The Rest of the Course
External Data
Introduction to Graphs

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 25, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- ✓ 2-3 Trees
- ✓ Other self-balancing search trees
- ✓ Hash Tables
- ✓ Prefix Trees
- ✓ Tables in the C++ STL & Elsewhere

Review

Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Review
Introduction to Tables

A **Table** allows for arbitrary key-based look-up.

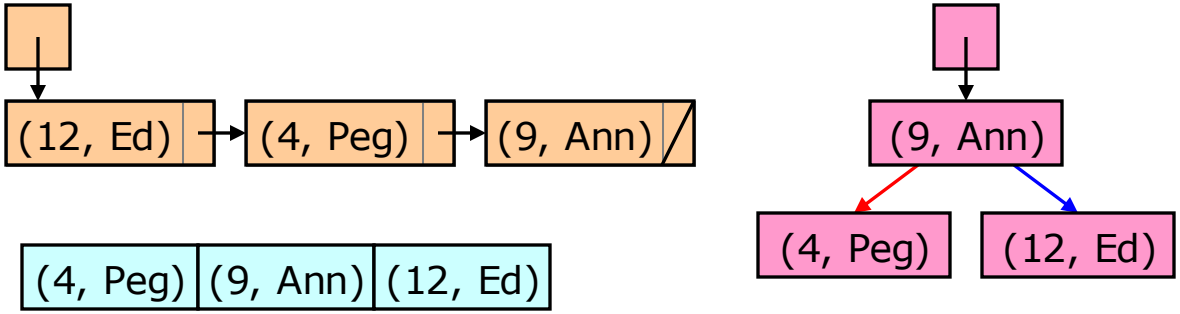
Three single-item operations: retrieve, insert, delete by key.

A Table implementation typically holds **key-value pairs**.

Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations



Three ideas for improving efficiency:

1. Restricted Table → Priority Queues
2. Keep a tree balanced → Self-balancing search trees
3. Magic functions → Hash Tables

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- ✓ 2-3 Trees
- ✓ Other self-balancing search trees
- ✓ Hash Tables
- ✓ Prefix Trees
- ✓ Tables in the C++ STL & Elsewhere

Several lousy implementations

Idea #1: Restricted Table

Idea #2: Keep a tree balanced

Idea #3: Magic functions

A special-purpose implementation: "the Radix Sort of Table implementations"

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - ✓ **2-3 Tree**
      - Up to 3 children
    - ✓ **2-3-4 Tree**
      - Up to 4 children
    - ✓ **Red-Black Tree**
      - Binary Tree representation of a 2-3-4 Tree
  - Or balance up front, for a strongly balanced Binary Search Tree again:
    - ✓ **AVL Tree**
- Alternatively, forget about trees entirely:
  - ✓ **Hash Table**
- Finally, "the Radix Sort of Table implementations":
  - ✓ **Prefix Tree**

DONE

Idea #2: Keep a tree balanced

Idea #3: Magic functions

Later, we cover other self-balancing search trees: B-Trees, B+ Trees.

The Rest of the Course

The Rest of the Course
From the First Day of Class: Course Overview — Topics

The following topics will be covered, roughly in order:

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting
- Data Abstraction
- Basic Abstract Data Types & Data Structures:
  - Smart Arrays & Strings
  - Linked Lists
  - Stacks & Queues
  - Trees (various kinds)
  - Priority Queues
  - Tables

DONE

Goal: Practical generic containers

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

- **Briefly: external data, graph algorithms.**

The Rest of the Course
Overview

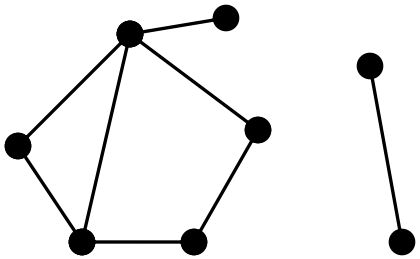
In the time remaining, we look briefly at two more topics.

External Data

- Previously, we dealt only with data stored in memory.
- Suppose 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?

