DATA STRUCTURES & ITS APPLICATIONS
UE20CS202

Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES & ITS APPLICATIONS

AVL Trees

Kusuma K V
Department of Computer Science & Engineering

Self-Balancing Binary Search Trees

- A self-balancing binary search tree or height-balanced binary search tree is a binary search tree (BST) that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically
- The disadvantage of a binary search tree is that its height can be as large as N-1
- Most operations on a BST take time proportional to the height of the tree, so it is desirable to keep the height small
- This means that the time needed to perform insertion, deletion and many other operations can be O(N) in the worst case

Self-Balancing Binary Search Trees

- We want a tree with small height
- A binary tree with N nodes has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree $O(\log N)$
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree

- A typical operation done by trees to maintain balance is rotation

AVL Tree

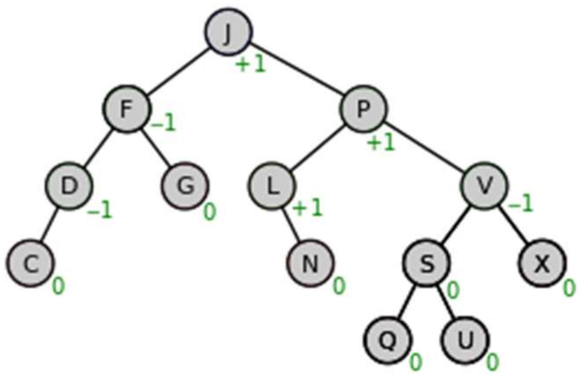
- An AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree
- It was the first such data structure to be invented
- **In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property**
- Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation
- Insertions and deletions may require the tree to be rebalanced by one or more tree rotations

https://en.wikipedia.org/wiki/AVL_tree

DATA STRUCTURES & ITS APPLICATIONS
AVL Tree

Balance Factor

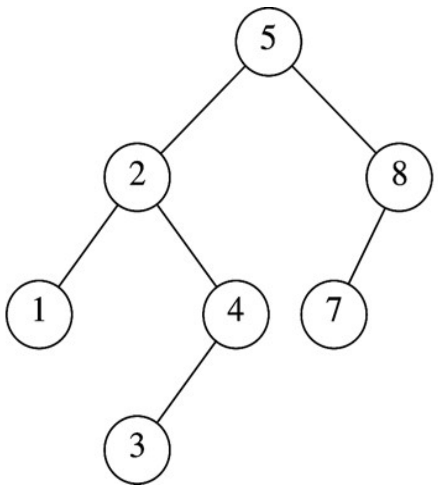
- In a binary tree the balance factor of a node X is defined to be the height difference
  BF(X) := Height(RightSubtree(X)) - Height(LeftSubtree(X))
  of its two child sub-trees
- A binary tree is defined to be an AVL tree if the invariant
- BF(X) = {-1,0,1} holds for every node X in the tree
- A node X with BF(X) < 0 is called "left-heavy", one with BF(X) > 0 is called "right-heavy", and one with BF(X) = 0 is sometimes simply called "balanced"



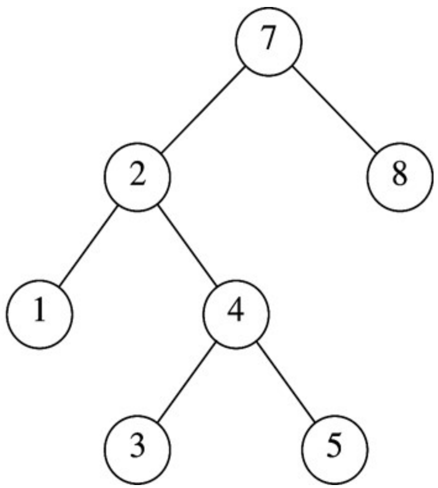
Note: It can also be BF(X) := Height(LeftSubtree(X)) - Height(RightSubtree(X))
In which case, the terminologies left heavy and right heavy will mean in reverse way

