Introduction to Graphs
Graph Traversals

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, December 7, 2009

Glenn G. Chappell

Department of Computer Science

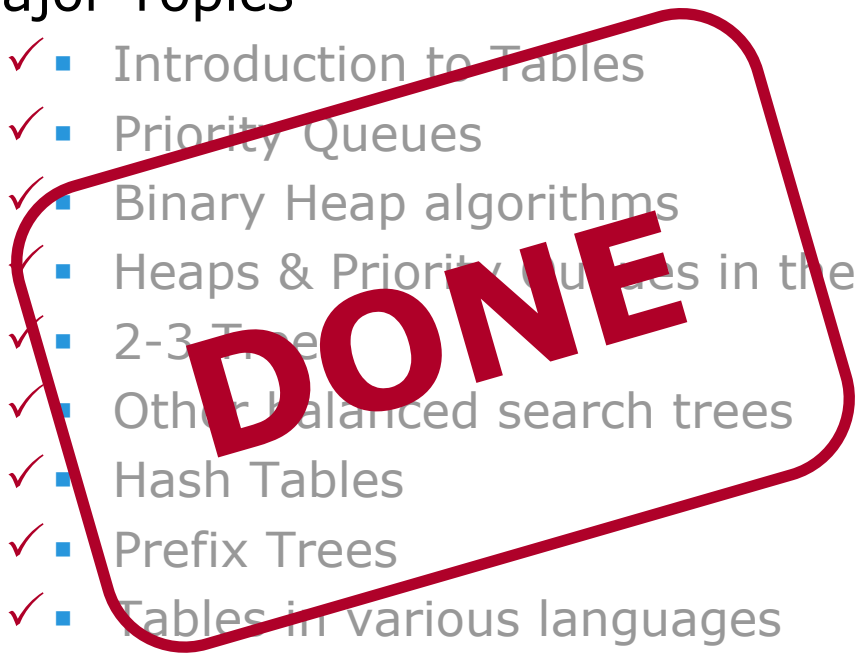
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- ✓ 2-3 Trees
- ✓ Other balanced search trees
- ✓ Hash Tables
- ✓ Prefix Trees
- ✓ Tables in various languages



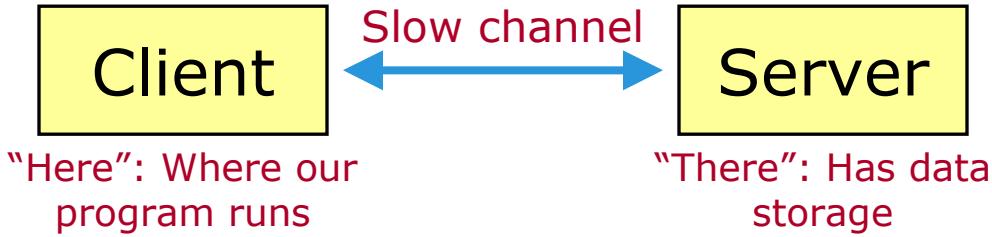
The Rest of the Course
Overview

Two Topics

- ✓ External Data
  - Throughout this semester, we have dealt-with data stored in memory.
  - What if we store data on an external device, accessed via a (relatively) slow connection. How does this change the design of algorithms and data structures?
- Graph Algorithms
  - A **graph** is a way of modeling relationships between pairs of objects.
  - This is a very general notion; thus, algorithms for graphs often have very general applicability.

Review
External Data — Introduction, Sorting, Tables: Hash Table

We considered **data** that are accessed via a slow channel.



- Overriding concern: **Minimize use of the channel.**

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

- Thus: **Minimize the number of block accesses.**

Sorting

- Stable Merge works well with block-access data.
- Use temporary files for the necessary buffers.
- Result: Reasonably efficient external Merge Sort.

Table Implementation — Hash Table

- Avoid open addressing.
- Each bucket takes one block (or some small number of blocks?).

External Data — Tables: B-Trees [1/3]

A **B-Tree of degree *m*** (*m* is odd) is essentially an (*m*+1)/2 … *m* Tree.

