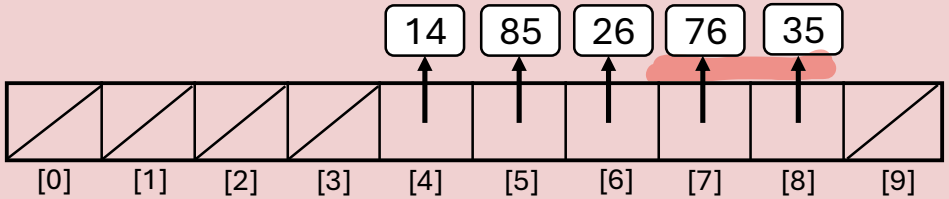
Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

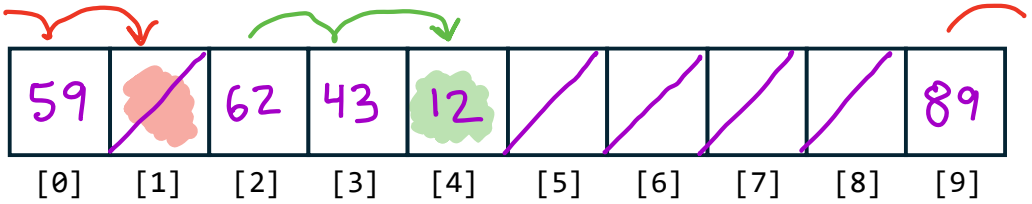
Suppose we add 14, 26, 35, 76, 85 (in some order) to a **probing** hash table (where their hash codes are their values). The state of the hash table is:



Which could *not* be the order these elements were added?

14, 85, 26, 76, 35	**(A)**	85, 26, 76, 35, 14	**(C)**
14, 26, 85, 35, 76	**(B)**	26, 14, 76, 85, 35	**(D)**

contains() in Probing Hash Tables



Always start at nominal index:

Best case: Nominal bucket contains element — contains(43) == true

Nominal bucket is empty — contains(16) == false

If nominal bucket contains different element, start probing until we find element or empty cell

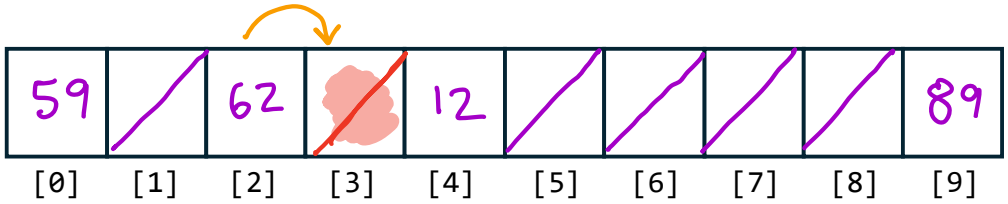
contains(12) == true

contains(19) == false

remove() in Probing Hash Tables

remove(43)

contains(12)



Just like other methods, start at nominal index and probe for element.

When we get there, we can't just remove the element (i.e., reassign cell to null).

- This could break other probing sequences

Instead: Use a "tombstone" to show element was there

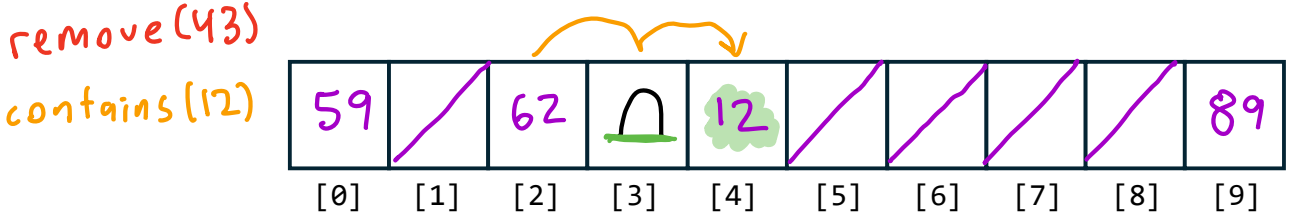
Incorporating Tombstones

When we add tombstones to our hash table in the `remove()` method, we'll need to account for them in our `add()` and `contains()` logic.

`add()`: Probe until we reach empty bucket or a tombstone (which we overwrite). Remove tombstones during resize

`contains()`: Probe past tombstones, since only empty buckets signal the end of a probing sequence.

