1

Balanced Trees, Graphs

CSC 340

April 20, 2016

2

Overview

- **Balanced Search Trees**
- Graphs

3

Balanced Search Trees

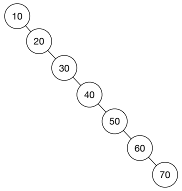
- The efficiency of the binary search tree implementation that we've seen (and implemented) is related to the tree's height. Why?

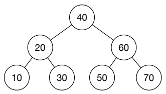
Sensitive to order of insertion and deletion

4

Balanced Search Trees

- Consider the following trees that could be generated from the same set of items, based on insertion order





First: 10, 20, 30, 40, 50, 60, 70

Second: 40, 20, 60, 10, 30, 50, 70

5

Balanced Search Trees

- Height of a binary search tree of n items:
  - Maximum: n
  - Minimum: log2(n + 1)

6

Balanced Search Trees

- Search trees can retain their balance despite order of insertions and deletions
  - 2-3 trees
  - 2-3-4 trees
  - Red-black trees
  - AVL trees

7

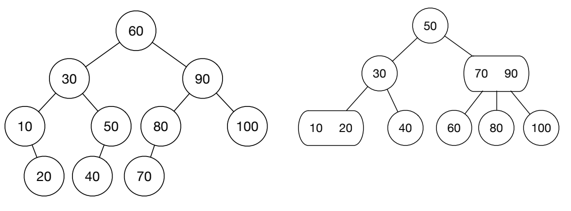
2-3 Trees

- Have 2-nodes and 3-nodes
  - A 2-node has one data item and two children
  - A 3-node has two data items and three children
- Are general trees, not binary trees
- Are never taller than a minimum hight binary tree
  - A 2-3 tree with n nodes never has height greater than log2(n + 1)

8

2-3 Trees - Examples

- Balanced binary search tree and a 2-3 tree with the same elements



9

2-3 Trees

- A leaf may contain either one or two data items
- To traverse a 2-3 tree, perform the analogue of an in-order traversal
- Searching a 2-3 tree is as efficient as searching the shortest binary search tree - O(log2 n)
- Insertion into a 2-node leaf is simple
- Insertion into a 3-node leaf causes it to divide

10

2-3 Trees - Insertions

- A 2-node contains a single data item whose search key S satisfies the following:
  - S > the left child's search key(s)
  - S < the right child's search key(s)

11

2-3 Trees - Insertions

- A 3-node contains two data items whose search keys S and L satisfy the following:
  - S > the left child's search key(s)
  - S < the middle child's search key(s)
  - L > the middle child's search key(s)
  - L < the right child's search key(s)

12

2-3 Trees - Insertions

- Locate the leaf at which the search for the item would terminate
- Insert the new item into the leaf
- If the leaf now contains only two items, you are done
- If the leaf now contains three items, a < b < c, split the leaf into two nodes containing a and c, and promote b to the parent
- When the root contains three items
  - Split the root into two nodes
  - Create a new root node
  - Tree grows in height

Deletions are slightly more complicated than I want to go into, but I encourage you to read about it

13

2-3 Trees

- Maintaining the balance of a 2-3 tree is relatively easy (not so with the binary search tree)
- A 2-3 implementation of a table is O(log2 n) for all table operations (insert, remove, lookup)
- A 2-3 tree is a compromise
  - Searching a 2-3 tree is not more efficient than searching a binary search tree of minimum height
  - The 2-3 tree might be shorter, but the advantage is offset by the extra comparisons in a node having two values

14

2-3-4 Trees

- Have 2-nodes, 2-nodes, and 4-nodes
  - 4-node has three data items and four children
- Are general trees, not binary trees
- Are never taller than a 2-3 tree
- Search and traversal algorithms are simple extensions of the corresponding 2-3 tree algorithms

15

Overview

- Balanced Search Trees
- **Graphs**

16

Definition

- A graph G consists of two sets
    - A set, V, of vertices (or nodes)
    - A set, E, of edges
- G = { V, E }
- A subgraph consists of a subset of a graph's vertices and a subset of its edges

17

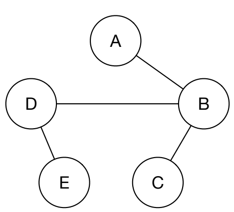
Terminology

- **Adjacent vertices**: Two vertices that are joined by an edge
- **Path**: A sequence of edges that begins at one vertex and ends at another vertex
    - May pass through the same vertex more than once
- **Simple Path**: A path that passes through a vertex only once
- **Cycle**: A path that begins and ends at the same vertex
- **Simple Cycle**: A cycle that does not pass through a vertex more than once