Graph Algorithms

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



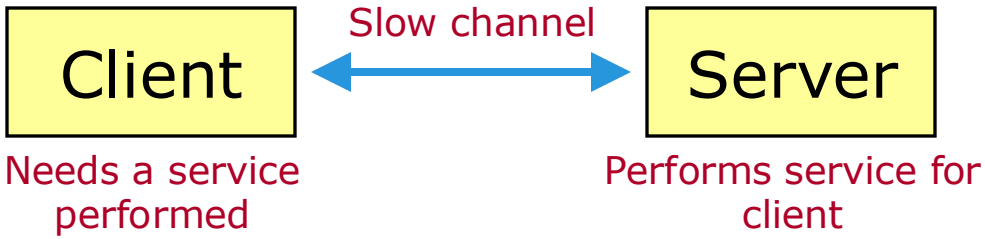
Drawing of a Graph

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

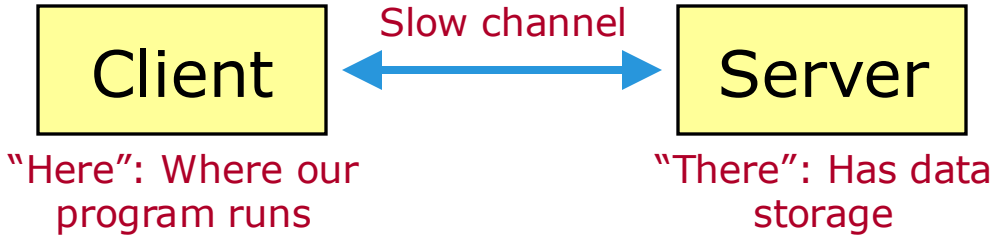
External Data

It is common for computing resources to be joined by a relatively poor (slow, perhaps unreliable) communication channel.

It can be helpful to think in terms of a **client/server** paradigm.



Now we consider *data* that are accessed via such a slow channel.



Overriding concern: *minimize use of the channel*.

This has a significant impact on data structure & algorithm design.

Introduction — External Storage

External storage is storage that is not part of main memory.

Compared with main memory, external storage is typically:

- More permanent.
- Larger.
- Much slower.

External storage is usually accessed in sizable chunks: **blocks**. It can be expensive (slow) to access a data item, but there is little additional cost to access other items in the same block.

In our discussion, we will:

- Do all processing on the client side of the channel.
- Usually *not* expect to hold an entire dataset in memory at once.
- Expect essentially unlimited storage to be available on the server.

In practice, external storage read & write operations are significantly less reliable than those to memory.

Q. What happens if a write fails in the middle of an algorithm?

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

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

A. The intermediate state of data should be either:

- A **valid** state,
- Or, if that is not possible, a state that can easily be **fixed**.

In particular: **when writing the equivalent of a pointer to data in external storage, write the data first, then the pointer.**

Introduction — Two Tasks

Two tasks have occupied much of our time this semester:

- Sorting.
- Table implementation.

We consider these in the context of data on external storage.

Sorting. Sort a file—perhaps line by line.

Table Implementation. Store a large Table externally.

We are interested in time efficiency of sorting, as well as the various Table operations—traverse, retrieve, insert, delete.

It may be helpful to change our model of computation, making our basic operation the block read/write.

External Data Sorting

We can do a reasonably efficient **Stable Merge** on two files.

- General-purpose Stable Merge uses an additional buffer. We can use temporary files for this.
- Stable Merge goes through both the data to be merged and the buffer in sequential order.
- Since we access data in order, a Stable Merge operation should not read/write any single block more than once.

This idea gives us a reasonably efficient **external Merge Sort**. Note that this will be stable.

External Data
Table Implementation — Options

Options for implementing an external Table are basically as before: Hash Tables and self-balancing search trees. But details vary.