https://en.wikipedia.org/wiki/AVL_tree

AVL Tree



AVL tree



Not an AVL tree

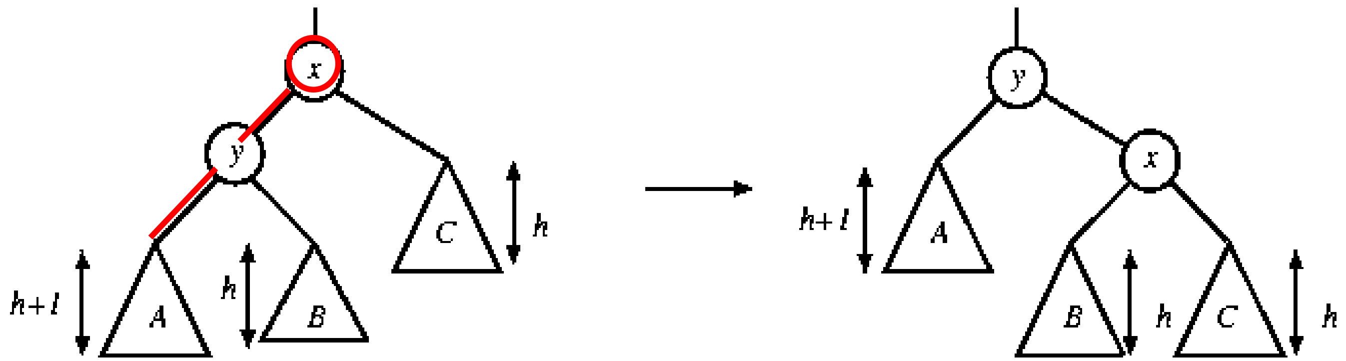
https://en.wikipedia.org/wiki/AVL_tree

AVL Tree

Types of imbalance and Rotations involved

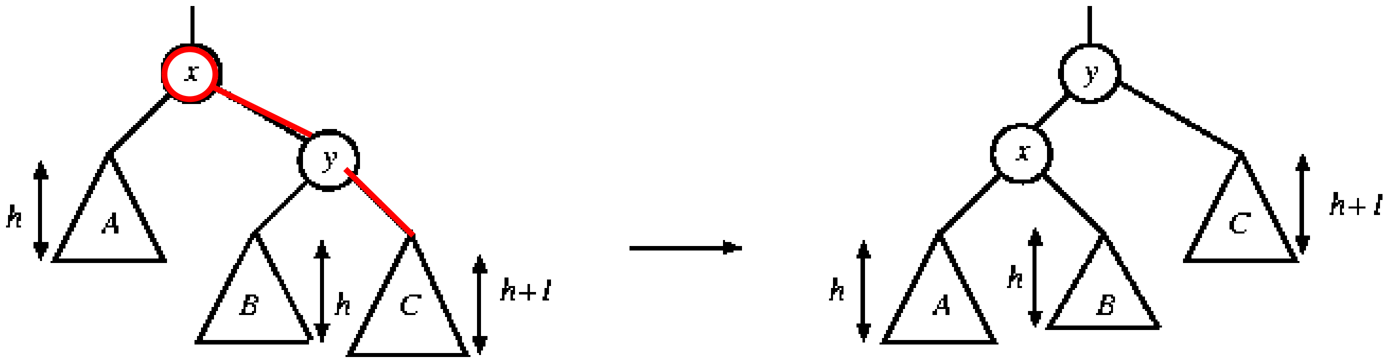
- LL imbalance : Right rotation (Single rotation)
- RR imbalance: Left rotation (Single rotation)
- LR imbalance: LR rotation (Double rotation)
- RL imbalance: RL rotation (Double rotation)

LL imbalance : Right rotation (Single rotation)



