Graph Traversals

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, December 2, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

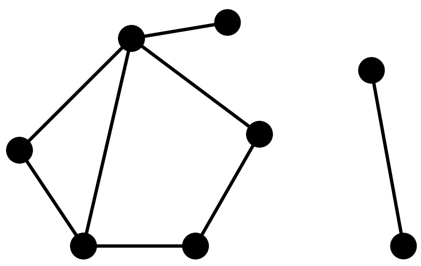
Some material contributed by Chris Hartman

The Rest of the Course
Overview

Final Topics

✓ - External Data
  - Previously, we dealt only with data stored in memory.
  - Suppose, instead, that we wish to deal with data stored on an external device, accessed via a relatively slow connection and available in chunks (data on a disk, for example).
  - How does this affect the design of algorithms and data structures?

(part) - Graph Algorithms
  - A **graph** models relationships between pairs of objects.
  - This is a very general notion. Algorithms for graphs often have very general applicability.



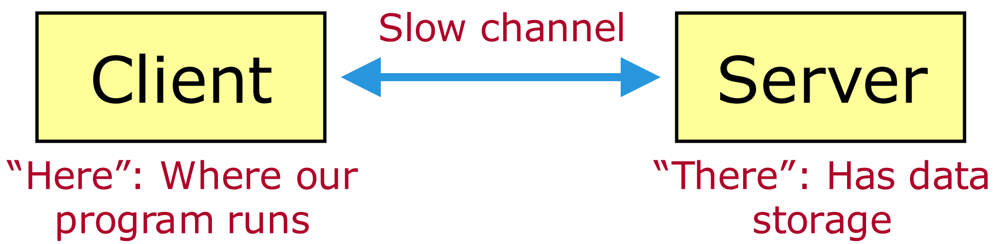
Drawing of
a Graph

This usage of "graph" has nothing to do with the graph of a function. It is a different definition of the word.

Review

Review
External Data [1/5]

We considered data that are accessed via a slow channel.



Typically, the channel transmits data in chunks: **blocks**.

Thus, *minimize the number of block accesses*.

External Sorting: Merge Sort variant

- Stable Merge works well with block-access data.
- Use temporary files for the necessary buffers.

External Table Implementation #1: Hash Table

- This works well: open hashing, with each bucket stored in as few blocks as possible. However it does not seem to be used much.

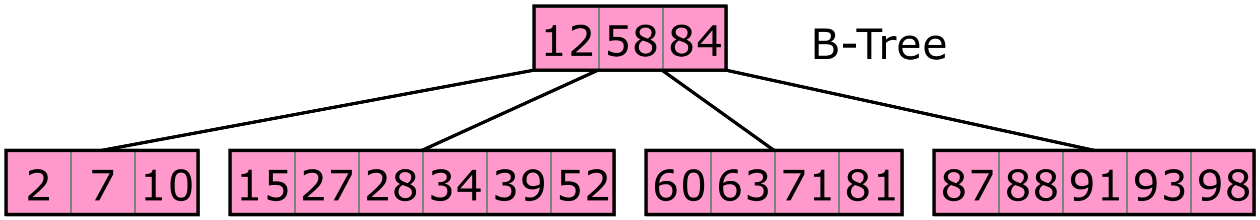
External Table Implementation #2: B-Tree

A **B-Tree of degree *m*** ($m \geq 3$) is a ceiling($m/2$) … m Tree.

- A node has ceiling($m/2$)−1 … m−1 items.
- Except: The root can have 1 … m−1 items.
- All leaves are at the same level.
- Non-leaves have 1 more child than # of items.
- The order property holds, as for 2-3 Trees and 2-3-4 Trees.
- Degree = max # of children = # of items in an **over-full** node.

2-3 Tree = B-Tree of degree 3. 2-3-4 Tree = B-Tree of degree 4.

Shown is a B-Tree of degree 7.