Hash Tables

- If there are no collisions, then the hash function tells us where an item is. Retrieve it with a single block read.
- Collision resolution is cheap, if the key is stored in that same block—or one of a small number of blocks with known locations.
- Therefore: open hashing, with each bucket stored in as few blocks as possible.
- This idea does not seem to be used very much. I am not sure why.

Self-Balancing Search Trees

- Red-Black Trees are optimized for in-memory work. For external data, they require looking at too many blocks.
- Next we look at a more appropriate structure: *B-Trees*.

External Data
Table Implementation — B-Trees [1/4]

We can generalize 2-3 Trees for external storage by **making the nodes large**. (Perhaps we store one node per block?)

Q. Think: why are 2-3 Trees *nice*?

A. An over-full node splits into 2 small nodes + 1 item to move up.

Can we make a *nice* self-balancing search tree with larger nodes?

- Consider a 2-3 Tree (max # of items in a node: 3–1 = 2).
  Over-full (3 items) splits 1 + 1 + 1 to move up.
- If max # of items in a node = 4:
  over-full (5 items) splits 2 + 2 + 1 to move up: a 3-4-5 Tree.
- If max # of items in a node = 6:
  over-full (7 items) splits 3 + 3 + 1 to move up: a 4-5-6-7 Tree.
- And so on …

These are **B-Trees**. Max # of children is the **degree** of the B-Tree.

For $m \geq 3$, a **B-Tree of degree *m*** is a ceiling(m/2) … m Tree.
[R. Bayer & E. McCreight 1970]

- A node has ceiling(m/2)–1 … m–1 items.
- Except: The root can have 1 … m–1 items.
- The order property holds, as for 2-3 Trees.
- All leaves are at the same level.
- Each node is either a leaf or has its maximum number of children.
- Degree = max # of children = # of items in an over-full node.

Why "B"?

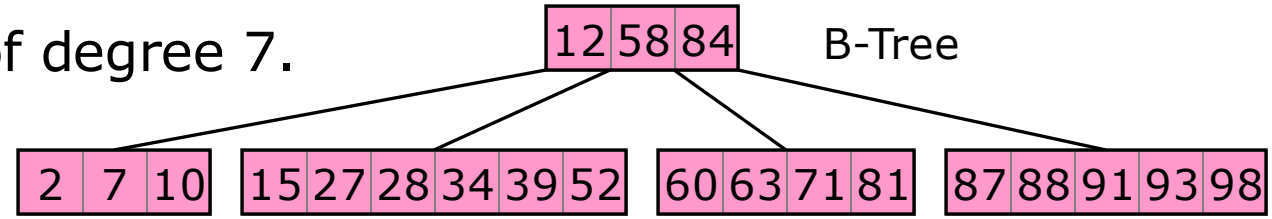
We do not know. Perhaps *balanced*, *broad*, *bushy*, *Bayer*, or *Boeing*—where Bayer & McCreight worked.

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

A B-Tree of degree 4 is a 2-3-4 Tree.

A B-Tree of degree 5 is a 3-4-5 Tree.

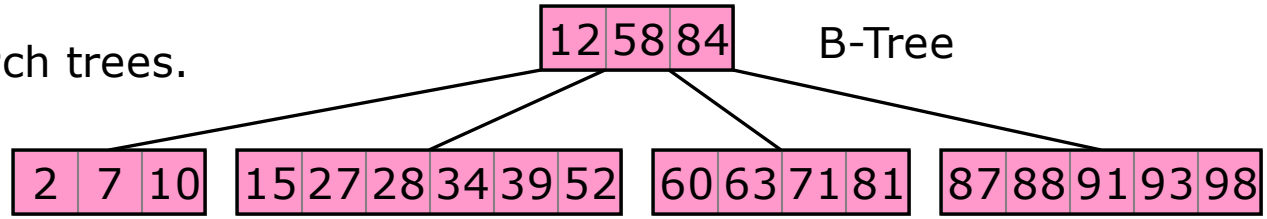
Shown is a B-Tree of degree 7.