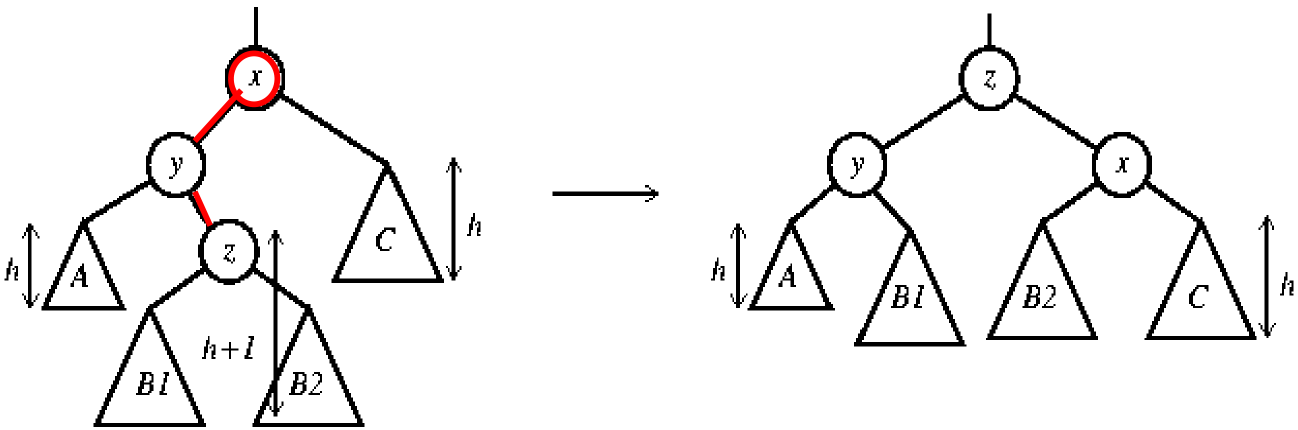
Rotate with left child

RR imbalance : Left rotation (Single rotation)



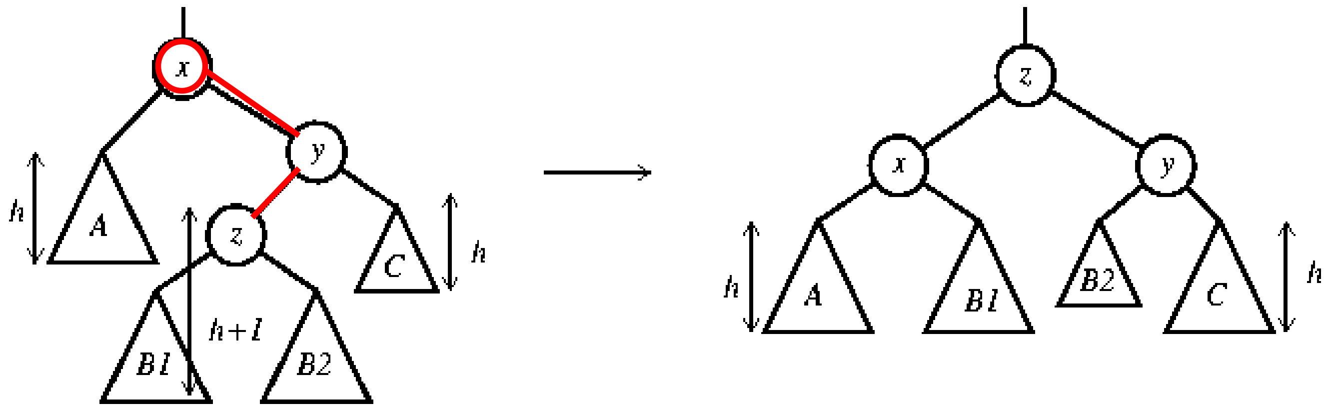
Rotate with right child

LR imbalance : LR rotation (Double rotation)



Double rotate with left child

RL imbalance : RL rotation (Double rotation)

