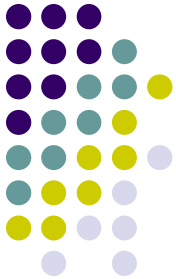
Graphs: Traversals

- Breadth vs Depth-first Search:
  - DFS: Good for visiting all nodes
  - BFS: Good for shortest path

Consider representing all the friendships in the entire world in a graph and trying to find a path of friendships between Ash and Vanessa.

In depth-first search, we could take a path like `Ash -> Brian -> Carleton -> Davis -> Eric -> Farah -> Gayle -> Harry -> Isabella -> John -> Kari`... and then find ourselves very far away. We could go through most of the world without realizing that, in fact, `Vanessa` is Ash's friend. We will still eventually find the path, but it may take a long time. It also won't find us the shortest path.

Graphs: Traversals

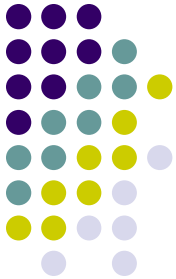


- Depth-first Search:

In DFS, we visit a node a and then iterate through each of a's neighbors. When visiting a node b that is a neighbor of a, we visit all of b's neighbors before going on to a's other neighbors. That is, a exhaustively searches b's branch before any of its other neighbors.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in an infinite loop.

Graphs: Traversals



- Depth-first Search:

The pseudocode below implements DFS.

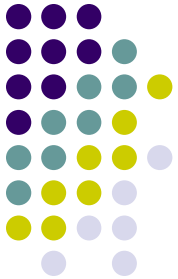
```
1   void search(Node root) {
2      if (root == null) return;
3      visit(root);
4      root.visited = true;
5      for each (Node n in root.adjacent) {
6         if (n.visited == false) {
7            search(n);
8         }
9      }
10  }
```

Graphs: Traversals

- Breadth-first Search:

In BFS, node a visits each of a's neighbors before visiting any of *their* neighbors. You can think of this as searching level by level out from a. An iterative solution involving a queue usually works best.

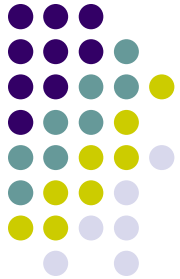
Graphs: Traversals



- Breadth-first Search:

```
1   void search(Node root) {
2      Queue queue = new Queue();
3      root.marked = true;
4      queue.enqueue(root); // Add to the end of queue
5
6      while (!queue.isEmpty()) {
7         Node r = queue.dequeue(); // Remove from the front of the queue
8         visit(r);
9         foreach (Node n in r.adjacent) {
10           if (n.marked == false) {
11              n.marked = true;
12              queue.enqueue(n);
13           }
14        }
15     }
16  }
```

Graphs: Traversals



- Bidirectional Search:

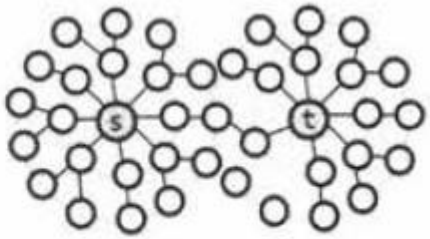
Bidirectional search is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous breadth-first searches, one from each node. When their searches collide, we have found a path.

Graphs: Traversals

- BFS vs Bidirectional Search:

Breadth-First Search
Single search from s to t that collides after four levels.

Bidirectional Search
Two searches (one from s and one from t) that collide after four levels total (two levels each).



