Graph Implementations 2

CS 225 - Fall 2025

Mattox Beckman / Based on slides from Brad Solomon

Objectives

Objectives

- Implement a graph using an Adjacency List
- Compare this to the Edge List and Adjacency Matrix

Graph ADT

Modification

- `insertVertex(K key)`
- `insertEdge(Vertex v1, Vertex v2, K Key)`
- `removeVertex(K key)`
- `removeEdge(Vertex v1, Vertex v2)`

Need storage for vertices and keys. Maybe weights.

Graph ADT

Query

- `getEdges(Vertex v)`
- `areAdjecent(Vertex v1,Vertex v2)`
- `origin(Edge e)`
- `destination(Edge e)`

Simplistic Idea Näive Adjacency List

We will keep a **hashmap** with **vertex** as the key.

Simplistic Idea Näive Adjacency List

We will keep a **hashmap** with **vertex** as the key.

Each entry will have:

- The **degree** of the vertex
- A pointer to an **adjacency list**

Simplistic Idea Näive Adjacency List

We will keep a **hashmap** with **vertex** as the key.

Each entry will have:

- The **degree** of the vertex
- A pointer to an **adjacency list**

The adjacency list nodes will have:

- The adjacent vertex
- The weight
- Pointers to next and previous edge

Simplistic Idea Näive Adjacency List

We will keep a **hashmap** with **vertex** as the key.

Each entry will have:

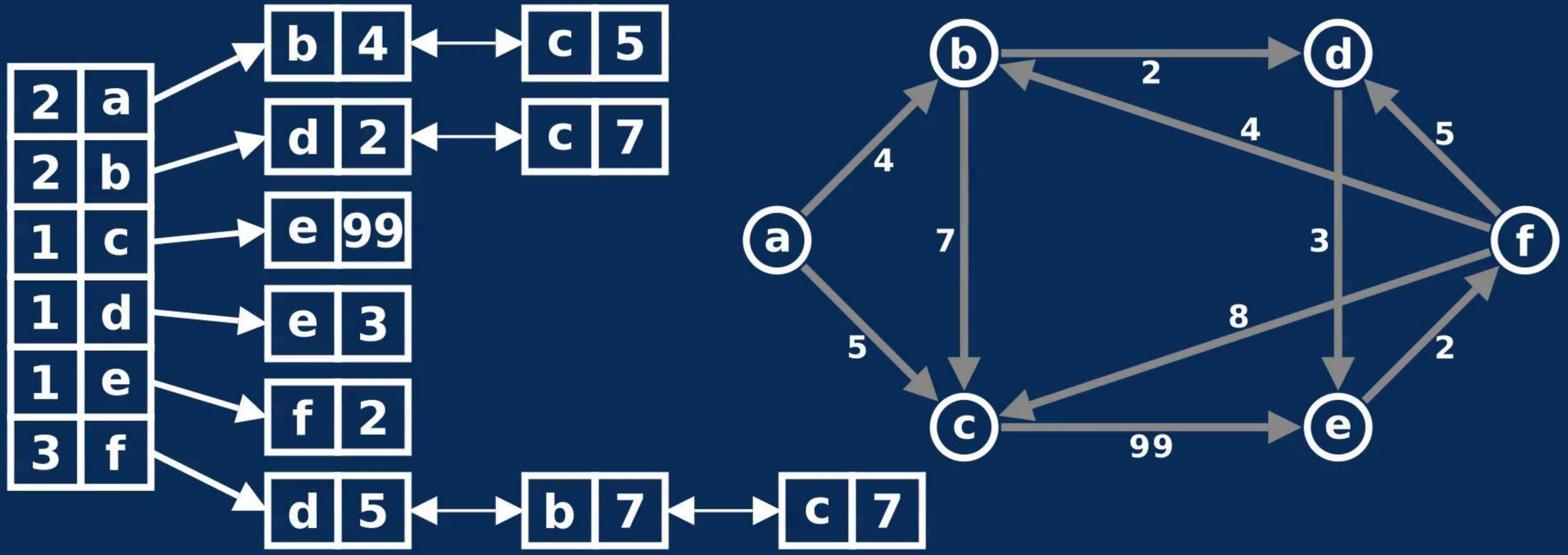
- The **degree** of the vertex
- A pointer to an **adjacency list**

The adjacency list nodes will have:

- The adjacent vertex
- The weight
- Pointers to next and previous edge

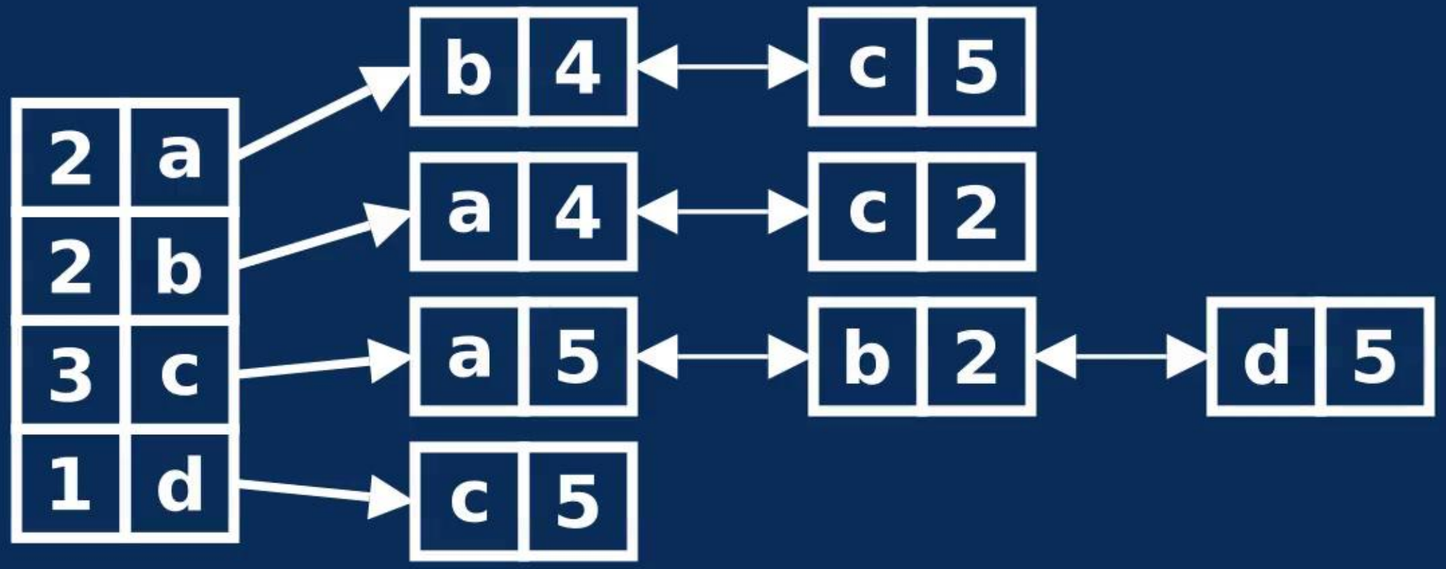
We will **NOT** show the pointers explicitly.

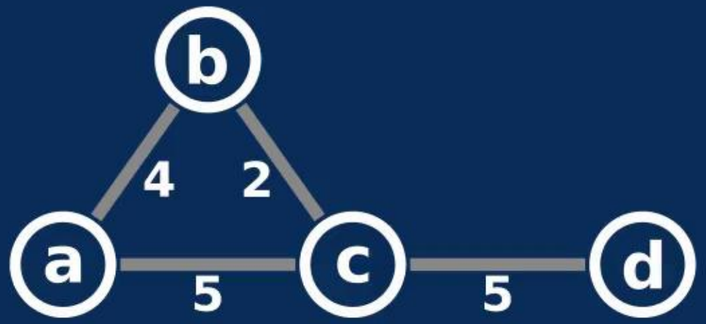
The Idea



Degree/Vertex pointing to adjacency list.

Undirected Näive Adjacency List





Insertion Operations

`InsertVertex(Vertex v, Key k)`

Add vertex to hash map:

- Degree zero
- Null list pointer

Insertion Operations

`InsertVertex(Vertex v, Key k)`

Add vertex to hash map:

- Degree zero
- Null list pointer

`InsertEdge(Vertex v1, Vertex v2, Key w)`

- For both `v1` and `v2`:
  - Insert list node for the edge

Insertion Operations

`InsertVertex(Vertex v, Key k)`

Add vertex to hash map:

- Degree zero
- Null list pointer

`InsertEdge(Vertex v1, Vertex v2, Key w)`

- For both `v1` and `v2`:
  - Insert list node for the edge

How long should these take?

Insertion Operations

`InsertVertex(Vertex v, Key k)`

Add vertex to hash map:

- Degree zero
- Null list pointer

Time: $\mathcal{O}(1)$

`InsertEdge(Vertex v1, Vertex v2, Key w)`

- For both `v1` and `v2`:
  - Insert list node for the edge

Time: $\mathcal{O}(1)$

How long should these take?