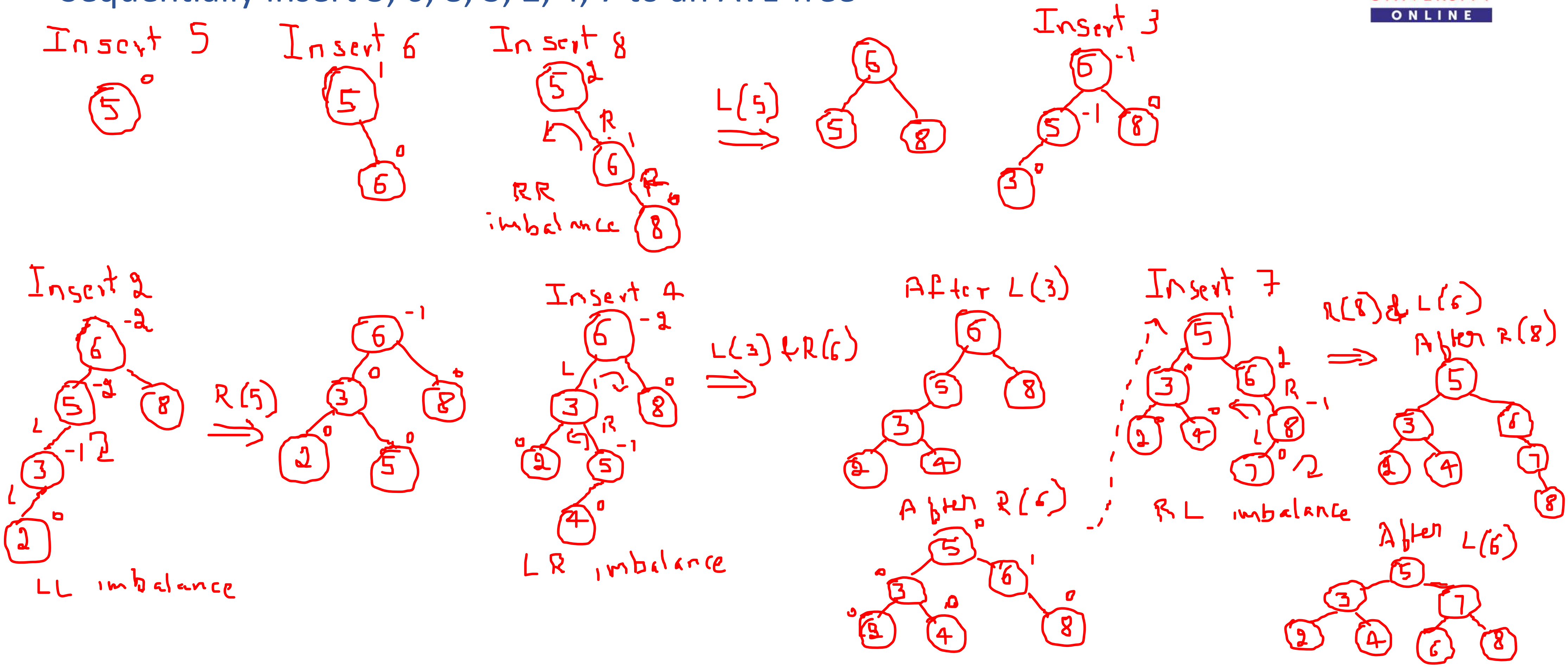


Double rotate with right child

Sequentially insert 5, 6, 8, 3, 2, 4, 7 to an AVL Tree

Self-Balancing Binary Search Trees: AVL Tree

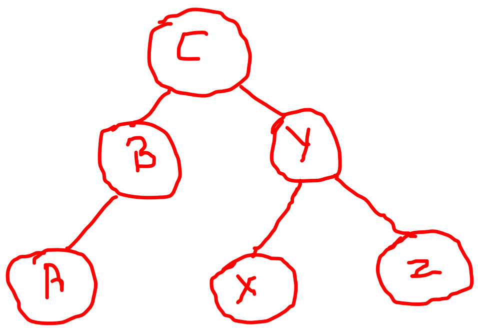
Sequentially insert 5, 6, 8, 3, 2, 4, 7 to an AVL Tree



Sequentially insert A, Z, B, Y, C, X to an AVL Tree

Try it Out

Final Tree



For steps the following visualization site may help, as said in the class !!