* Tombstones count toward load factor.



Probing Hash Table Performance

The expected runtime of all operations in a probing hash table is also upper bounded by a function of λ, so it also amortizes to $O(1)$ with proper resizing.

Comparing Chaining vs. Probing:

- Probing has worse dependence on λ since nearby hash values can affect performance, not just exact collisions.
- Probing hash tables have smaller, centralized storage, so they can perform better due to hardware considerations like *cache locality* that fall outside of our scope.



Lecture 21: Graphs

CS 2110

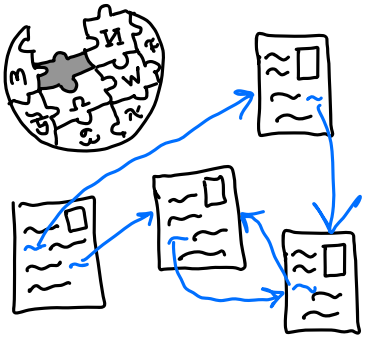
November 5, 2026

Today's Learning Outcomes

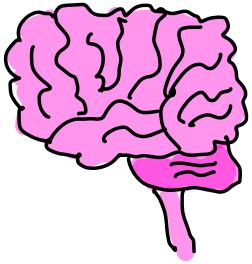
80. Identify substructures in graphs such as neighborhoods, paths, and cycles both visually and programmatically.

81. Describe adjacency list and matrix representations of a graph and compare the space/time complexities of operations on each of these representations.

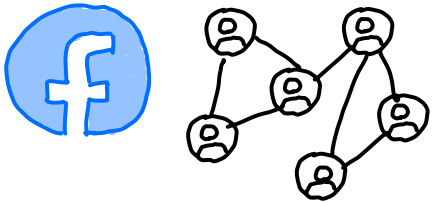
Modeling Structure in Real-World Settings



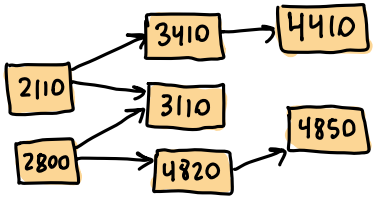
Wikipedia Links



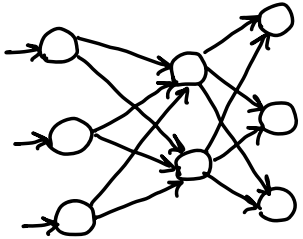
Neural Networks



Social Networks



Course Planning





Route Planning

Directed Graphs

A directed graph is a mathematical structure described by two sets.

Vertices (V): units we're connecting

(people, addresses, cities, webpages, ...)

Edges (E): directed connections between pairs of vertices

(friendships, roads, flights, prereqs)

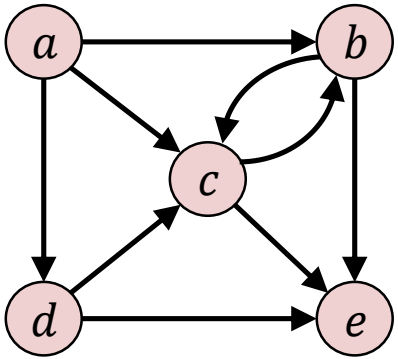
written as ordered pairs

(a,b) = a → b (tail → head)

not b → a

$V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (a,d), (b,c), (b,e), (c,b), (c,e), (d,c), (d,e)\}$



Other Graph Varieties

Undirected graphs: All edges connect in both directions



Graphs with self-loops: Edges that start/end at same vertex



Multigraphs: Have multiple parallel edges between the same pair of vertices (in same order)



Simple graphs don't have self-loops or parallel edges.

* We'll focus on simple directed graphs in CS2110.

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the maximum number of edges that a *simple* directed graph with 6 vertices can have?