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

How B-Tree Algorithms Work

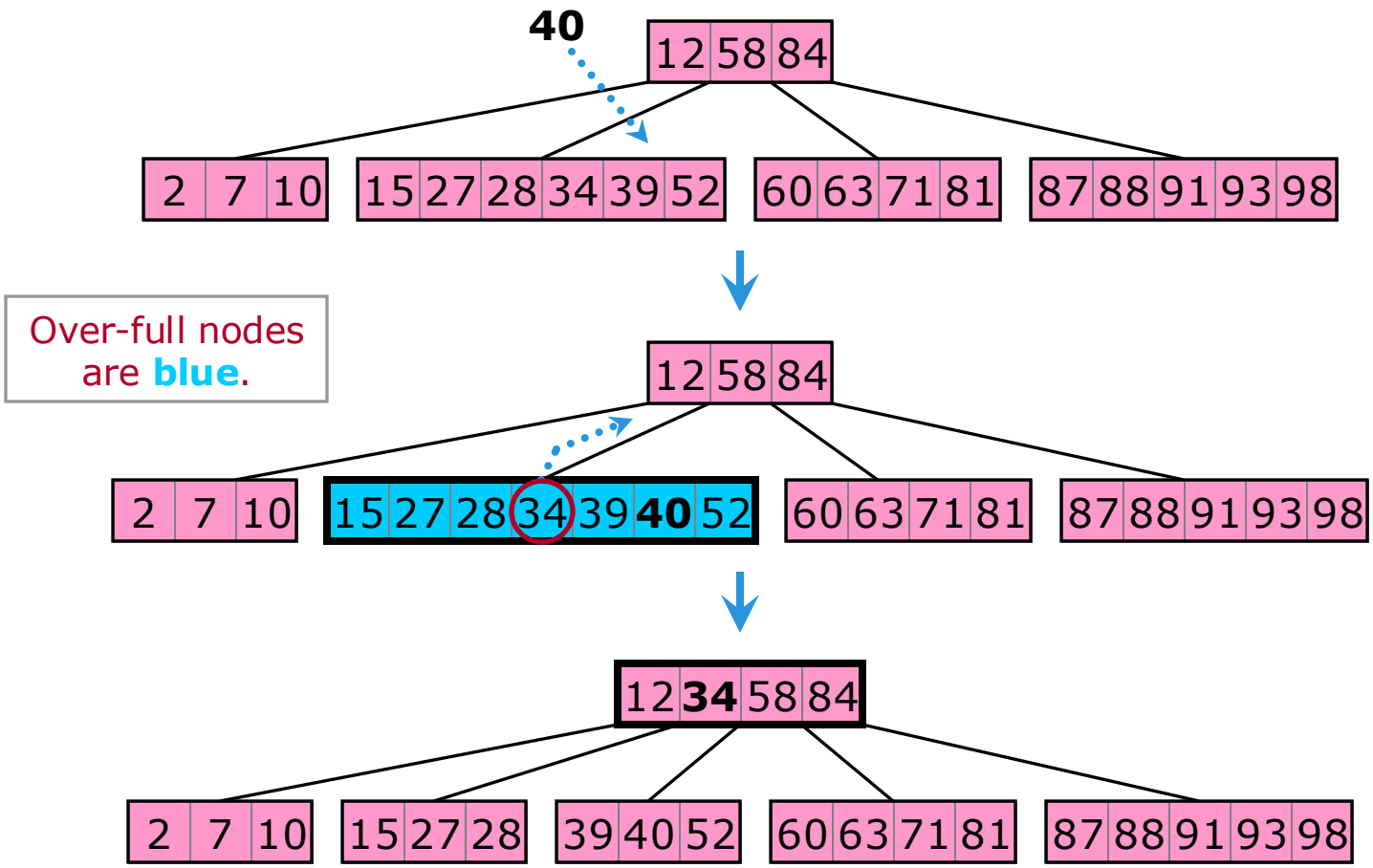
- Traverse
  - Like other search trees (generalize inorder traversal).
  - If we have in-memory storage for *h* nodes, where *h* is the height of the tree, then we only need to read each node once.
- Retrieve
  - Like other search trees.