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Sequentially insert A, Z, B, Y, C, X, D, W to an AVL Tree

AVL Tree

- AVL trees are often compared with red–black trees because both support the same set of operations and take O(logn) time for the basic operations
- For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced

https://en.wikipedia.org/wiki/AVL_tree

Applications

- AVL trees are used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required
- It is used in applications that require improved searching apart from the database applications

Graphs

Given graph G = (V , E)

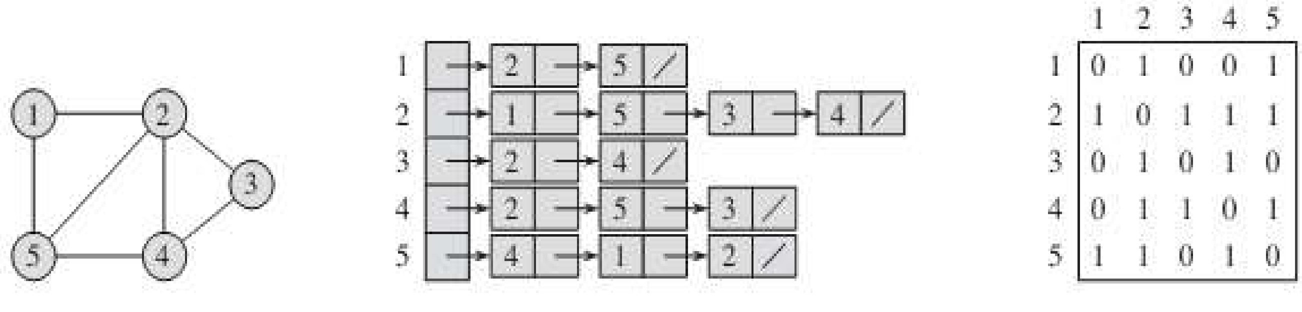
- May be either directed or undirected
- Two common ways to represent:

1. Adjacency lists
2. Adjacency matrix

- Adjacency-list representation provides a compact way to represent sparse graphs—those for which |E| much less than $|V|^2$ - it is usually the method of choice
- Adjacency matrix representation is preferred when the graph is dense - |E| close to $|V|^2$ or when we need to be able to tell quickly if there is an edge connecting two given vertices

Graphs - representation

Two representations of an undirected graph



(a) (b) (c)

(a) An undirected graph G with 5 vertices and 7 edges
(b) An adjacency – list representation of G
(c) The adjacency matrix representation of G

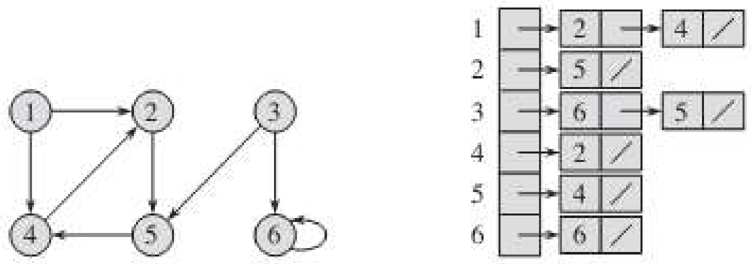
- The adjacency-list representation of a graph G=(V,E) consists of an array Adj of |V| lists, one for each vertex in V
- Adj[u] consists of all the vertices adjacent to u in G (Alternatively, it may contain pointers to these vertices)

If vertices are numbered 1, 2, … , |V| in some arbitrary manner then the adjacency matrix representation of a graph G consists of a |V| x |V| matrix $A=(a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 \text{ if}(i,j) \in E \\ 0 \text{ otherwise} \end{cases}$$

Graphs - representation

Two representations of a directed graph



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

(a) A directed graph G with 6 vertices and 8 edges

(b) An adjacency – list representation of G

(c) The adjacency matrix representation of G

Graphs - representation

- If G is a directed graph, the sum of the lengths of all the adjacency lists is |E| since an edge of the form (u,v) is represented by having v appear in Adj[u]
- If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2|E| ,since if (u,v) is an undirected edge, then u appears in v's adjacency list and vice versa
- For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is , $\Theta(V+E)$

Graphs - representation

- A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u,v) is present in the graph than to search for v in the adjacency list Adj[u]
- An adjacency-matrix representation of the graph remedies this disadvantage, but at the cost of using asymptotically more memory
- The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph

Breadth First Search

Breadth First Search (BFS)