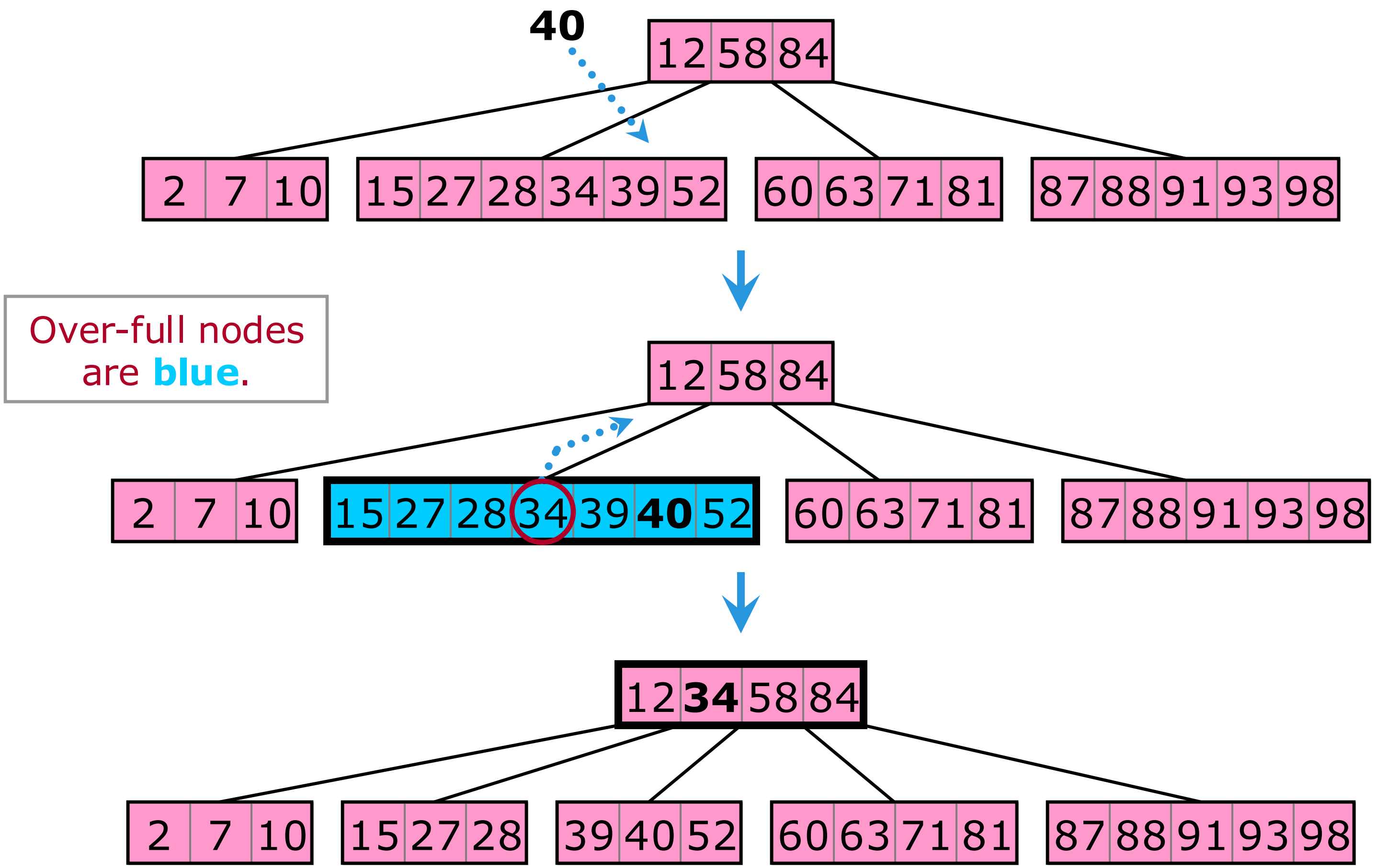
In practice, the degree may be much higher (for example, 50).

Review
External Data [3/5]

Illustration of B-Tree insert.

- Insert 40 into this B-Tree of degree 7.

An over-full node has 7 items.