- Insert
  - Generalizes 2-3 Tree Insert algorithm:
    - Find the leaf that an item *should* go in.
    - Insert into this leaf.
    - If over-full, then split the node, with **middle** item moving up. Recursively insert this item into the parent node.
    - If the root becomes over-full, then split and create a new root.
- Delete
  - Generalizes 2-3 Tree Delete algorithm. We will not cover the details.

Illustration of B-Tree insert.

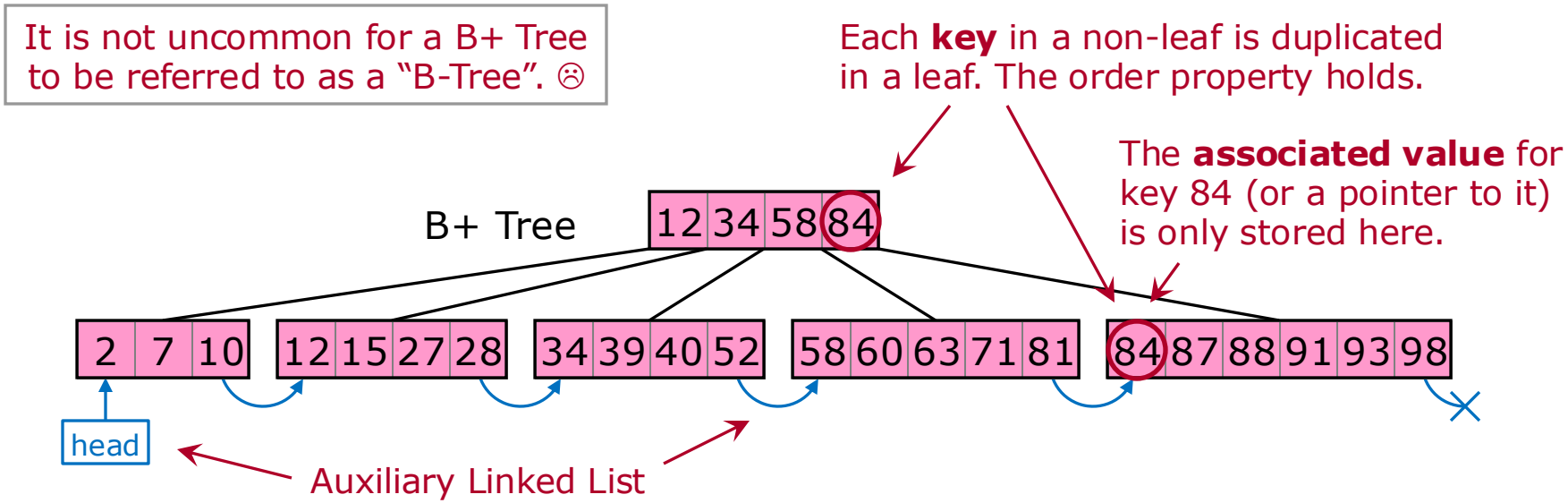
- Insert 40 into this B-Tree of degree 7.



External Data
Table Implementation — B+ Trees [1/2]

There are a number of B-Tree variations. Probably the most common are **B+ Trees**. These are just like B-Trees, 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.



External Data
Table Implementation — B+ Trees [2/2]

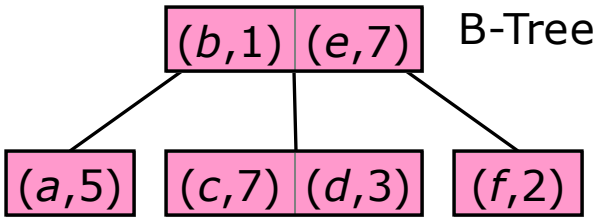
To fully illustrate the differences between B-Trees and B+ Trees, we picture two such trees holding the same dataset, with both keys and associated values shown.