6 choices for tail * 5 choices for head = 30 possible edges

More generally, a graph can have up to $|V| \cdot (|V|-1)$ $= O(|V|^2)$ edges

Connected graphs have

$|E| = O(|V|^2)$ ← upper bound

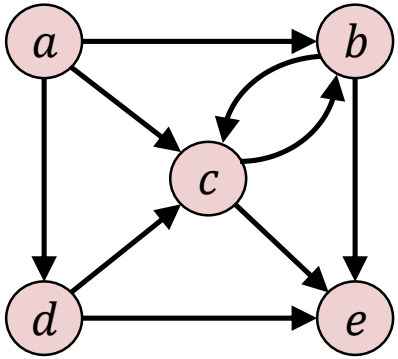
$|E| = \Omega(|V|)$ ← lower bound

Neighborhoods in Graphs

If $(u,v) \in E$, we say v is a neighbor (or an out-neighbor) of u.

u's neighborhood includes all of u's neighbors. $N_G(u)$

u's degree is the size of its neighborhood. $d_G(u)$



$N_G(c) = \{b, e\}$

$d_G(c) = 2$

Paths and Cycles

A path is a sequence of distinct contiguous edges in a graph.

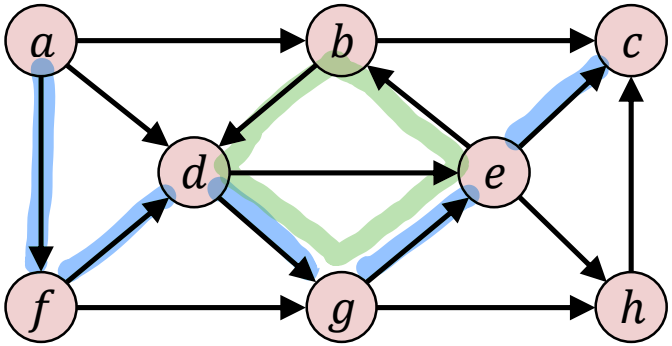
$P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{\ell-1}, v_\ell))$ where $(v_0, v_1) \in E$, $(v_1, v_2) \in E$, $(v_{\ell-1}, v_\ell) \in E$

source; same vertex; destination

length of path = # of edges (ℓ)

shorthand: $v_0 \to v_1 \to v_2 \to \ldots \to v_\ell$

a $v_0 \rightsquigarrow v_\ell$ path



Path $a \to f \to d \to g \to e \to c$ has length 5

Cycle $d \to g \to e \to b \to d$

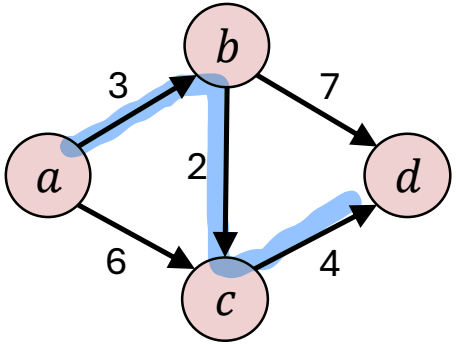
A cycle is a path with source = destination

Labeled/Weighted Graphs

Sometimes, we *label* the vertices or edges in a graph with extra info.

Common: Label edges with number
- cost, weight, length

Length of paths overloaded term that refers to sum of edge labels along path



shortest a ⇝ d path

a → b → c → d

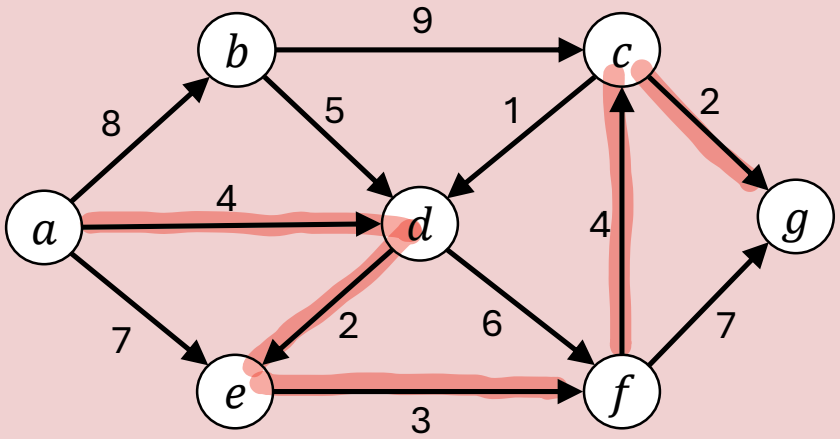
has length 9

Shortest path has minimum label sum, not necessarily fewest edges.

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the length of the shortest $a \rightsquigarrow g$ path in this graph?



15	(A)
16	(B)
17	(C)
18	(D)

A Graph ADT

Brainstorm: What operations should a Graph support?