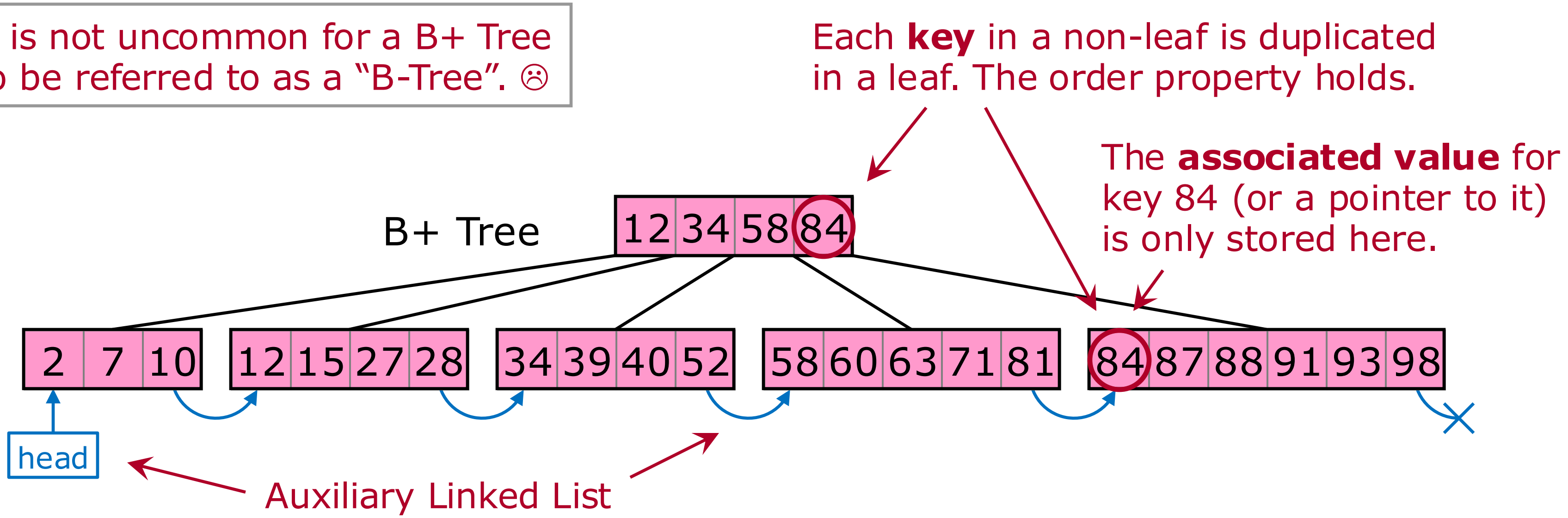


Review
External Data [4/5]

There are a number of B-Tree variations. A common one: **B+ Tree**. This is just like a B-Tree, except:

- Keys in non-leaf nodes are duplicated in the leaves, while maintaining the order property.
- Associated values—if any—are stored only in the leaves.
- Leaves are joined into an auxiliary Linked List. This minimizes the number of blocks we must read during a traversal.

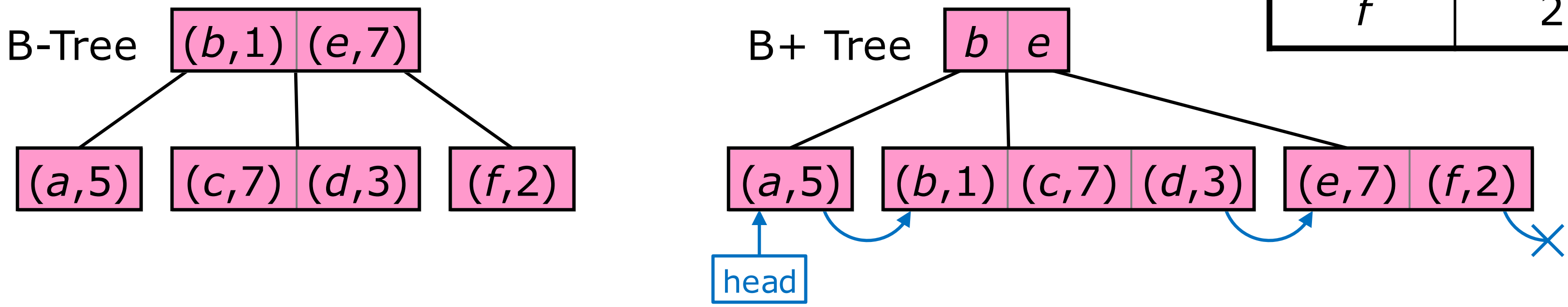
It is not uncommon for a B+ Tree to be referred to as a “B-Tree”. ☹



Review
External Data [5/5]

To the right is a Table dataset. Below on the left is a B-Tree holding this dataset. Below on the right is the corresponding B+ Tree. Both keys and associated values are shown.

Key	**Value**
a	5
b	1
c	7
d	3
e	7
f	2



Modern filesystems typically involve a B-Tree or variant internally. B+ Trees are a particularly common variant.

These trees are also used in relational-database implementation.