Insertion Operations

`InsertVertex(Vertex v, Key k)`

Add vertex to hash map:

- Degree zero
- Null list pointer

Time: $\mathcal{O}(1)$

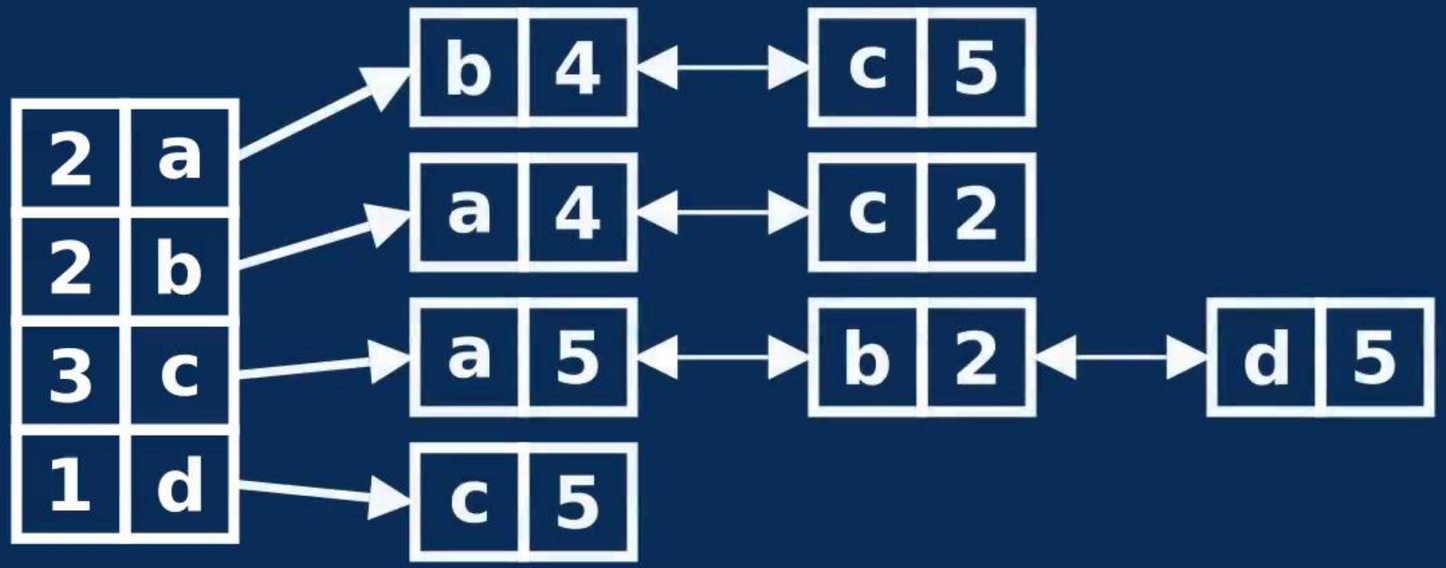
`InsertEdge(Vertex v1, Vertex v2, Key w)`

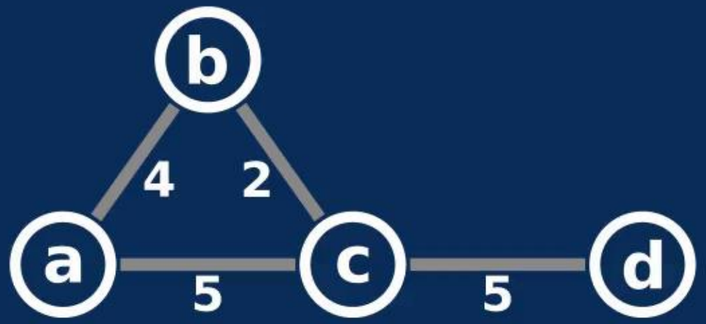
- For both `v1` and `v2`:
  - Insert list node for the edge

Time: $\mathcal{O}(1)$

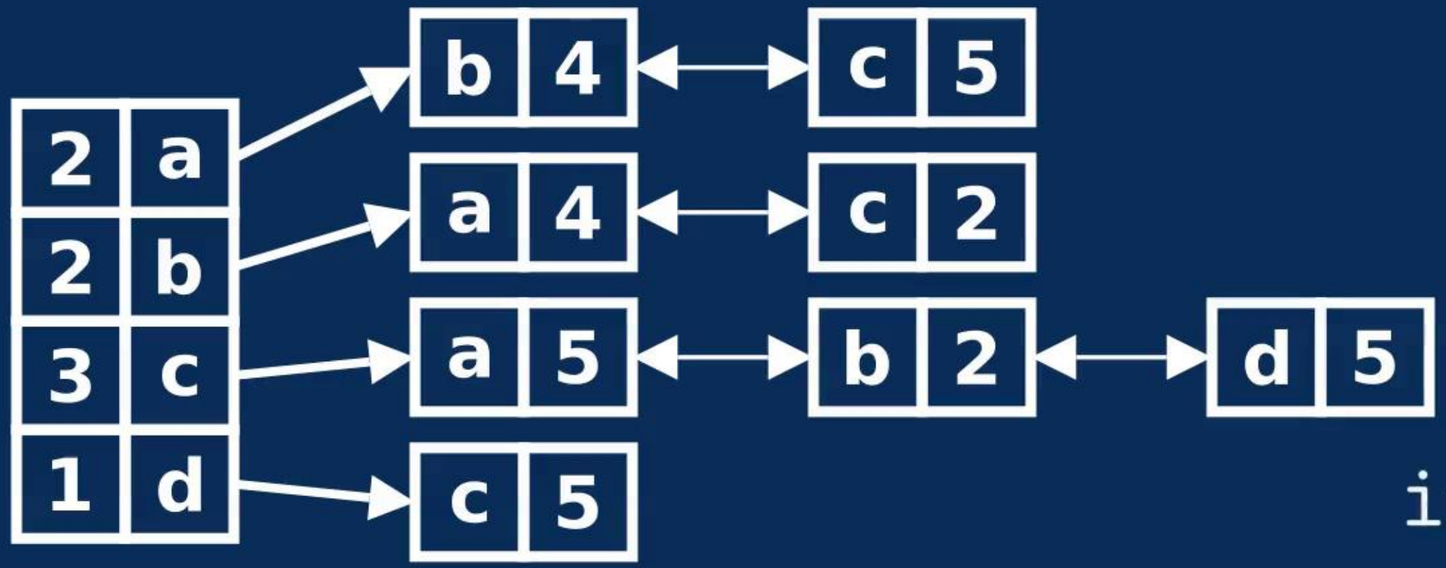
How long should these take?