Graphs Algorithms: Topological sort

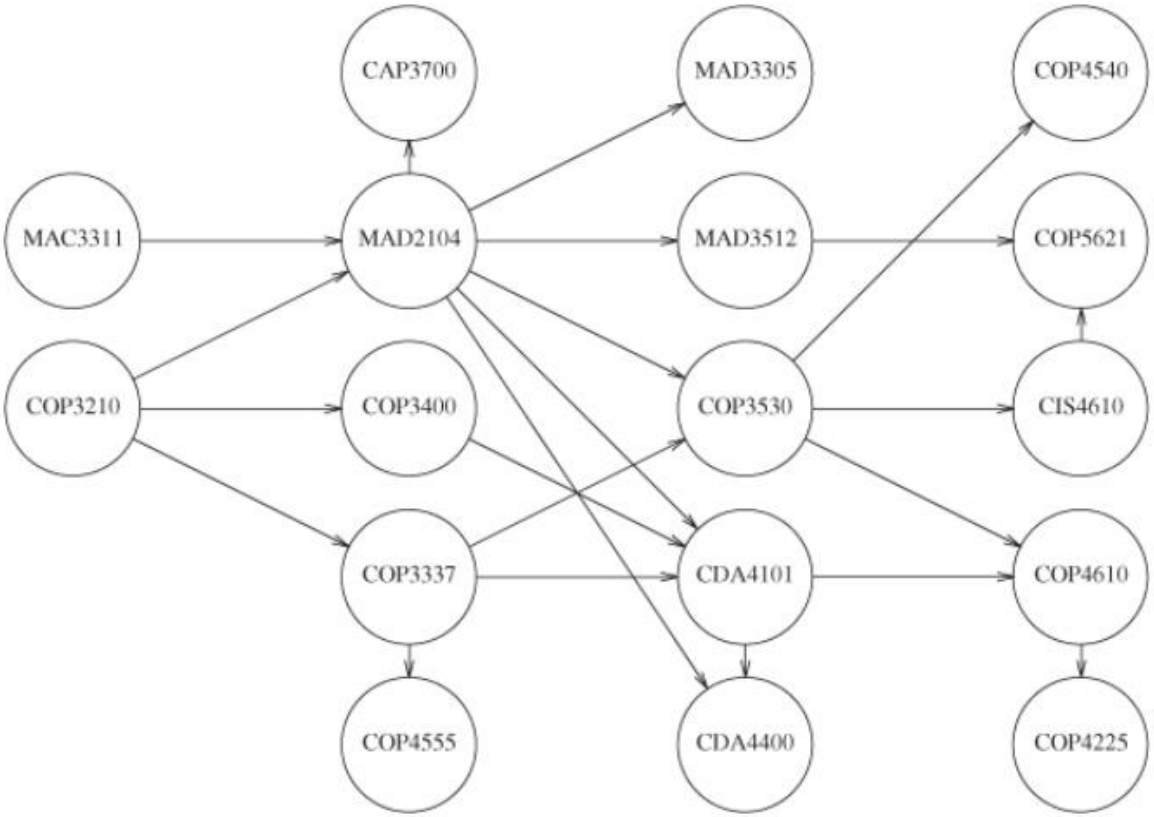


Figure 9.3 An acyclic graph representing course prerequisite structure

Graphs Algorithms: Topological sort

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

Graphs Algorithms: Topological sort

Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

Indegree Before Dequeue #							
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v3,v7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

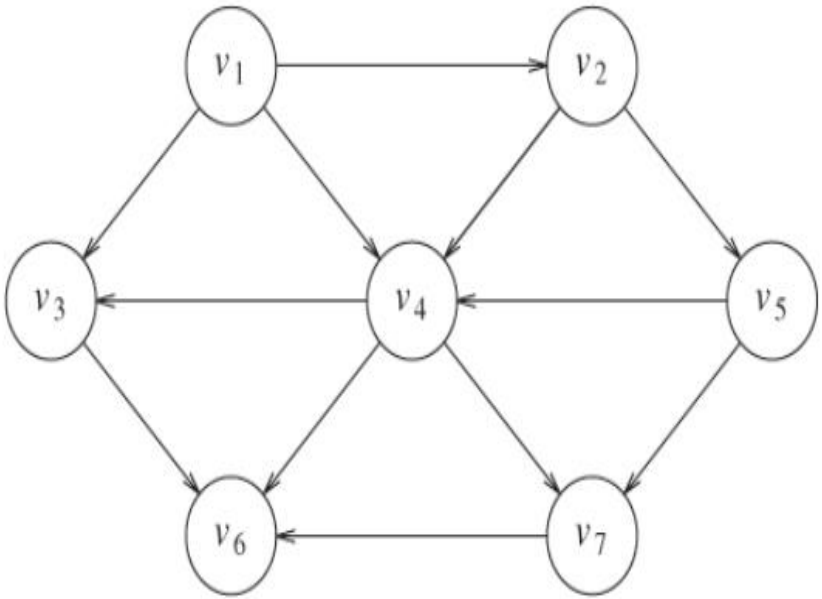


Figure 9.4 An acyclic graph

Graphs Algorithms: Topological sort

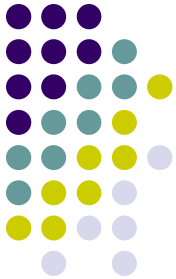


Figure 9.7 Pseudocode to perform topological sort

```
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

        while( !q.isEmpty( ) )
        {
            Vertex v = q.dequeue( );
            v.topNum = ++counter;  // Assign next number

            for each Vertex w adjacent to v
                if( --w.indegree == 0 )
                    q.enqueue( w );
        }
        if( counter != NUM_VERTICES )
            throw new CycleFoundException( );
}
```

Graph Algorithms: Dijkstra's Shortest Path

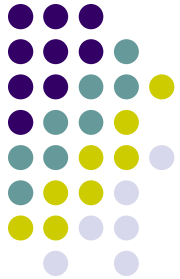
```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Graph Algorithms: Dijkstra's Shortest Path

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Figure 9.31 Pseudocode for Dijkstra's algorithm

Sample Problems



4.1 **Route Between Nodes:** Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Hints: #127

pg 241

4.2 **Minimal Tree:** Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

Hints: #19, #73, #116

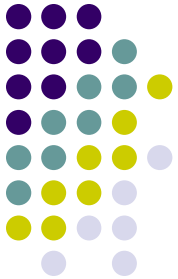
pg 242

4.3 **List of Depths:** Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).

Hints: #107, #123, #135

pg 243

Sample Problems



4.4 **Check Balanced:** Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

Hints: #21, #33, #49, #105, #124

pg 244

4.5 **Validate BST:** Implement a function to check if a binary tree is a binary search tree.

Hints: #35, #57, #86, #113, #128

pg 245

4.6 **Successor:** Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

Hints: #79, #91

pg 248

Sample Problems

4.7 **Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

Output: `f, e, a, b, d, c`

Hints: #26, #47, #60, #85, #125, #133

Sample Problems

pg 250

4.8 **First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

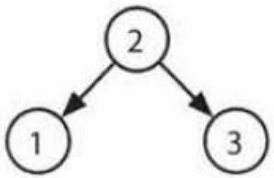
Hints: #10, #16, #28, #36, #46, #70, #80, #96

pg 257

4.9 **BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:

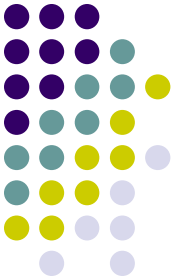


Output: `{2, 1, 3}, {2, 3, 1}`

Hints: #39, #48, #66, #82

pg 262

Sample Problems



4.10 **Check Subtree:** T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

Hints: #4, #11, #18, #31, #37

pg 265

4.11 **Random Node:** You are implementing a binary tree class from scratch which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

Hints: #42, #54, #62, #75, #89, #99, #112, #119

pg 268

4.12 **Paths with Sum:** You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

Hints: #6, #14, #52, #68, #77, #87, #94, #103, #108, #115

pg 272