- Each node has (*m*−1)/2 up to *m*−1 items.
- Exception: The root can have 1 … *m*−1 items.
- All leaves are at the same level.
- All non-leaves have 1 more child than # of items.
- Order property holds, as for 2-3 Trees and 2-3-4 Trees.

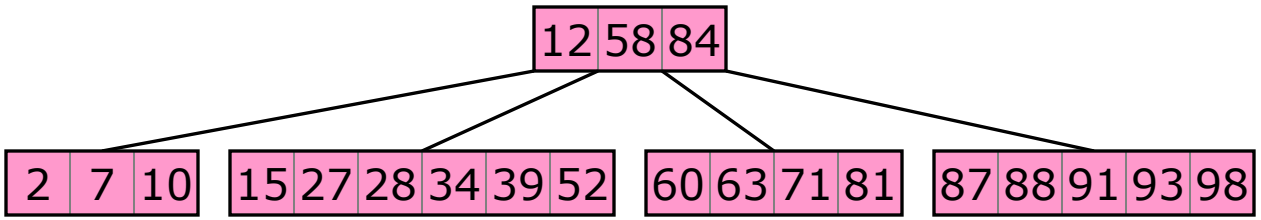
Terminology varies. This is what the text uses.

A 2-3 Tree is precisely a B-Tree of degree 3.

- Degree = max # of children = # of items in an *overfull* node.

Below is an example of a B-Tree of degree 7.

- In practice, a B-Tree could have much higher degree than this.



External Data — Tables: B-Trees [2/3]