Module Overview
Graph Algorithms

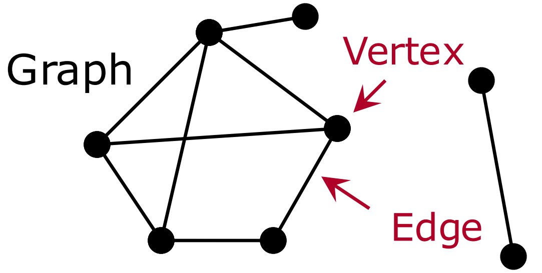
Topics

- ✓ Introduction to Graphs
- Graph Traversals
- Spanning Trees
- Other Graph Topics

Introduction to Graphs [1/3]

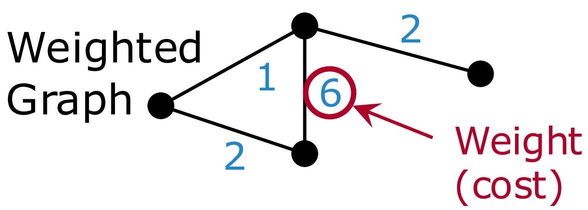
A **graph** consists of **vertices** and **edges**.

- An edge joins two vertices: its **endpoints**.
- 1 **vertex**, 2 vertices (Latin plural).
- Two vertices joined by an edge are **adjacent**; each is a **neighbor** of the other.



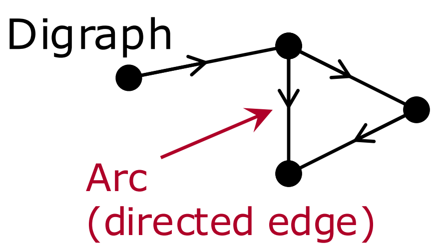
In a **weighted graph**, each edge has a **weight** (or **cost**).

- The weight usually gives the resource expenditure required to use that edge.
- We typically choose edges to minimize the total weight of some kind of collection.



If we give each edge a direction, then we have a **directed graph**, or **digraph**.

- Directed edges are called **arcs**.

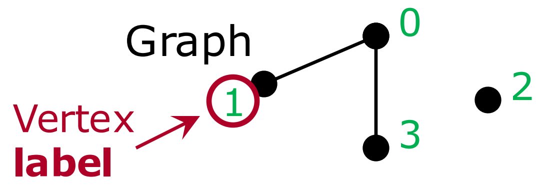


Introduction to Graphs [2/3]

Two common ways to represent graphs.

Both can be generalized to handle digraphs.

Graph



Adjacency matrix. 2-D array of 0/1 values.

- "Are vertices *i*, *j* adjacent?" in $\Theta(1)$ time.
- Finding all neighbors of a vertex is slow for large, sparse graphs.

Adjacency Matrix

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency lists. List of lists (arrays?).
List *i* holds neighbors of vertex *i*.

- "Are vertices *i*, *j* adjacent?" in $\Theta(\log N)$ time if lists are sorted arrays; $\Theta(N)$ if not.
- Finding all neighbors can be faster.

N: the number of vertices.

Adjacency Lists

0: 1, 3
1: 0
2:
3: 0

Review
Introduction to Graphs [3/3]

When we analyze the efficiency of graph algorithms, we consider *both* the number of vertices and the number of edges.

- N = number of vertices
- M = number of edges

When analyzing efficiency, we consider adjacency matrices & adjacency lists separately.

The *total* size of the input is:

- For an adjacency matrix: N^2. So $\Theta(N^2)$.
- For adjacency lists: $N + 2M$. So $\Theta(N + M)$.

Some particular algorithm might have order (say) $\Theta(N + M \log N)$.

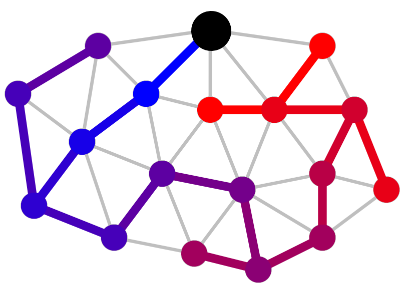
Graph Traversals

We covered Binary Tree traversals: preorder, inorder, postorder.

We **traverse** graphs as well: visit each vertex once.