18

Terminology

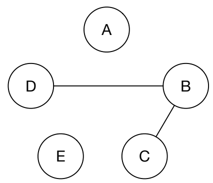
- Connected Graph: A graph that has a path between each pair of distinct vertices



19

Terminology

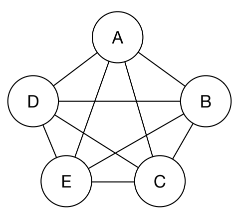
- Disconnected Graph: A graph that has at least one pair of vertices without a path between them



20

Terminology

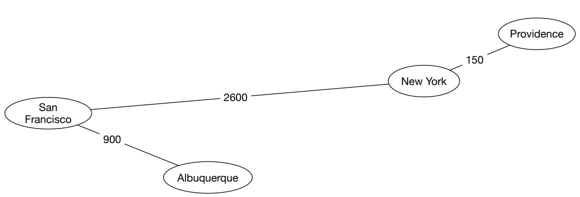
- Complete Graph: A graph that has an edge between each pair of distinct vertices



21

Weighted Graph

- A graph whose edges have numeric labels



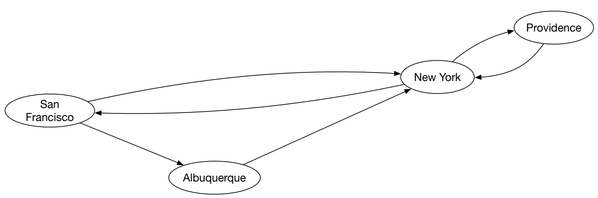
Directed Graph

22

- Examples we've seen so far are *undirected*
- In a directed graph,
  - Each edge has a direction (directed edges)
  - Can have two edges between a pair of vertices (one in each direction)
  - Directed path is a sequence of directed edges between two vertices
  - Vertex Y is adjacent to vertex X if there is a directed edge from X to Y

Directed Graph Example

23



Note this is a directed, unweighted graph, but distinct edges can have weights as well, and the weights can vary in each direction

Graphs as ADTs

24

- Variations of an ADT graph are possible
  - Vertices may or may not contain values (city names for example) - many problems are solved with only the relationship among vertices
  - Directed or undirected
  - Weighted or unweighted
- Insertion and deletion operations for graphs apply to vertices and edges
- Graphs can have traversal operations

25

Implementations

- Most common implementation of a graph
  - Adjacency Matrix
  - Adjacency List
- Adjacency matrix for a graph that has N vertices numbers 0, 1, …, N - 1 is an N by N array matrix such that matrix[i][j] indicates whether an edge exists from vertex i to vertex j (and possibly a weight)

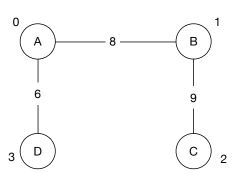
26

Adjacency Matrix

- For an unweighted graph, matrix[i][j] is
  - 1 (or true) if an edge exists from vertex i to vertex j
  - 0 (or false) if no edge exists from vertex i to vertex j
- For a weighted graph, matrix[i][j] is
  - The weight of the edge from vertex i to vertex j
  - Infinity if no edge exists from vertex i to vertex j

27

Adjacency Matrix



		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

28

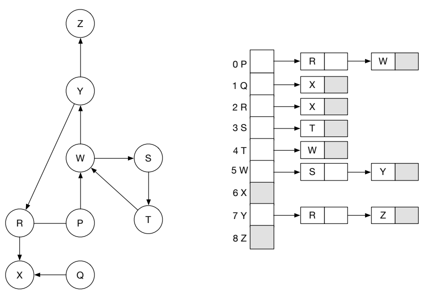
Adjacency Lists

- Adjacency list for a directed graph that has N vertices numbered 0, 1, …, N - 1
  - An array of N linked lists
  - The ith linked list has a node for vertex j iff an edge exists from vertex i to vertex j
  - The list's node can contain either
    - Vertex j's value (if any)
    - An indication of vertex j's identity

29

Adjacency Lists

- For an undirected graph, treat each edge as if it were two directed edges in opposite directions