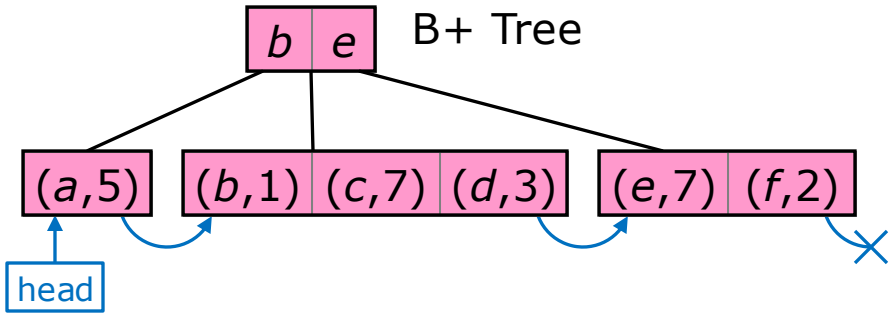
To the right is a Table dataset.

Below-left is a B-Tree holding this dataset.

Below-right is the corresponding B+ Tree.

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





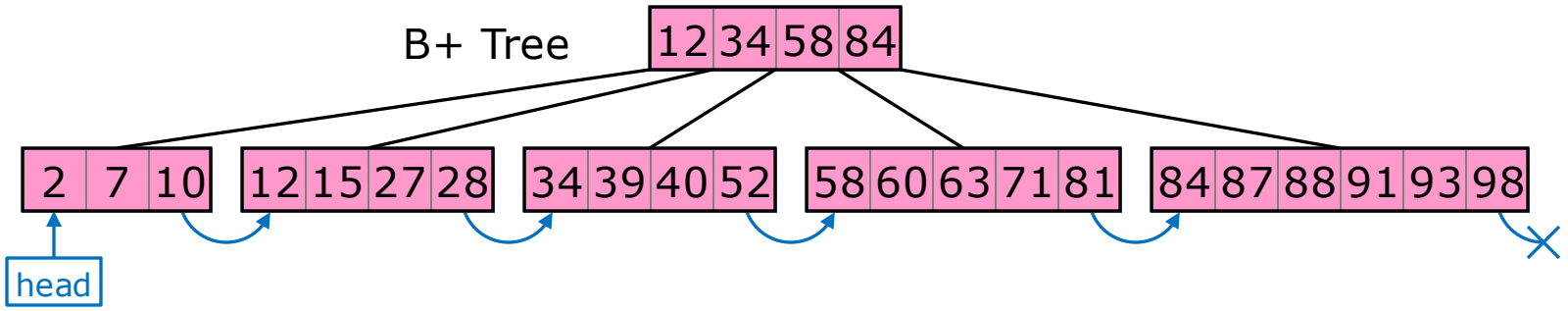
External Data
Table Implementation — Notes

For each of the covered external Table implementations, order of operations is the same as for the in-memory versions.

In particular, for a B-Tree or a B+ Tree, retrieve/insert/delete by key are Θ(log *n*), and traverse is Θ(*n*).

Modern filesystems typically involve a B+ Tree or variant internally. (Only major exceptions that I know of: Microsoft's FAT filesystems.)

These trees are also used in relational-database implementation.



Module Overview
Graph Algorithms

Topics

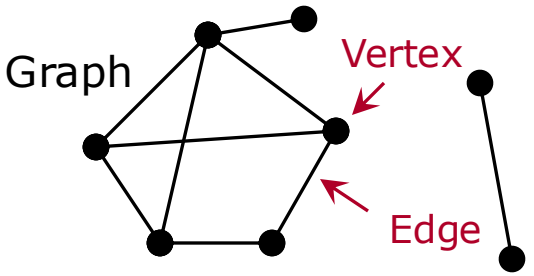
- Introduction to Graphs
- Graph Traversals
- Spanning Trees
- Other Graph Topics

Introduction to Graphs

Introduction to Graphs
Definition

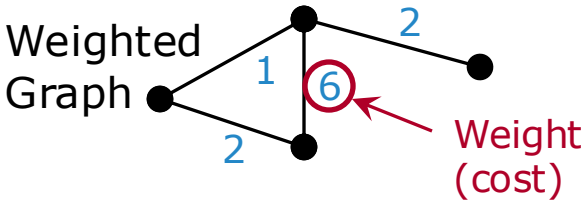
A **graph** consists of **vertices** and **edges**.