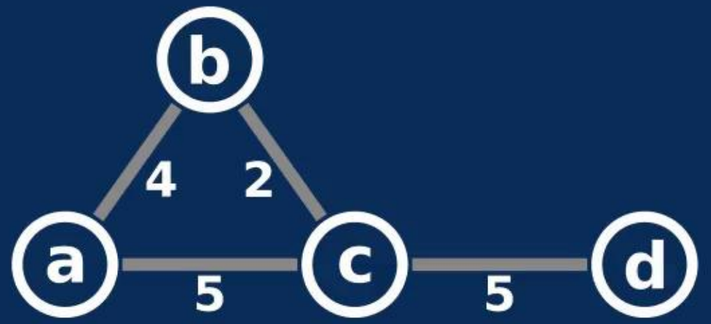
Insert Vertex and Edge





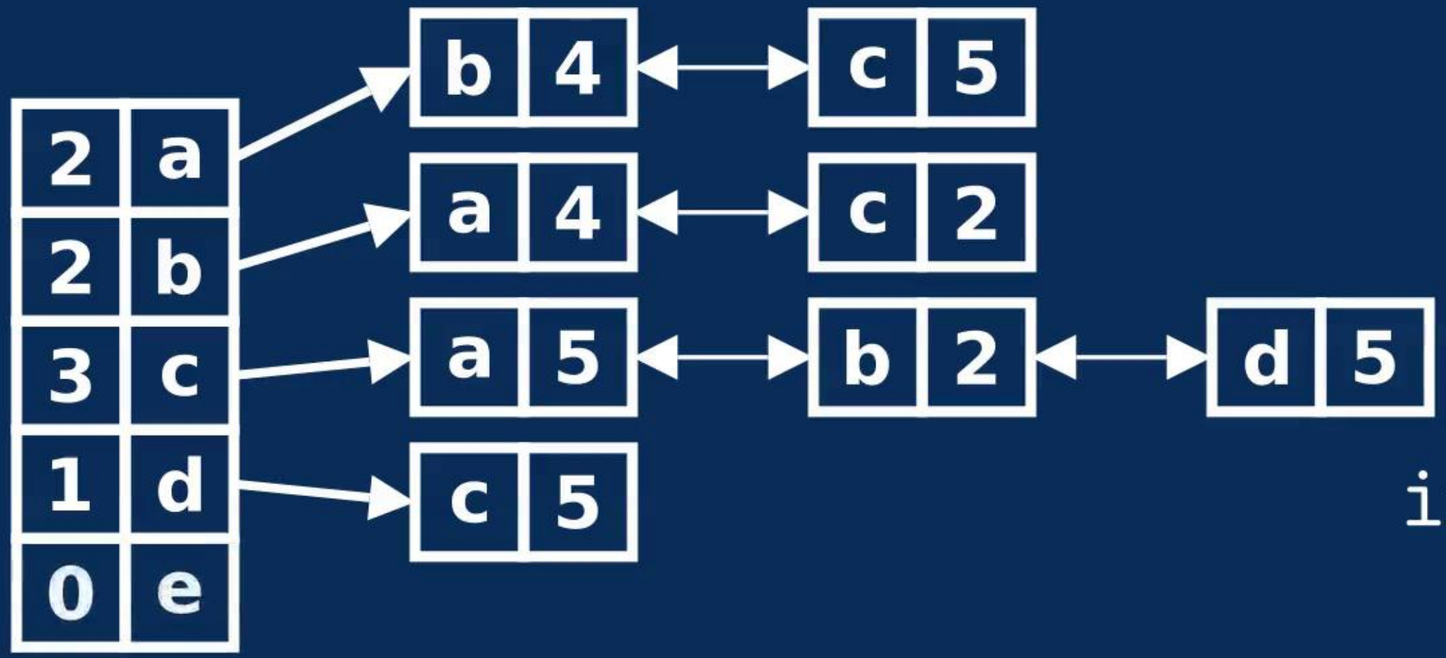
Insert Vertex and Edge

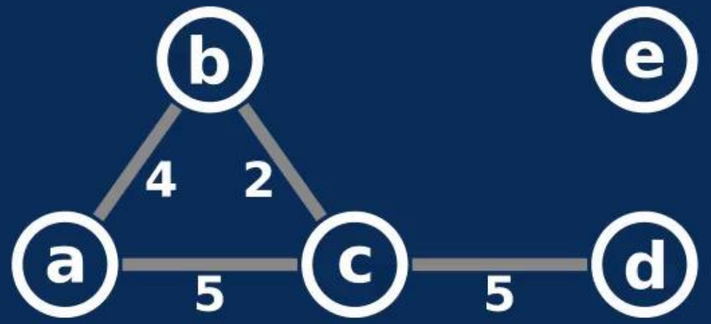





```
insertVertex("e")
```

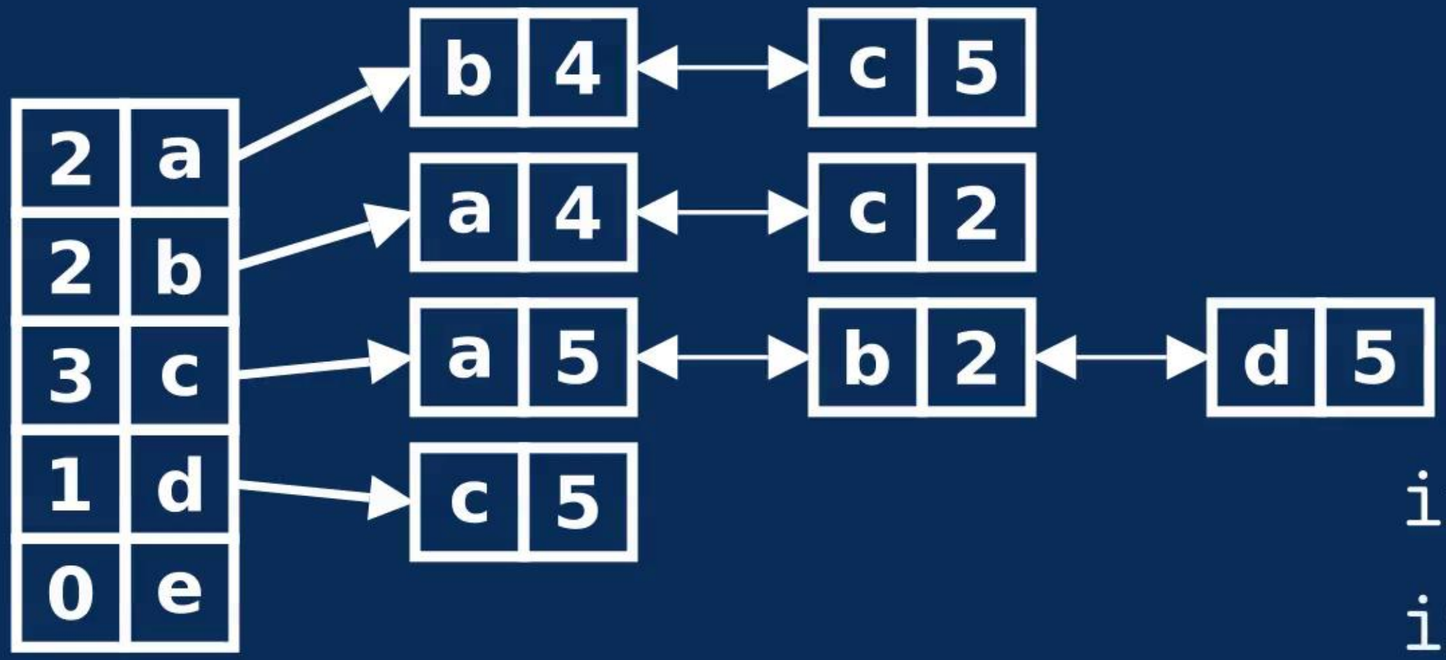
Insert Vertex and Edge

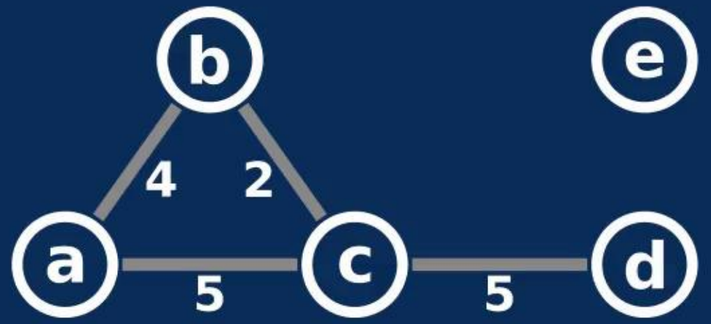





```
insertVertex("e")
```

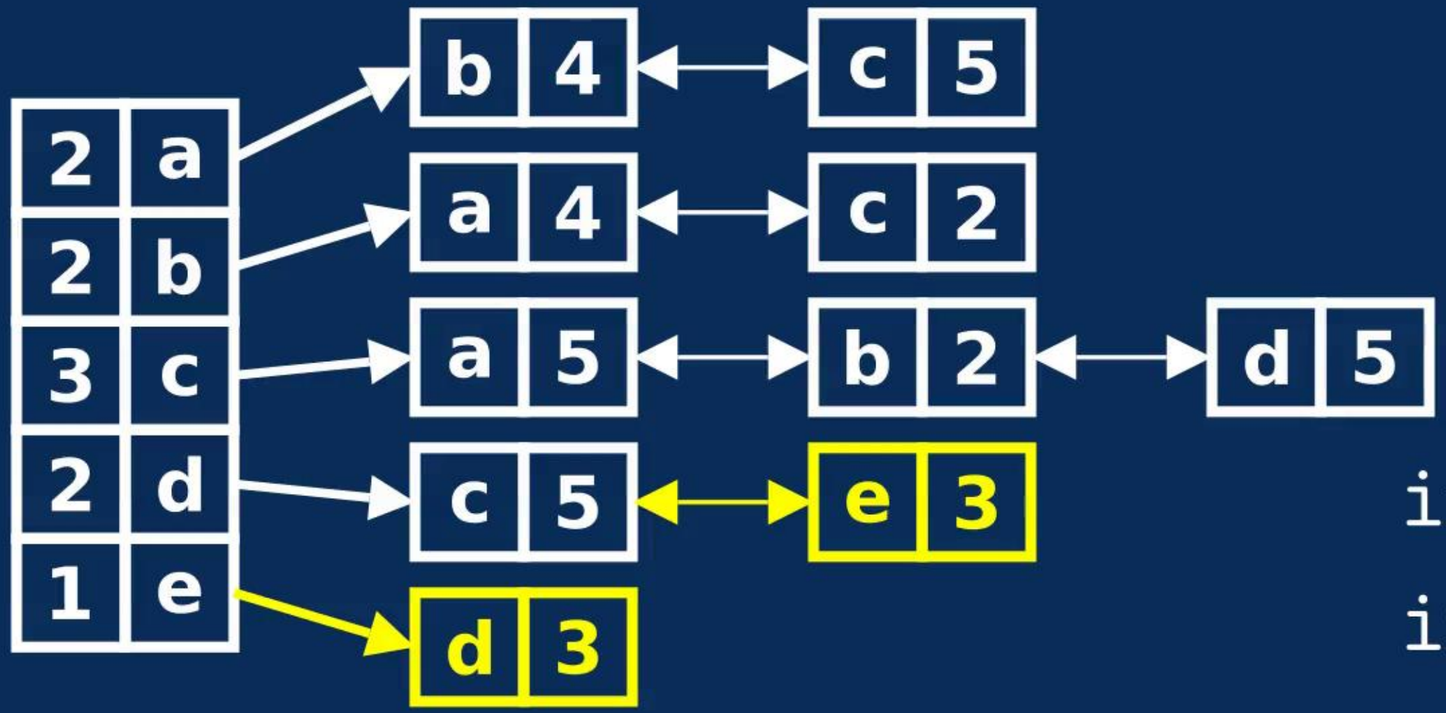
Insert Vertex and Edge

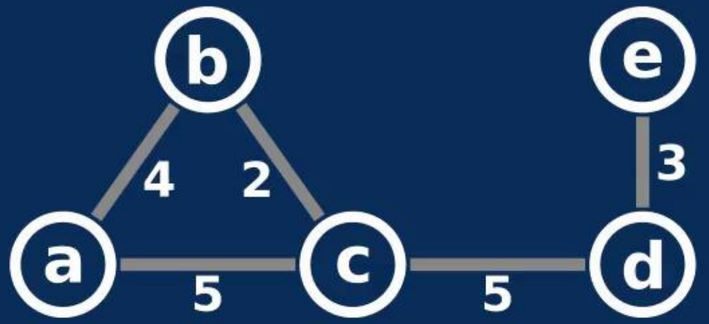





```
insertVertex("e")
inesrtEdge("d","e",3)
```