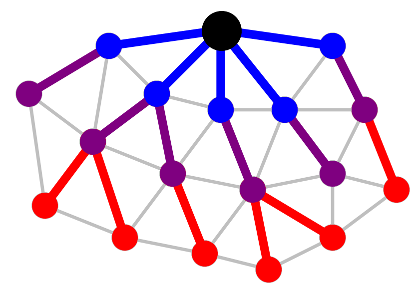
Depth-first search (**DFS**)

Walk along edges, visiting vertices as we go. When there is nowhere new to go, backtrack.



Breadth-first search (**BFS**)

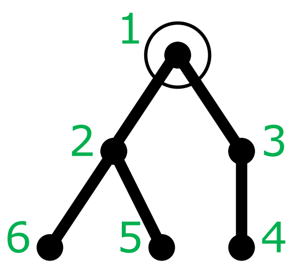
Proceed along all edges from a vertex, visiting each of its neighbors. Then visit its neighbors' neighbors, etc.



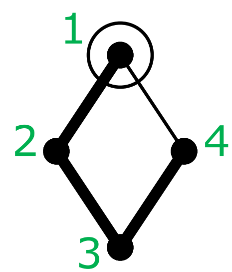
In both cases, we prefer to visit lower-numbered vertices first.

DFS has a natural recursive formulation:

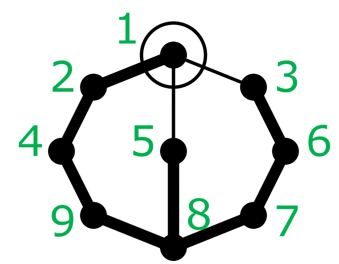
- Given a *start* vertex, visit it, and mark it as visited.
- For each of the start vertex's neighbors:
  - If this neighbor is unvisited, then do a DFS with this neighbor as the start vertex.



DFS: 1, 2, 5, 6, 3, 4



DFS: 1, 2, 3, 4



DFS: 1, 2, 4, 9, 8, 5, 7, 6, 3

We will write a traversal by listing the vertex labels in the order they are visited.

Recursion can be eliminated with a Stack—of course. But we can be more intelligent than the brute-force method.

Algorithm DFS [non-recursive]

- Mark all vertices as unvisited.
- For each vertex:
  - Do algorithm DFS’ with this vertex as *start*.

Algorithm DFS’ [non-recursive]

- Set Stack to empty.
- Push *start* vertex on Stack.
- Repeat while Stack is non-empty:
  - Pop top of Stack.
  - If this vertex is not visited, then:
    - Visit it.
    - Push its not-visited neighbors on the Stack.

This part is all we need, if the graph is **connected** (all one piece).

The above makes it work for all graphs, including **disconnected** graphs.

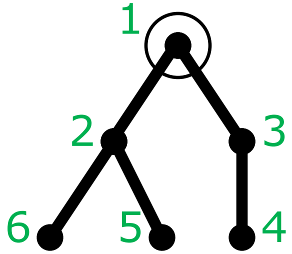
TO DO

- Based on the above, write a non-recursive function to do a DFS on a graph, given adjacency lists.

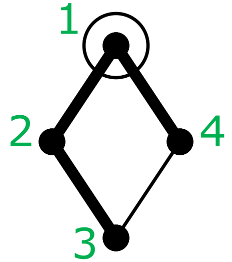
Done. See `graph_traverse.cpp`*.*

Graph Traversals
BFS [1/2]

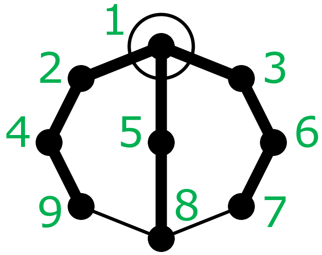
We can easily do a BFS manually by keeping track of those vertices for which we have looked at all neighbors.



BFS: 1, 2, 3, 5, 6, 4



BFS: 1, 2, 4, 3



BFS: 1, 2, 3, 5, 4, 6, 8, 9, 7

Graph Traversals
BFS [2/2]

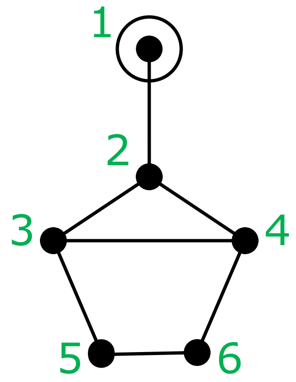
TO DO

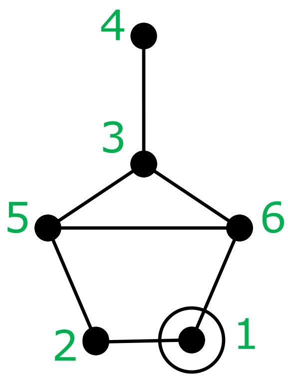
- Modify our DFS function to do BFS.
  - BFS reverses the priority of neighbors of vertex visited most recently vs. neighbors of vertices visited earliest.
  - Thus, replace the Stack with a Queue.