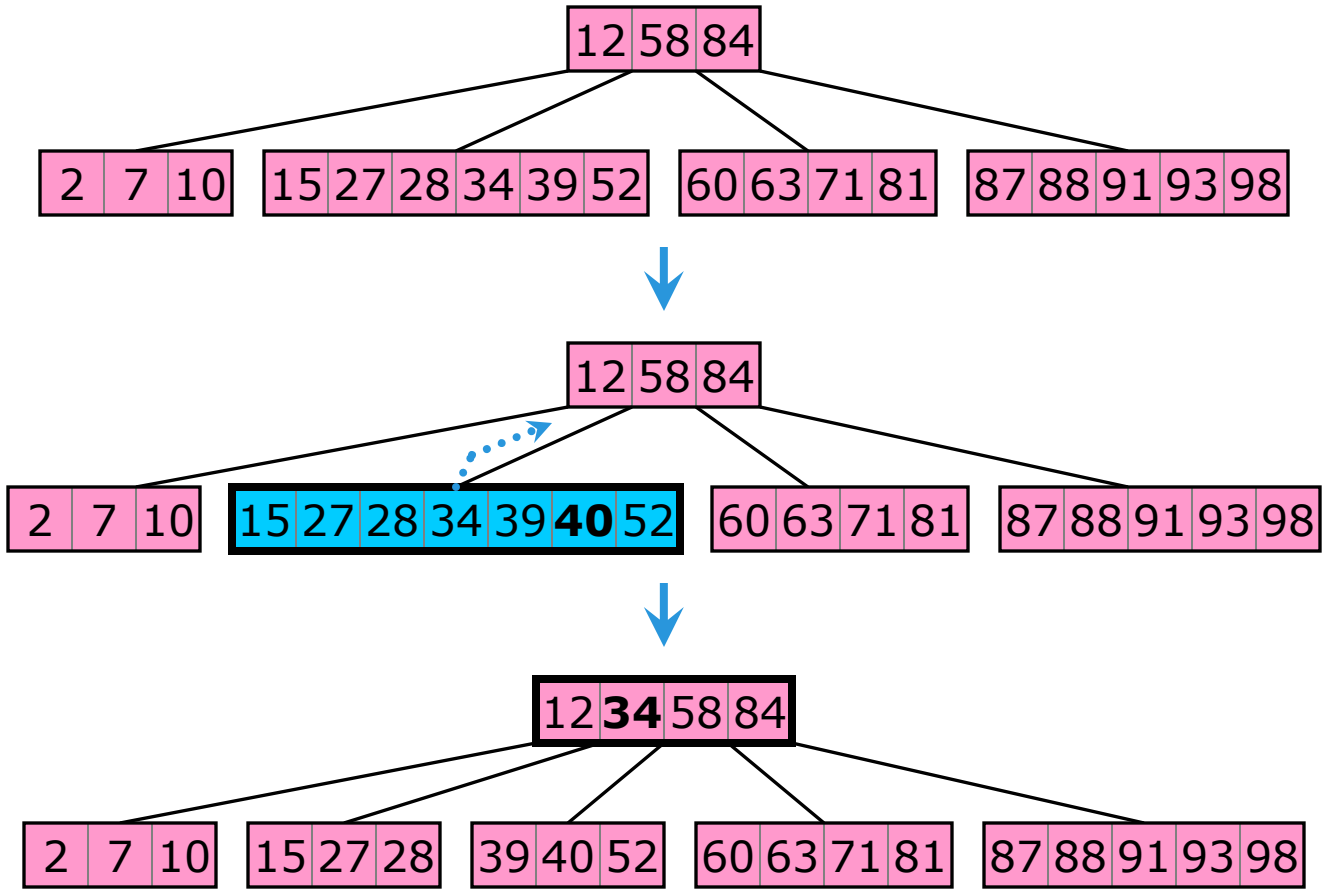
How B-Tree Algorithms Work

- Retrieve
  - Like other search trees.
- Traverse
  - Like other search trees (generalized inorder traversal).
  - Note that we need only read each block once, **if** we have in-memory storage for *h* blocks, where *h* is the height of the tree.
- Insert
  - Generalizes 2-3 Tree Insert algorithm:
    - Find the leaf that an item "should" go in.
    - Insert into this leaf.
    - If overfull, split it and move up the middle item, recursively inserting it in the parent node.
    - If the root becomes overfull, split and create a new root.
- Delete
  - Generalizes 2-3 Tree Delete algorithm.

External Data — Tables: B-Trees [3/3]

Here is an illustration of B-Tree insert.

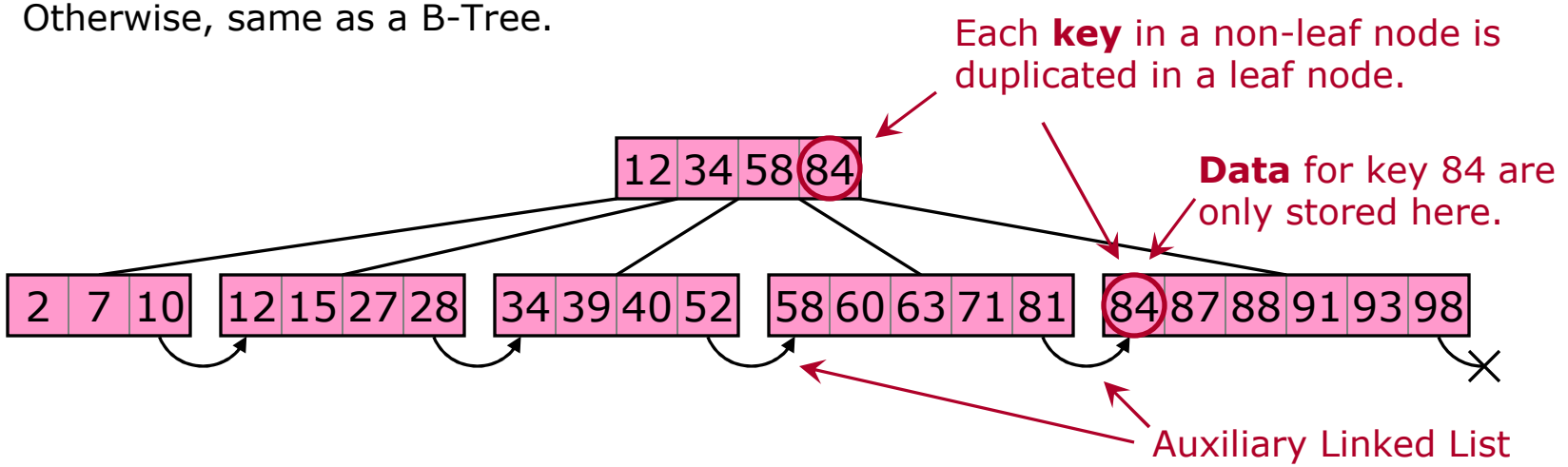
- We insert 40 into this B-Tree of degree 7.