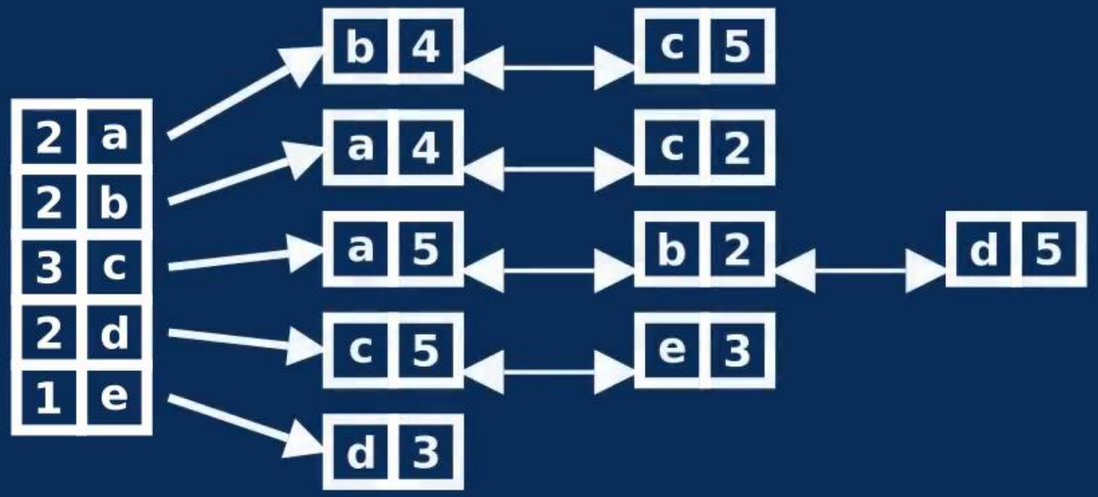
Insert Vertex and Edge

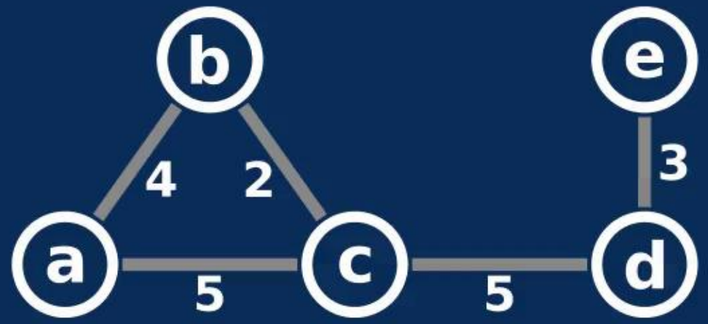




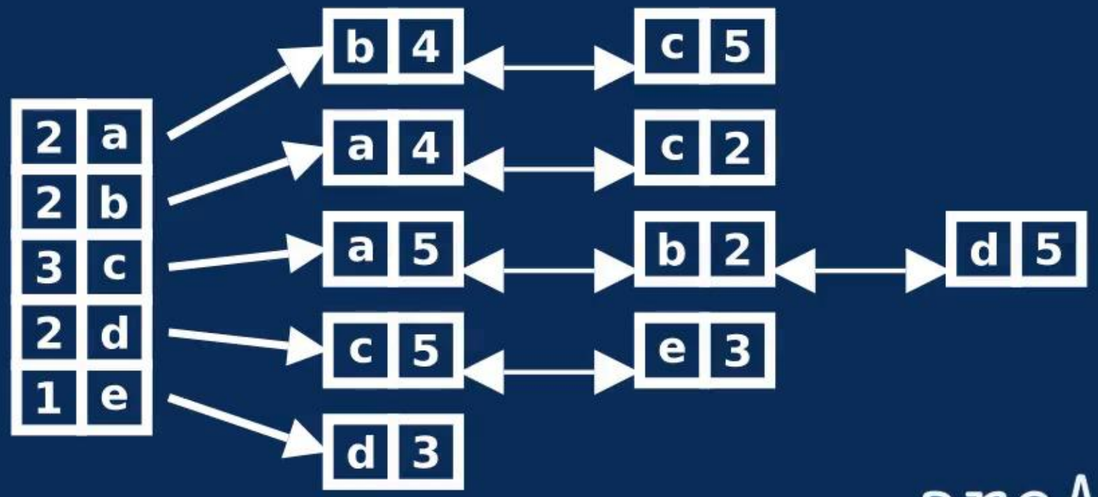

```
insertVertex("e")
inesrtEdge("d","e",3)
```

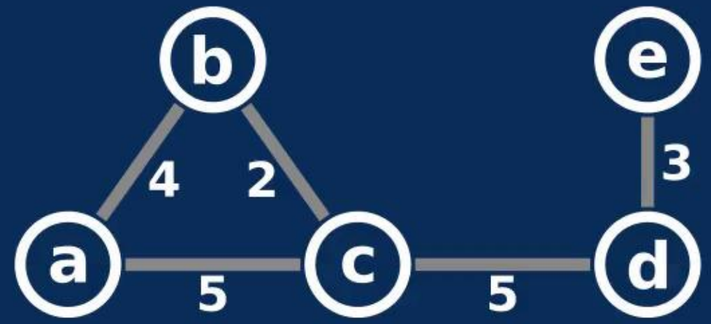
Queries





Queries





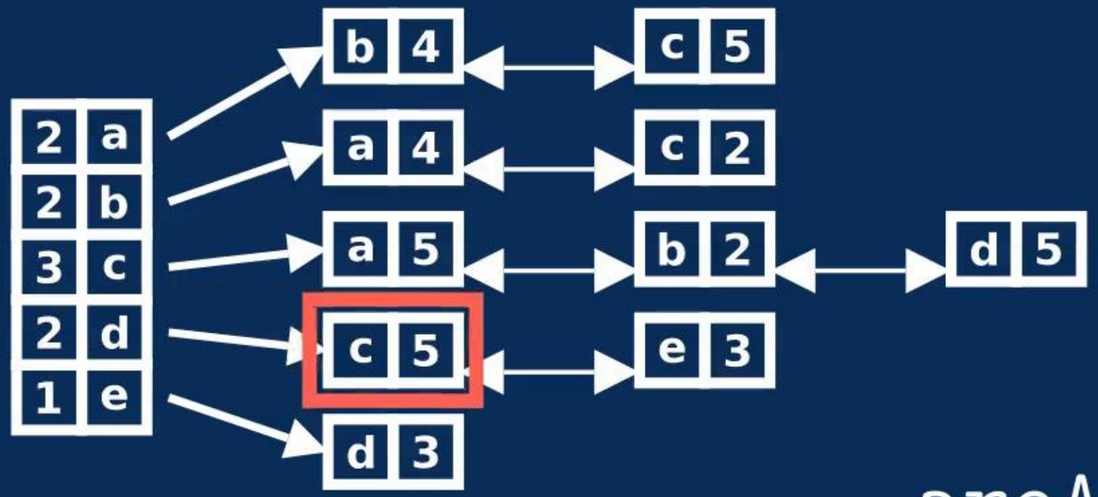
`areAdjacent(v1,v2)`

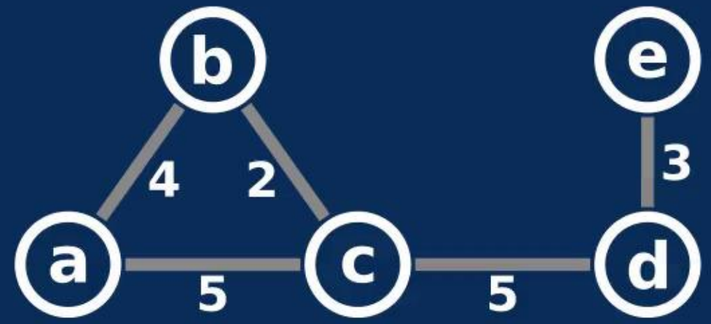
Check vertex with min degree

Traverse adjacency list looking for other

Eg `isAdjacent('c','d')`

Queries





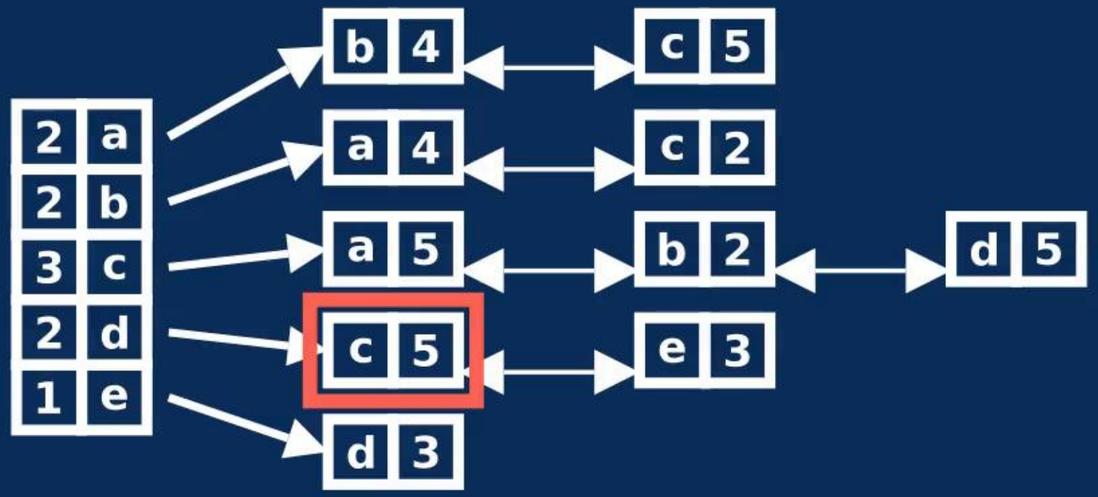
`areAdjacent(v1,v2)`

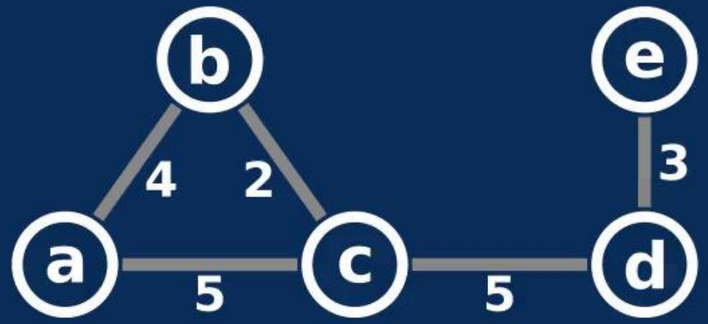
Check vertex with min degree

Traverse adjacency list looking for other

Eg `isAdjacent('c','d')`

Queries





`incedentEdges(v)`

Just walk across the adjacency list.