Done. See `graph_traverse.cpp`*.*

Graph Traversals
Try It! [1/2]

For each graph below, write the order in which the vertices will be visited in a DFS and in a BFS.

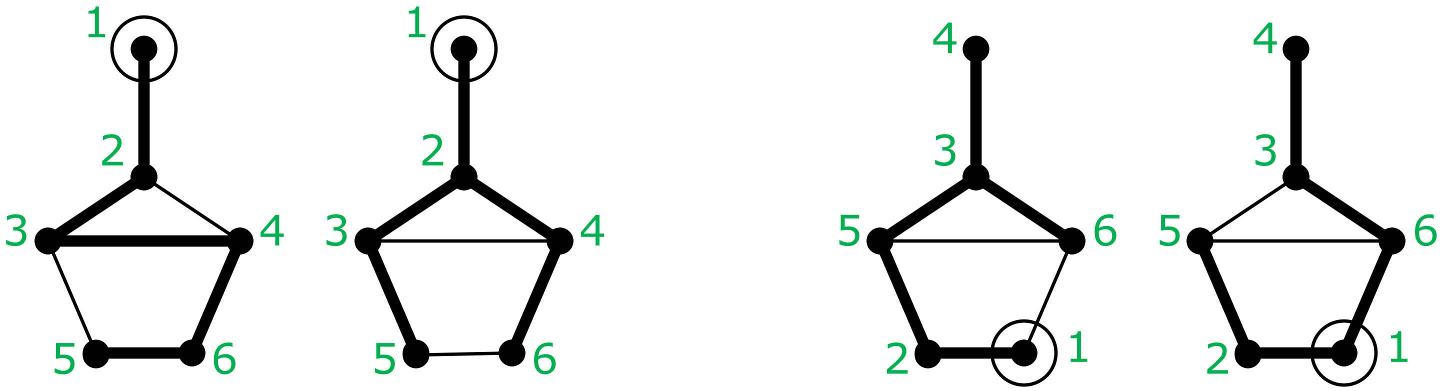




Answers on next slide.

Graph Traversals
Try It! [2/2]

For each graph below, write the order in which the vertices will be visited in a DFS and in a BFS.



Answers

DFS: 1, 2, 3, 4, 6, 5

BFS: 1, 2, 3, 4, 5, 6

DFS: 1, 2, 5, 3, 4, 6

BFS: 1, 2, 6, 5, 3, 4

Graph Traversals
Efficiency [1/2]

What is the order of our DFS & BFS algorithms, when given adjacency lists?

Treat *push*/*pop* & *enqueue*/*dequeue* as constant time.

- *Push* & *enqueue* may be amortized constant time, due to reallocate-and-copy, but since we are doing a lot of *push*/*enqueue* operations, they average out to constant time.

The concept of amortized constant time is very useful in reasoning of this kind.

We process each vertex.

- There are N of these.

We also do *push* and *pop* operations.

- Each time we *push*—or check if we should *push*—we are moving across an edge from one vertex to another. There are two directions to move across an edge: toward one endpoint or the other.
- So the number of *push* operations is no more than $2M$.
- The number of *pop* operations is the same.

Conclusion. Each algorithm is $\Theta(N + 2M) = \Theta(N + M)$.

Graph Traversals
Efficiency [2/2]

What would the order of our DFS & BFS algorithms be, if they were given an adjacency matrix?

Abbreviated Argument

- We process each vertex. There are N vertices.
- When looking for vertices to push, we examine an entire row of the adjacency matrix.
- Eventually, we will examine every entry of every row: N^2.

Conclusion. Each algorithm is $\Theta(N + N^2) = \Theta(N^2)$.

Regardless of whether it is given adjacency lists or an adjacency matrix, the order of each algorithm is the same as the size of the input—which is the fastest efficiency category possible for an algorithm that reads all of its input.