Query: Access info

- check for vertex
- check for edge
- get # vertices
- get # edges
- get edge weight
- get degree
- check for path/cycle

Mutate: Change

- add vertex
- add edge
- remove vertex
- remove edge
- update weight

Iterate

- over all vertices
- over all edges
- over neighbors of one vertex
- over edges in a path

No built-in Graph ADT...
we'll need to build ourselves

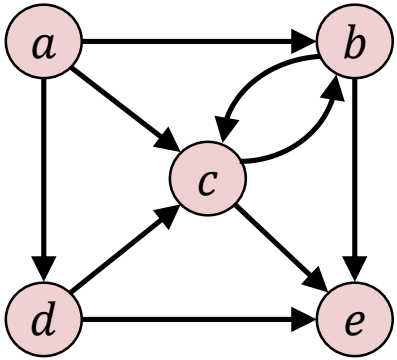


Coding Demo: Graph ADTs



Adjacency Matrix Representation

Use a 2D array/list structure to keep track of which edges are present



tail

head

	a	b	c	d	e
a		✓	✓	✓	
b			✓		✓
c		✓			✓
d			✓		✓
e					

can be booleans to indicate presence/absence or Edge variables with references/null

AdjMatrixGraph State

```
public class AdjMatrixGraph implements Graph<AMVertex, Edge<AMVertex>> {
  /** A vertex in this graph */
  public class AMVertex implements Vertex<Edge<AMVertex>> {
    /** The label of this vertex */
    private final String label;

    /** The index of this vertex in the edges 2D ArrayList */
    private final int index;
  }

  /** Map associating labels with AMVertex objects. */
  private final HashMap<String, AMVertex> vertices;

  /** edges.get(i).get(j) = reference to Edge (i,j), or null if
   *  this edge is not present in the graph. */
  private final ArrayList<ArrayList<Edge<AMVertex>>> edges;
}
```

vertex knows position in matrix

General usage pattern

1. Use vertices map to locate vertex O(1)
2. Access index in vertex O(1)
3. Access entry(s) in edges to query/update graph structure

Adjacency Matrix Operation Complexities

```
public class AdjMatrixGraph implements Graph<...> {
  public class AMVertex implements Vertex<...> {
    private String label;
    private int index;

    // methods
  }

  private HashMap<String, AMVertex> vertices;
  private ArrayList<ArrayList<Edge>>> edges;

  // methods
}
```

Count/Iterate vertices:

$O(1)$ `vertices.size()` $O(|V|)$

Count/Iterate neighbors:

$O(|V|)$ linear scan over matrix row

Check for an edge:

$O(1)$ using vertices and index

Count/Iterate edges:

$O(|V|^2)$ scan over whole matrix

Add edge:

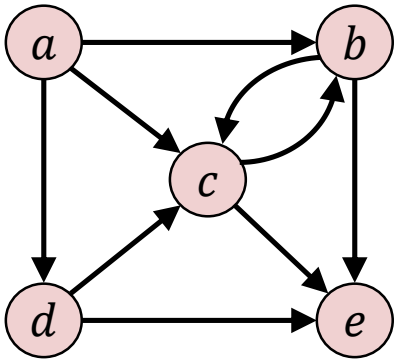
$O(1)$ using vertices and index

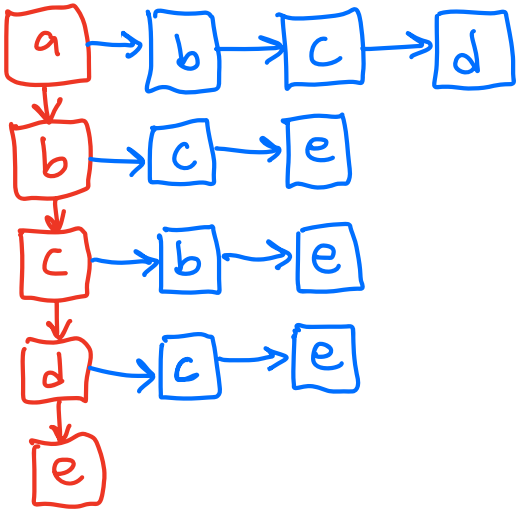
Add vertex:

$O(|V|)$ amortized

Adjacency List Representation (Classical)

- Maintain a list of all vertices.
- Have each vertex maintain a list of all its neighbors.