$\mathcal{O}(deg(v))$

Deletion

`deleteEdge(v1,v2)`

- Check `v1` and `v2` for the edge
- Remove it

Deletion

`deleteEdge(v1,v2)`

- Check `v1` and `v2` for the edge
- Remove it

`deleteVertex(v)`

- Traverse v's adjacency list and remove edges
- But you must remove corresponding edges from the other vertices!

Deletion

`deleteEdge(v1,v2)`

- Check `v1` and `v2` for the edge
- Remove it

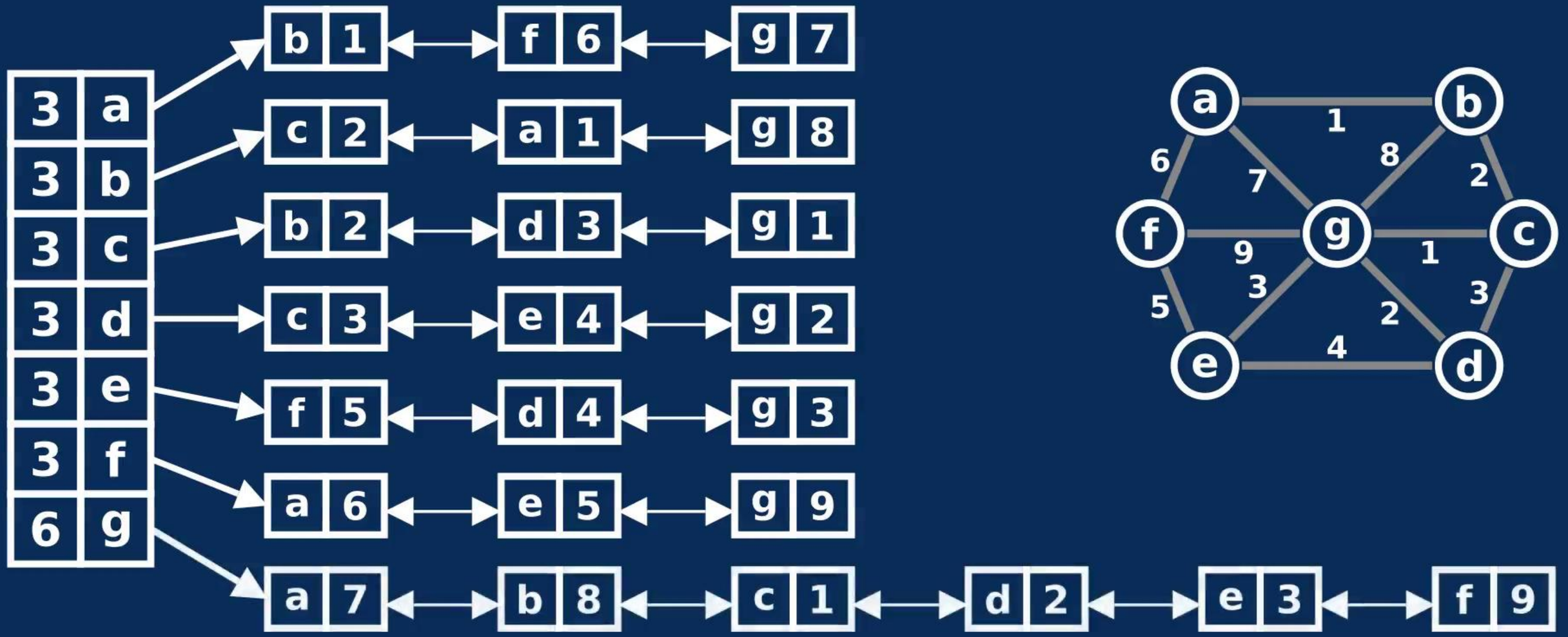
$\mathcal{O}(deg(v_1) + deg(v_2))$

`deleteVertex(v)`

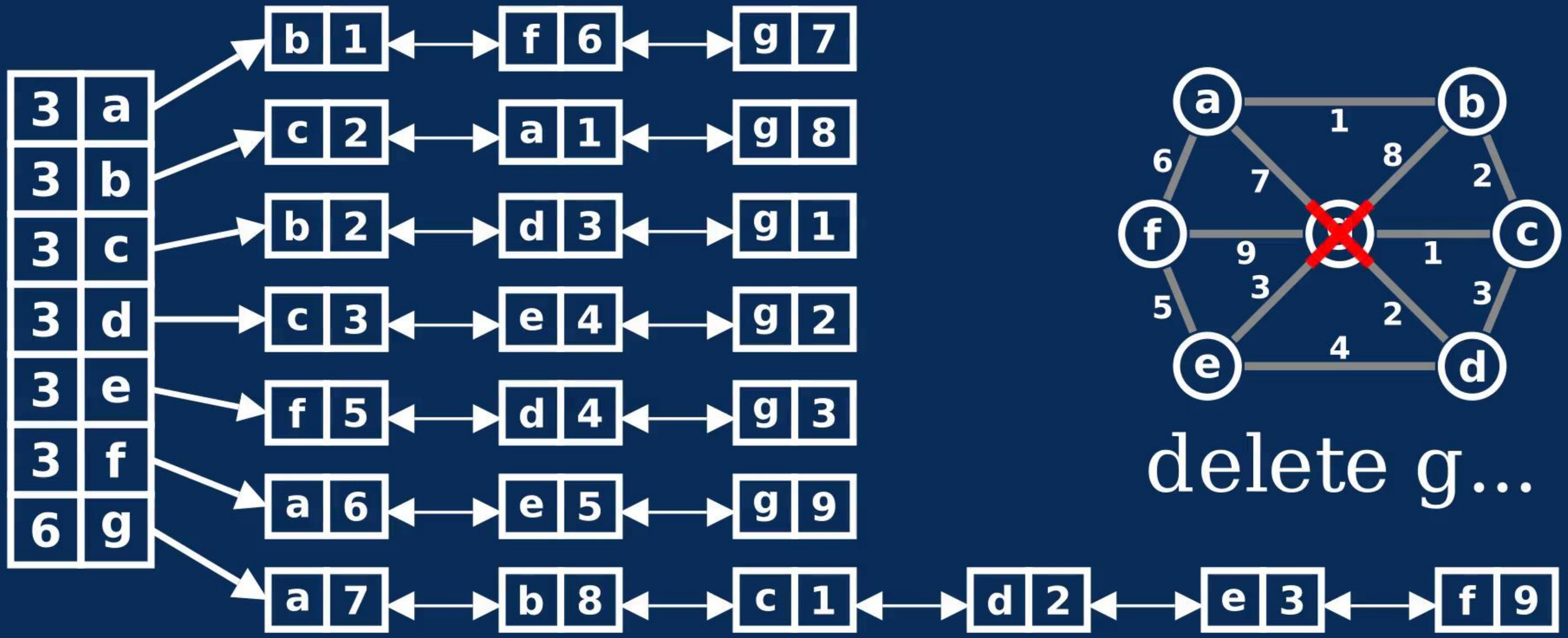
- Traverse v's adjacency list and remove edges
- But you must remove corresponding edges from the other vertices!

$\mathcal{O}(|E|)$. Can be $\mathcal{O}(|V|^2)$ in a dense graph!

Deletion Example

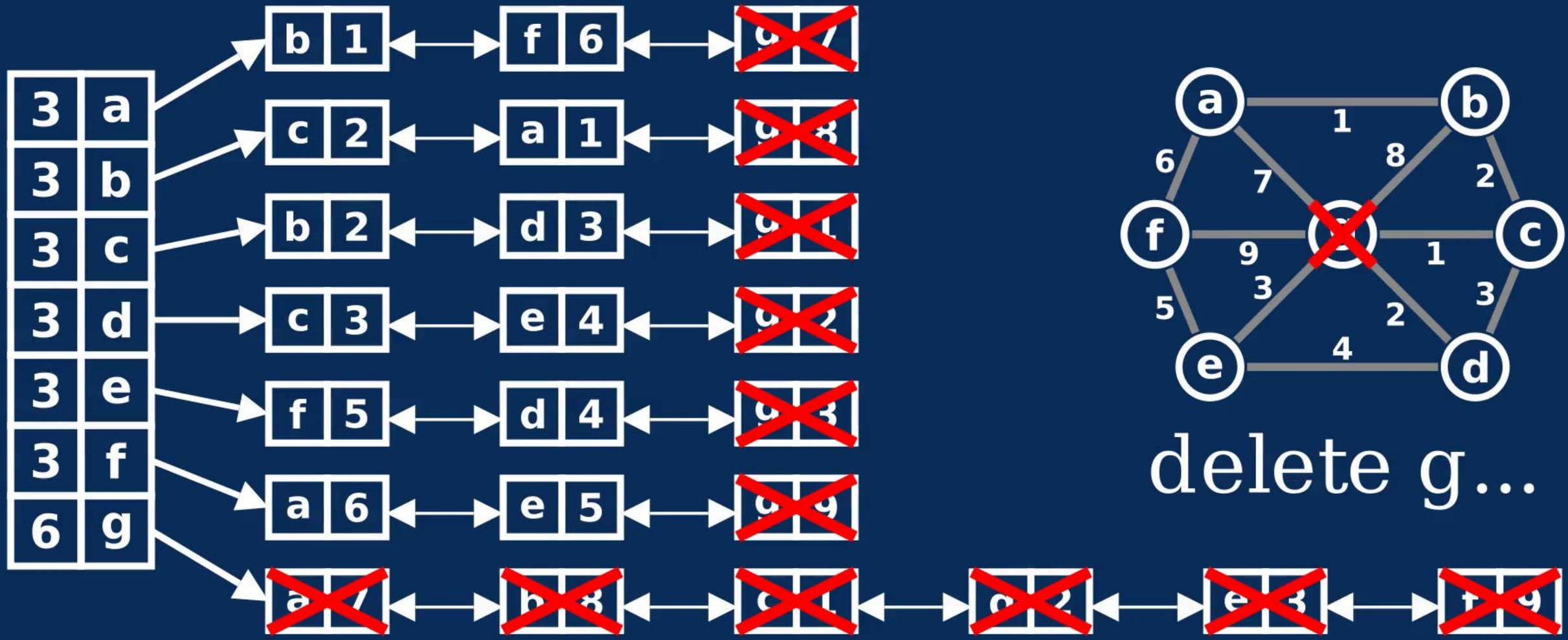


Deletion Example



delete g...