Review
External Data — Tables: B+ Trees

A common variation of a B-Tree is a **B+ Tree**.

- All data (the non-key portion of the value) is in the leaves.
- Keys in non-leaf nodes are duplicated in the leaves.
- Leaves are typically joined in an auxiliary Linked List. This minimizes the number of block accesses required for a traversal.
- Otherwise, same as a B-Tree.



From the Wikipedia "B+ Tree" article (4 Dec 2009):

> btrfs, NTFS, ReiserFS, NSS, XFS, and JFS filesystems all use this type of tree for metadata indexing. Relational database management systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASI, PostgreSQL, Firebird and MySQL support this type of tree for table indices. Key-value database management systems such as Tokyo Cabinet and Tokyo Tyrant support this type of tree for data access.

Review
External Data — Tables: Reliability Issues

In practice, external storage is significantly less reliable than a computer's main memory.

Consider: What happens if the communications channel to external storage device fails in the middle of some algorithm?

- The data on the device may be left in an intermediate state.

How can we take this into account when designing algorithms that deal with data on external storage?

- As much as possible, the intermediate state of data should be either:
  - A **valid** state,
  - Or, if that is not possible, a state that can easily **fixed** (made valid).

In particular:

- **When writing the equivalent of a pointer to data on an external storage, write the data first, then the pointer.**

Introduction to Graphs
Definition

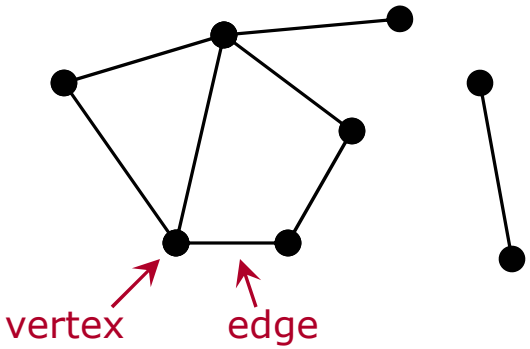
A **graph** is consists of **vertices** and **edges**. An edge joins two vertices.

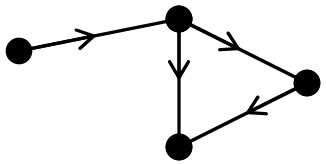
- An example of a graph is shown at right.

Sometimes we give each edge a direction.

- The result is a **directed graph** or **digraph.**

Graphs represent situations in which objects are related in pairs.





Introduction to Graphs
Applications

We use graphs to model:

- Networks
  - Vertices are nodes in network; edges are connections.
  - Examples
    - Communication
    - Transportation
    - Electrical
    - Web (edges are links)
- State Spaces
  - Vertices are states; edges are transitions between states.
  - See CS 451 for more info.
- Generally, situations in which objects are related in pairs:
  - Vertices are people, edges indicate relationships (friendship? common work?)
  - Vertices are events at a conference; edges join events that cannot be held simultaneously.
  - Vertices are data structure nodes; (directed) edges indicate (owning?) pointers.

Two common ways to represent graphs:

- **Adjacency matrix**. 2-D array of Boolean values. Entry (i, j) answers the question "Does edge (i, j) exist?"
  - Answer "does edge (i, j) exist?" in $O(1)$.
  - Finding all neighbors of a vertex can be slow for large, sparse graphs.
    - A **sparse** graph is one with relatively few edges.
  - Space used: $O(n^2)$, where n = number of vertices.
    - Note: For graphs in general, we cannot do better than this.
- **Adjacency list**. Array of lists. Entry i is a list of the neighbors of vertex i.
  - Answer "does edge (i, j) exist?" in $O(n)$.
  - Finding all neighbors of a vertex is fast.
  - Much better space usage for large, sparse graphs.