The ints are indices with corresponding vertex value

30

Implementation v. Efficiency

- Two common graph operations:
  - Determine whether there is an edge from vertex i to vertex j
  - Find all vertices adjacent to a given vertex i
- Adjacency Matrix
  - Supports the first operation more efficiently
- Adjacency List
  - Supports the second operation more efficiently
  - Requires less space than an adjacency matrix

31

Breadth First Search

- Algorithm for traversing or searching tree or graph data structure - visits neighbors **first**
- Take a starting node, N
- Uses an array to record vertices that have been visited (and, optionally, the distance from the starting node)
- Uses a queue to record neighbor nodes that need to be visited
- From starting node, enqueue neighbors, and visit each in turn, setting distance in visited node iff the node has not yet been visited

32

Breadth First Search

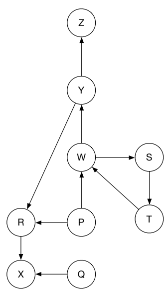
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	∞
T	∞
W	1
X	∞
Y	∞
Z	∞

Next to Visit Queue

R → W



33

Dequeued R, Adding R's adjacent vertex X

Breadth First Search

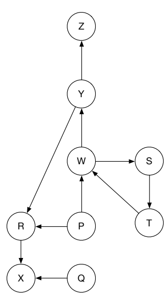
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	∞
T	∞
W	1
X	2
Y	∞
Z	∞

Next to Visit Queue

W → X



Breadth First Search

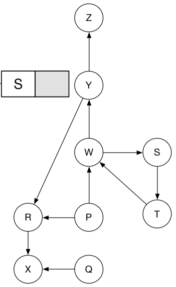
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	∞
W	1
X	2
Y	2
Z	∞

Next to Visit Queue

X → Y → S



34

Note that Y is now the value at W + 1 (or the weight, if this were a weighted graph)
Dequeued W, adding W’s adjacent vertices Y and S

Breadth First Search

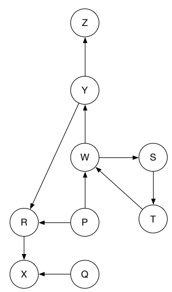
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	∞
W	1
X	2
Y	2
Z	∞

Next to Visit Queue

Y → S



35

Dequeued X, X has no adjacent vertices

Breadth First Search

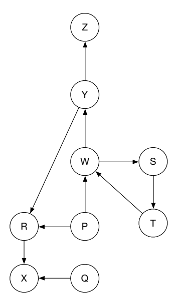
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	∞
W	1
X	2
Y	2
Z	3

Next to Visit Queue

S → Z



36

Dequeued Y, adding Y’s adjacent vertex Z

37

Breadth First Search

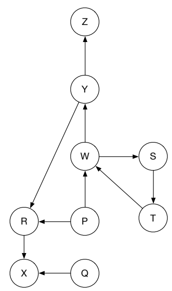
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	3
W	1
X	2
Y	2
Z	3

Next to Visit Queue

Z → T



Dequeued S, adding S’s adjacent vertex T

38

Breadth First Search

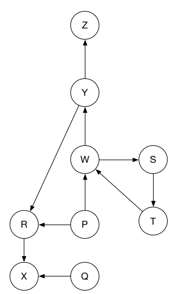
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	3
W	1
X	2
Y	2
Z	3

Next to Visit Queue

T



Dequeued Z, Z has no adjacent vertices

39

Breadth First Search

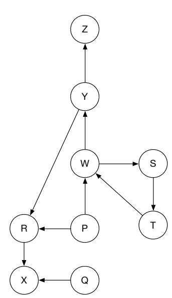
- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	3
W	1
X	2
Y	2
Z	3

Next to Visit Queue

W



Dequeued T, adding T’s adjacent vertex W

40

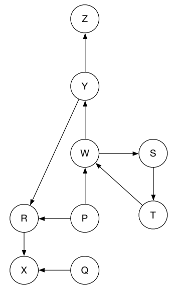
Breadth First Search

- Starting at P

Visited Array

P	0
Q	∞
R	1
S	2
T	3
W	1
X	2
Y	2
Z	3

Next to Visit Queue



Dequeued W, do not add W’s adjacent vertices S and Y as they have already been visited

41

Depth First Search

- Explore as far as possible along one path from a node before visiting additional nodes
- Same concept as BFS - how would you modify the solution for BFS to realize DFS?

Use a Stack instead of a Queue

42

Finding Paths

- Either BFS or DFS can be used to determine if a path exists between two vertices
- To determine if a path exists from I to J
  - Starting from a node I, if the visited array contains a value for node J, a path exists