Deletion Example



delete g...

Fixing it!

Deleting edge (v_1, v_2)...

- Need to delete the adjacency list node linked from v_1
- And also delete the corresponding adjacency list node linked from v_2

Fixing it!

Deleting edge (v_1, v_2)...

- Need to delete the adjacency list node linked from v_1
- And also delete the corresponding adjacency list node linked from v_2

We need them to know each other's locations. Yay! More pointers!

Fixing it!

Deleting edge (v_1, v_2)...

- Need to delete the adjacency list node linked from v_1
- And also delete the corresponding adjacency list node linked from v_2

We need them to know each other's locations. Yay! More pointers!

Option 1: Link adjacency list nodes together.

Fixing it!

Deleting edge (v_1, v_2)...

- Need to delete the adjacency list node linked from v_1
- And also delete the corresponding adjacency list node linked from v_2

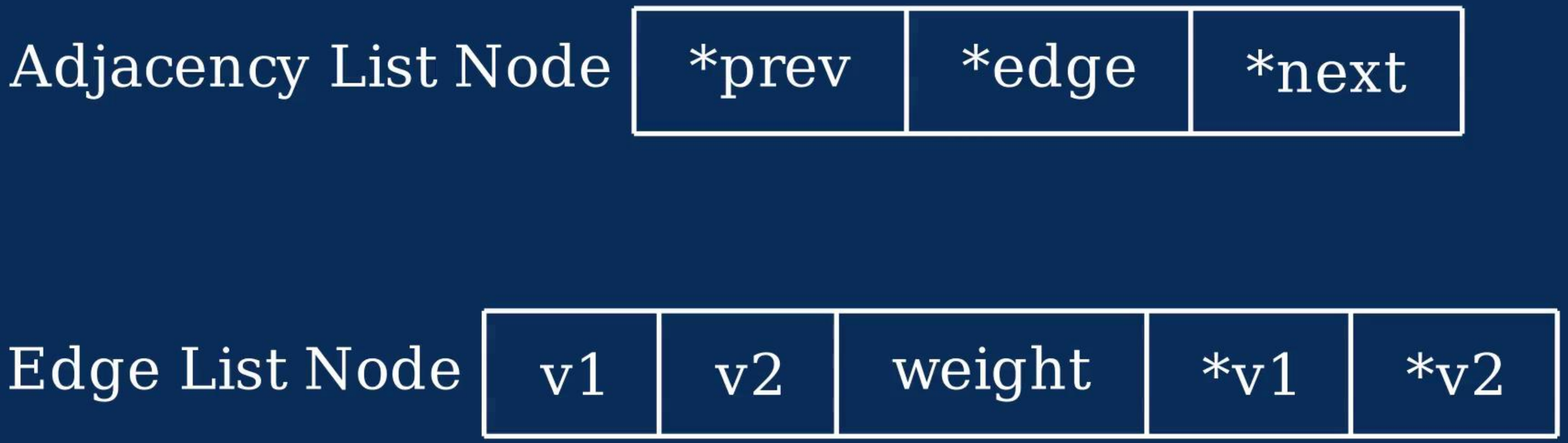
We need them to know each other's locations. Yay! More pointers!

Option 1: Link adjacency list nodes together.

Option 2: Keep a separate edge list and link to that!

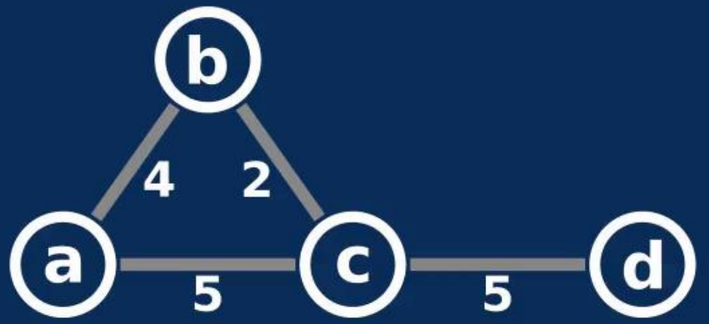
- Adjacency list nodes will link to an edge node
- Edge list nodes will link back to adjacency list node.
- Bonus: better storage and better runtime for edge-list algorithms.

Node Structure Good Adjacency List



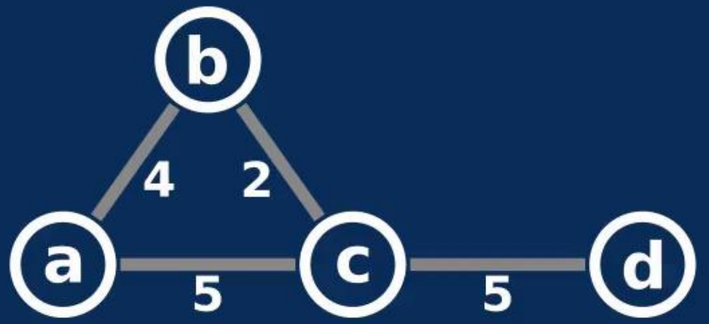
Example

0	a
0	b
0	c
0	d



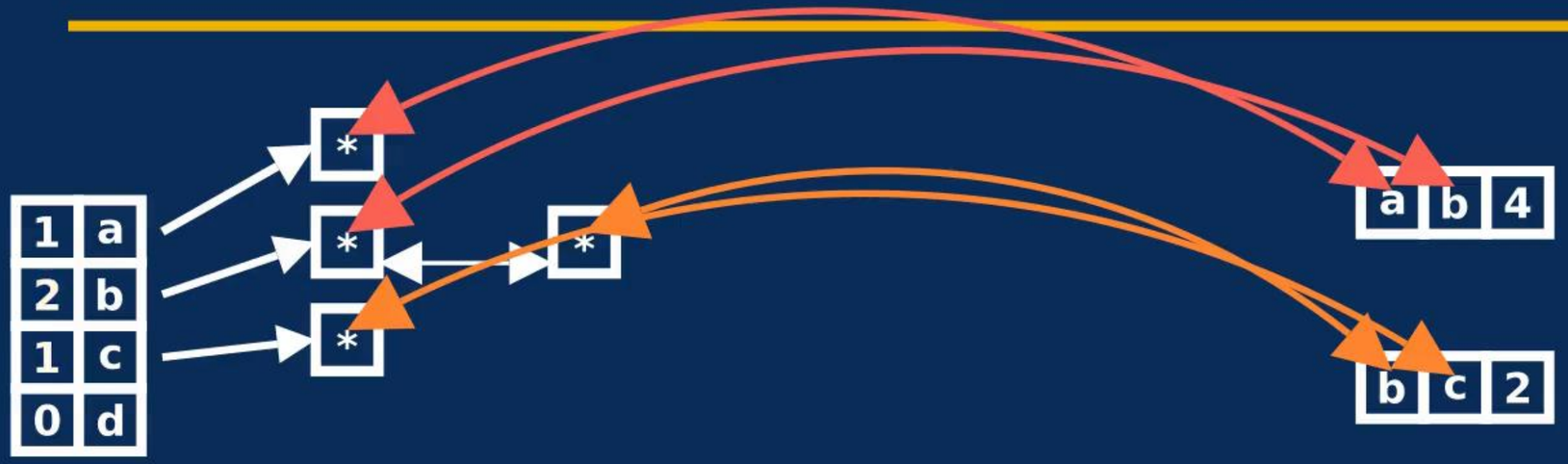
Let's create this graph. Assume we just created the vertices.

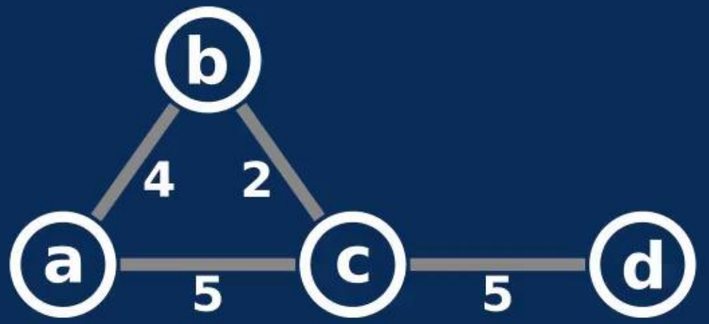
Example



Let's insert edge a-b now.