Both of these can be tweaked in obvious ways to handle digraphs.

Other graph representations are used.

Graph Traversals
Introduction

We have discussed traversals of Binary Trees.

- Preorder, inorder, postorder.

We traverse graphs as well.

- To "traverse" here means to visit each vertex once.
- Traditionally, graph traversal is viewed in terms of a "search".

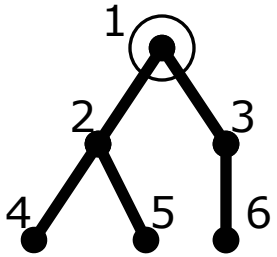
Two kinds of graph traversals.

- Depth-first search (DFS).
  - "Last visited, first explored."
  - Like preorder tree traversal.
  - When we visit a vertex, give priority to visiting its unvisited neighbors (and their unvisited neighbors, etc.).
- Breadth-first search (BFS).
  - "First visited, first explored."
  - Visit all of a vertex's unvisited neighbors before visiting their neighbors.

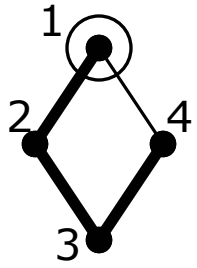
Graph Traversals
DFS [1/2]

DFS has a nice 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, do a DFS with this neighbor as the start vertex.



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



DFS: 1, 2, 3, 4

We get a **DFS tree** (shown in bold above).

DFS is convenient if we think about traveling through the graph, minimizing the number of edges we cross.

Graph Traversals
DFS [2/2]

We can, as usual eliminate recursion using a Stack.

- "Last visited, first explored."
- Then we have an iterative DFS algorithm using a local Stack.
  - And we can use it in a more intelligent way than the "brute-force" recursion elimination method.

Algorithm

- 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.

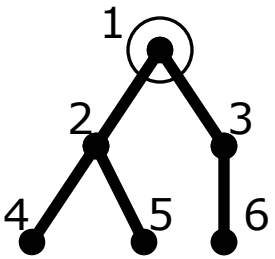
TO DO

- Write a non-recursive function to do a DFS on a graph, given an adjacency matrix.

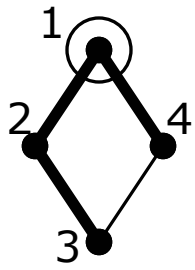
Done. See **`graphtraverse.cpp`**, *on the web page.*

Graph Traversals
BFS

BFS is "first visited, first explored".



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



BFS: 1, 2, 4, 3

Thus: replace the Stack with a Queue.

BFS is not as nice in several ways.

- No elegant recursive formulation.
- Not a convenient way to travel around a graph.

But BFS is *useful*.

- BFS is good for finding the shortest paths to other vertices.
- Also, looking for things "nearby first".

TO DO

- Modify our DFS function to do BFS.

Done. See **`graphtraverse.cpp`**, *on the web page.*

Graph Traversals
Generalization: Shortest Path

Sometimes we place **costs** (or **weights**) on edges of a graph.

- Cost might represent distance, time, money, etc.
- In general, the "cost" of an edge is how much resource expenditure it takes to "use" the edge in some way.
- We want to do things in a way that minimizes the total cost.

Example: Shortest Path

- Use a variation on BFS called **Dijkstra's Algorithm**.
  - Edsger Dijkstra, 1959.
- Choose a start vertex.
- Label each vertex with the length of the shortest known path from the start to it.
  - So, for now, start gets 0, others get ∞ (no path is known).
- Repeat:
  - Among the unvisited vertices, find the one with the smallest label ("me" below).
  - For each neighbor, if my label plus the cost of the edge between us is less than its label, then replace its label with the new value.
  - Mark me as visited.