- 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.



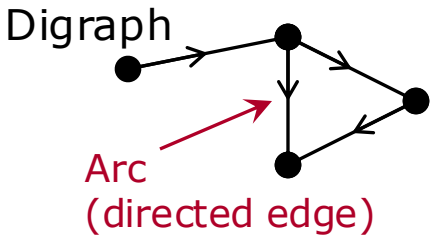
In a **weighted graph**, each edge has a **weight** (or **cost**).

- 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**.



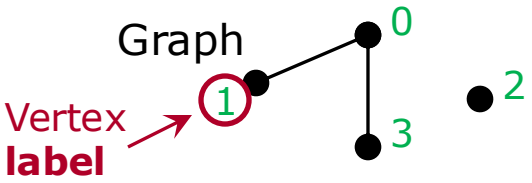
Introduction to Graphs
Applications

We use graphs to model:

- Networks
  - Vertices are nodes in network; edges are connections.
  - Examples
    - Communication
    - Transportation
    - Electrical
    - Worldwide Web (edges are links)
- State Spaces
  - Vertices are states; edges are transitions between states.
- Generally, situations in which things are related in pairs:
  - Vertices are data-structure nodes; directed edges indicate pointers.
  - Vertices are people, edges indicate relationships (friendship?).
  - Vertices are tasks or events; edges join pairs that cannot occur at the same time (e.g., because of conflicting resource needs).

Introduction to Graphs
Representations [1/3]

How do we represent a graph in a computer program?



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

- "Are vertices *i*, *j* adjacent?" in Θ(1) time.
- Finding all neighbors of a vertex is slow for large, sparse graphs.
  - **Sparse** graph: one with relatively few edges.

Adjacency Matrix

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

Labels are not part of the matrix.

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

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

Adjacency Lists

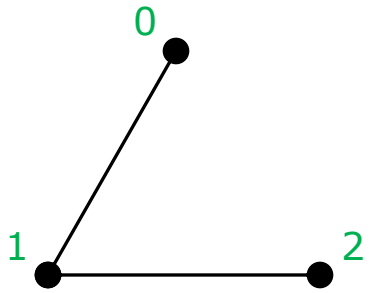
0: 1, 3
1: 0
2:
3: 0

N: the number of vertices (more on this soon)

Both adjacency matrices and adjacency lists can be generalized to handle digraphs.

Introduction to Graphs
Representations [2/3] (Try It!)

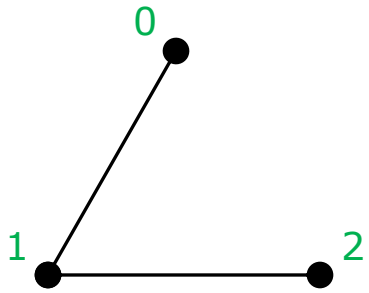
For the following graph, write (a) the adjacency matrix, and (b) adjacency lists.



Answers on the next slide.

Introduction to Graphs
Representations [3/3] (Try It!)

For the following graph, write (a) the adjacency matrix, and (b) adjacency lists.



Answers

(a)

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

(b)

0: 1
1: 0, 2
2: 1

Green *numbers are optional in the answer to part (a).*

When an algorithm takes a graph, what is our "*n*"?

The number of vertices? The number of edges? Some combination?

We consider *both* the number of vertices and the number of edges.

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

I use upper case (N, M) to make it clear that we are talking about vertices and edges, not the size of the input as a whole.

Adjacency matrices & adjacency lists are considered separately.

The *total* size of the input is:

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

The "2" is because each edge corresponds to two entries in the adjacency lists—one for each endpoint of the edge.

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