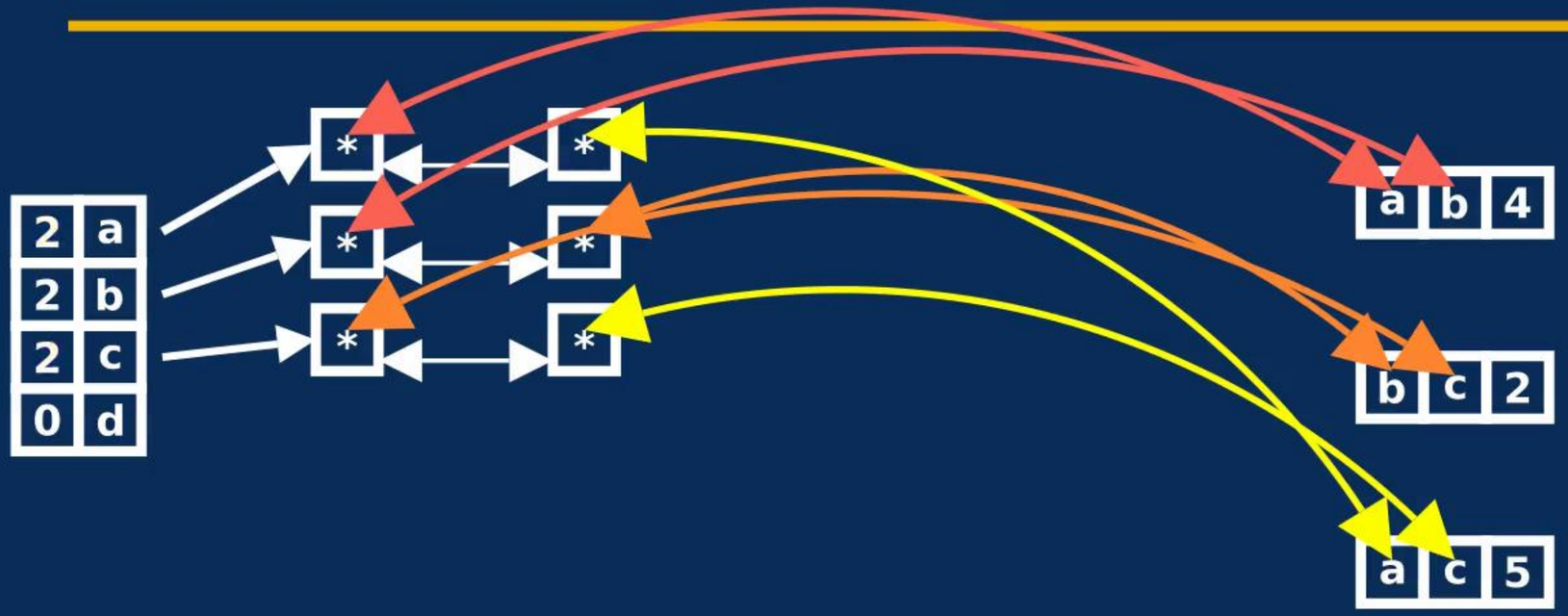
Example

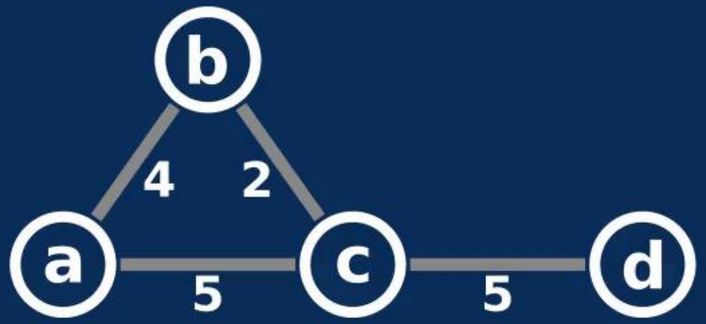




Insert edge b-c.

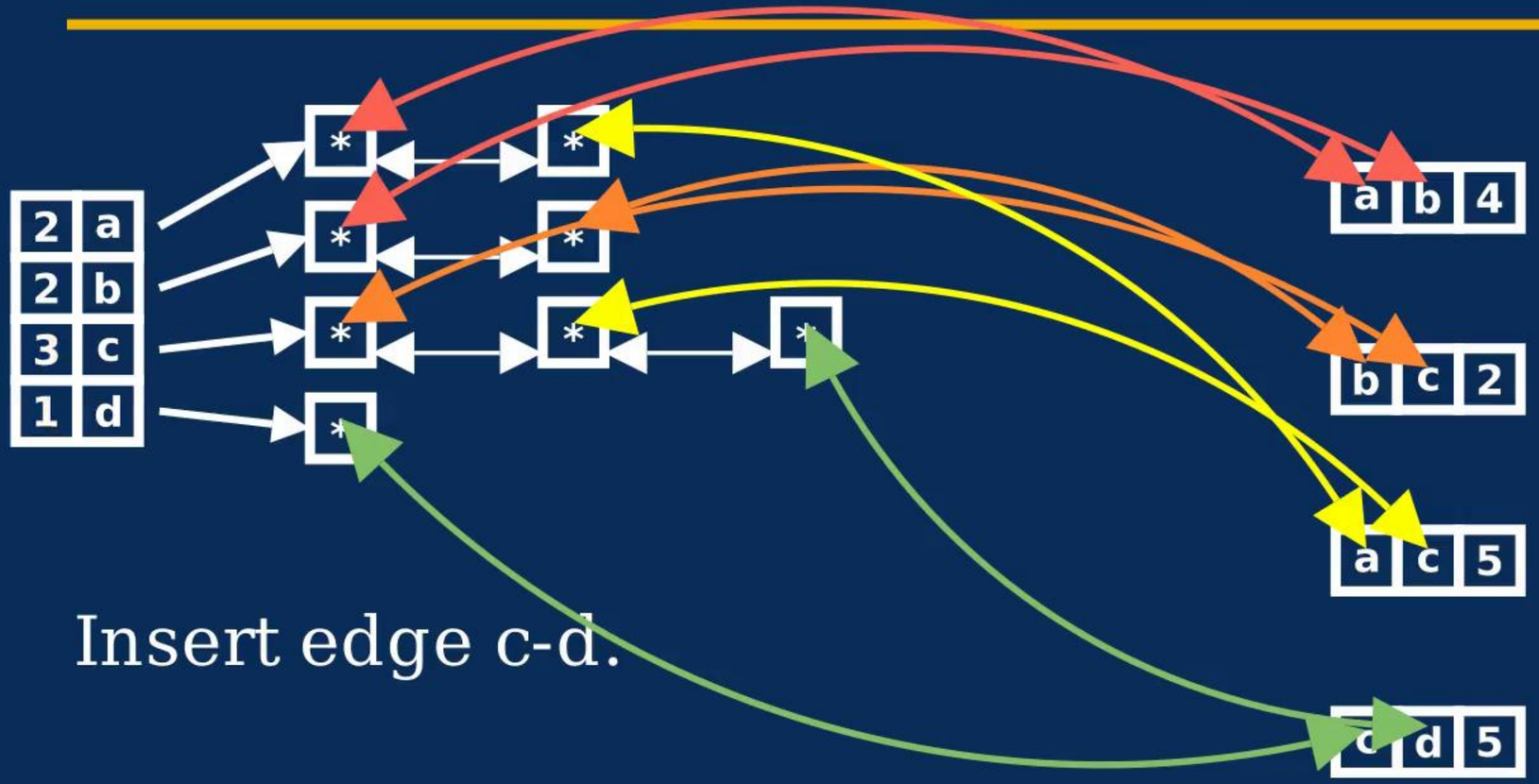
Example

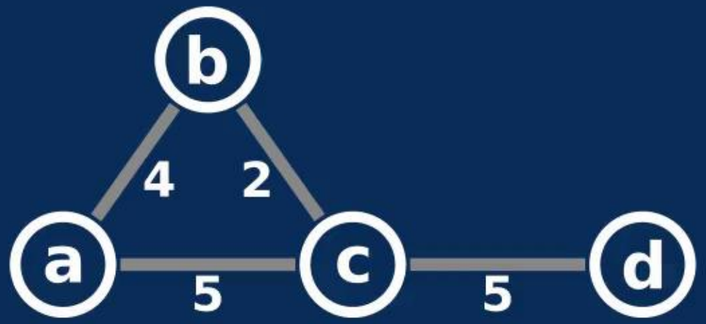




Insert edge a-c.