- Algorithm used for traversing a graph data structure
- It starts at the tree root (or some arbitrary node of a graph) and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level

- Moore discovered BFS in the context of finding paths through mazes
- Lee independently discovered the same algorithm in the context of routing wires on circuit boards
- Similar to level order traversal on trees

PES UNIVERSITY ONLINE

Depth First Search (DFS)

- Algorithm for traversing (or searching) tree (or graph) data structures
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking

- DFS is widely used in artificial intelligence problems
- Similar to pre order traversal in trees

Applications of BFS, DFS

Applications of Breadth First Search (BFS)

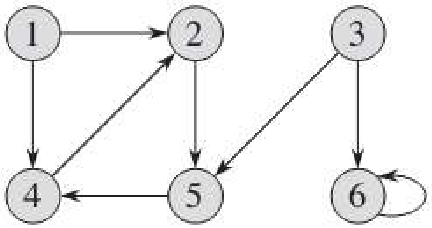
- Path Finding Algorithms
- Finding shortest path between two nodes
- In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels
- …
- …

Applications of Depth First Search (DFS)

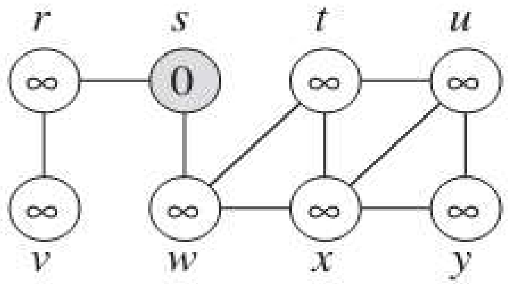
- Path Finding Algorithms
- To find the connected components in a graph
- Topological sorting
- To detect cycles in a graph
- …
- …

BFS

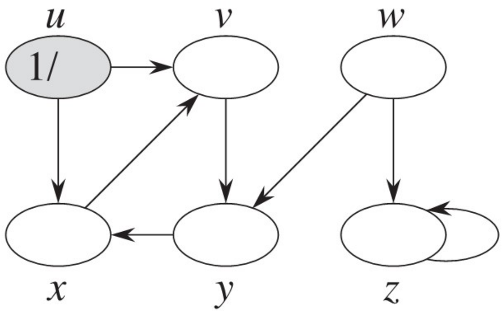
BFS example: Consider 3 to be the source vertex



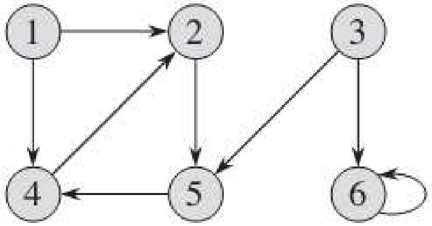
Consider s to be the source vertex



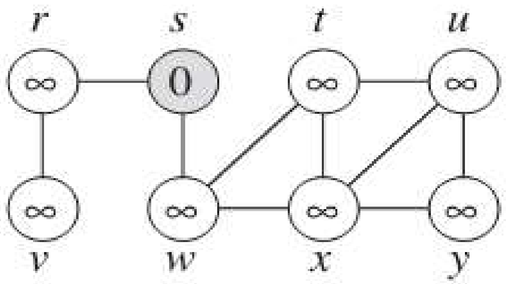
Consider u to be the source vertex



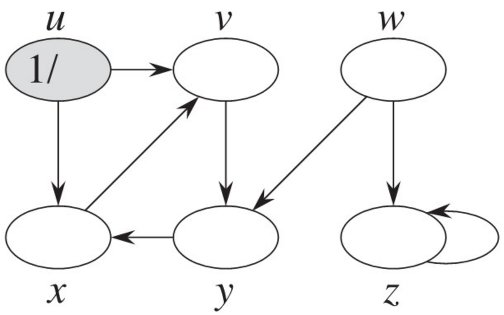
DFS example: Consider 3 to be the source vertex



Consider s to be the source vertex



Consider u to be the source vertex



THANK YOU

Kusuma K V

Department of Computer Science
& Engineering

kusumakv@pes.edu