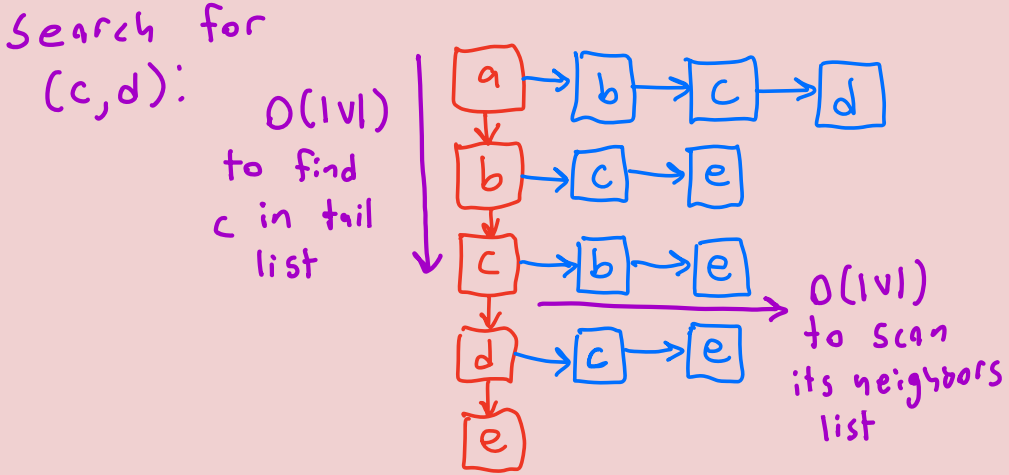




Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Given an adjacency list representation of a graph backed by nested `LinkedLists`, what is the tightest runtime complexity we can obtain for `hasEdge()`?



$O(\|V\|)$	**(A)**
$O(\|E\|)$	**(B)**
$O(\|V\| + \|E\|)$	**(C)**
$O(\|V\|^2)$	**(D)**

Adjacency List Representation (Improved)

Iterating over nested list structure is slow

- linear scan over vertices to find tail's neighbors list
- linear scan over neighbor list to search for edge

To improve efficiency: Replace both layers of lists with hash maps keyed on vertex labels

O(|V|) scan → expected O(1) lookup

we'll pick up here next lecture... rest of the slides are FYI

AdjListGraph State

```
public class AdjListGraph implements Graph<ALVertex, Edge<ALVertex>> {
  /** A vertex in this graph responsible for tracking its neighbors. */
  public static class ALVertex implements Vertex<Edge<ALVertex>> {
    /** The label of this vertex */
    private final String label;

    /** Map associating neighbor's labels with Edges connecting to them. */
    private final HashMap<String, Edge<ALVertex>> outEdges;
  }

  /** Map associating labels with ALVertex objects. */
  private final HashMap<String, ALVertex> vertices;
}
```

↖ vertex manages its neighborhood

graph simply collects vertices

General usage pattern

① Use vertices map to locate vertex O(1)

② Query for edges through tail vertex O(1)

Adjacency List Operation Complexities

```
class AdjListGraph implements Graph<...> {
  static class ALVertex implements Vertex<...> {
    private String label;
    private HashMap<String, Edge<...>> outEdges;

    // methods
  }

  private HashMap<String, ALVertex> vertices;

  // methods
}
```

Count vertices/neighbors:

O(1) using Map size()

Iterate over neighbors:

O(degree) using KeySet()

Check for an edge:

O(1) using Map get()

Count edges:

O(|V|) using outEdges.size()

Iterate over edges:

O(|V| + |E|)

Add edge/vertex:

O(1) using Map put()

Comparing Representations

Representation	Memory	Time to Query Edge	Time to Iterate Neighbors	Time to Iterate all Edges
Adj Matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$	$O(\|V\|^2)$
Adj List (Linked)	$O(\|V\| + \|E\|)$	$O(\|V\|)$	$O(\|V\|)$	$O(\|V\| + \|E\|)$
Adj List (Map)	$O(\|V\| + \|E\|)$	$O(1)$	$O(\text{degree})$	$O(\|V\| + \|E\|)$

Dense Graphs: have $|E| \approx O(|V|^2)$: AM + AL have similar performance
memory layout likely makes AM ops faster

Sparse Graphs: have $|E| \approx O(|V|)$: AL has much better performance
↖ most real-world graphs