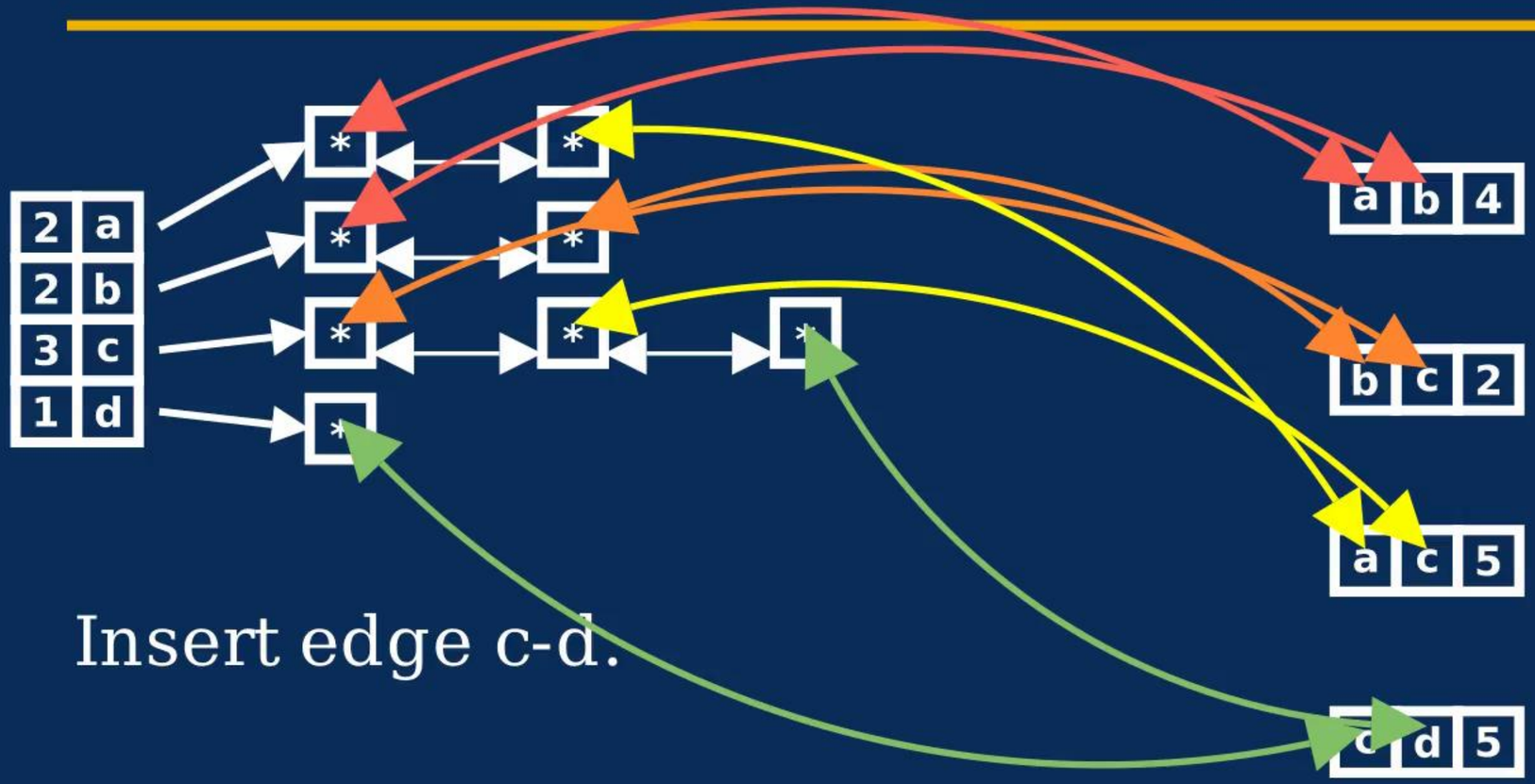
Example

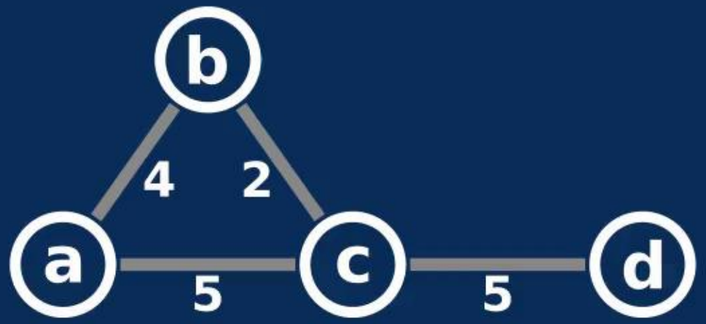




Insert edge c-d.

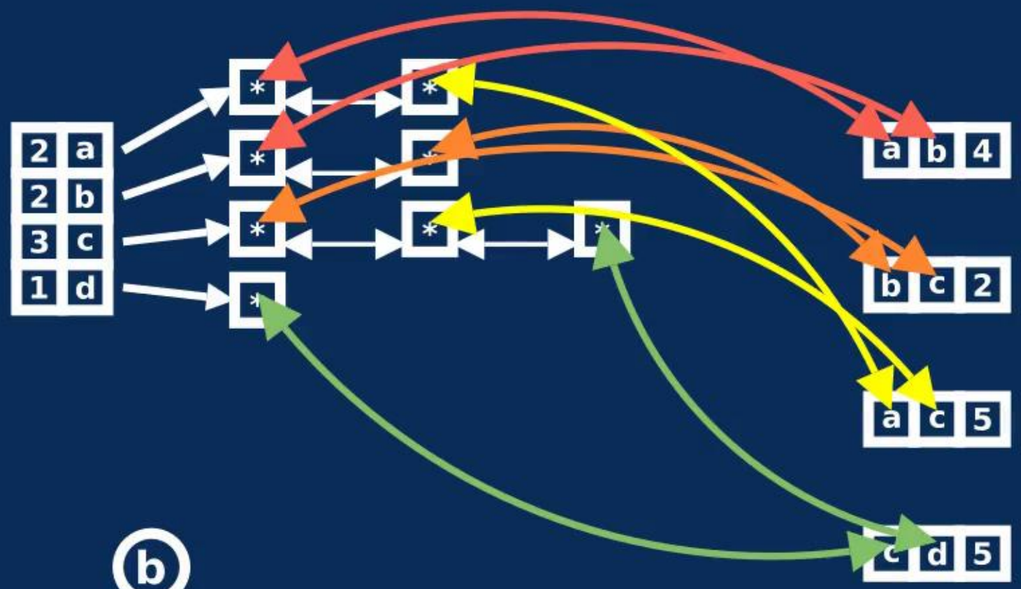
Example

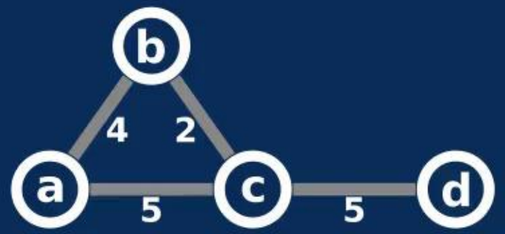




Insert edge c-d.

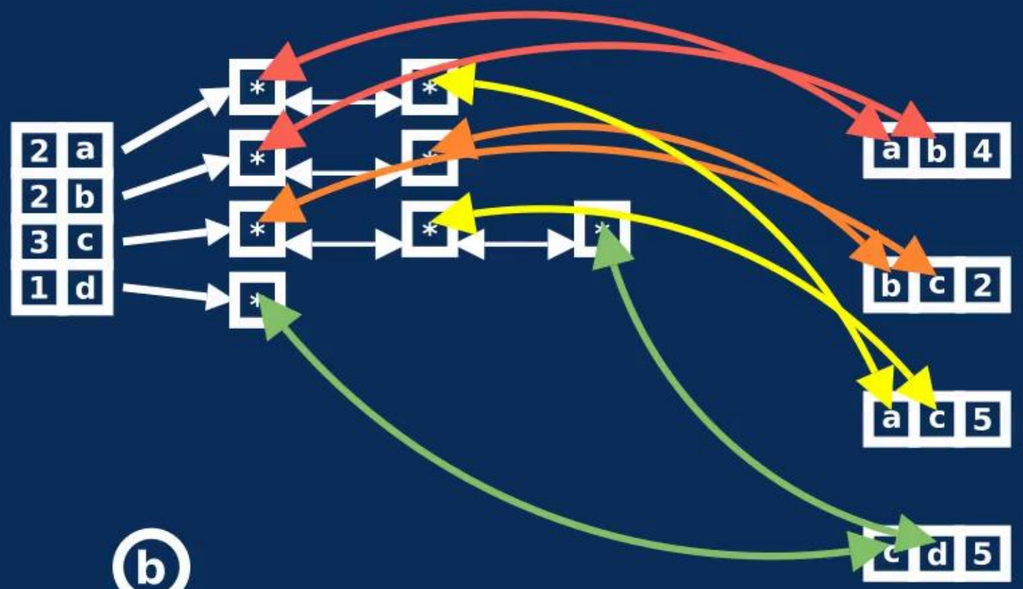
Timing

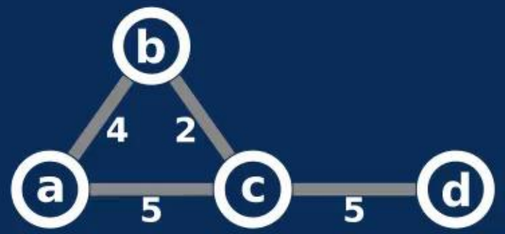




`incidentEdges(v1)`

Timing

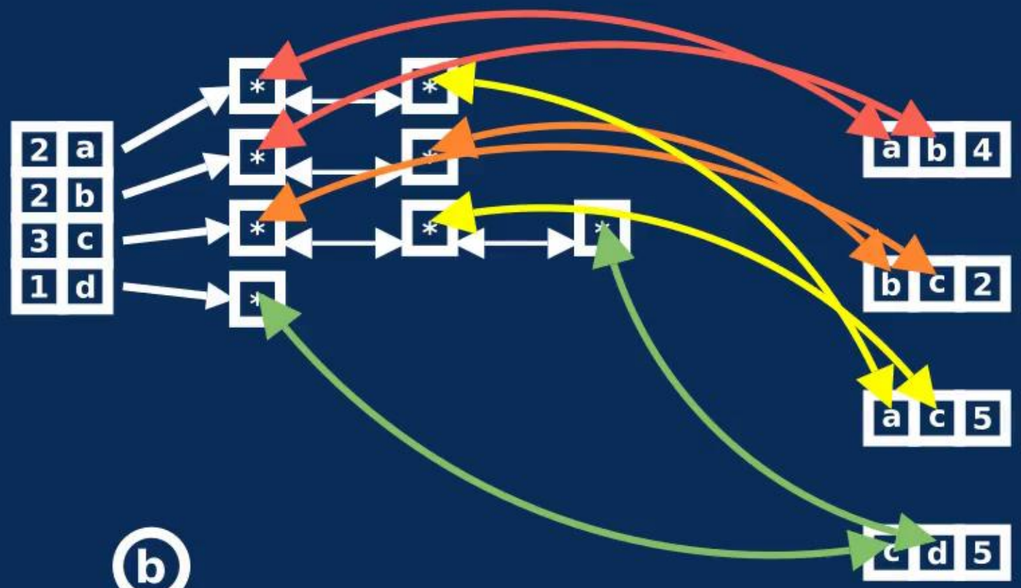


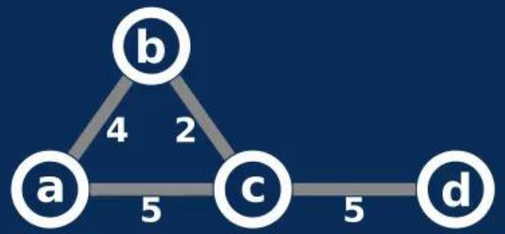


`incidentEdges(v1)`

- Look up `v1` and walk edges.
- Lookup should be $\mathcal{O}(1)$
- $\mathcal{O}(deg(v_1))$

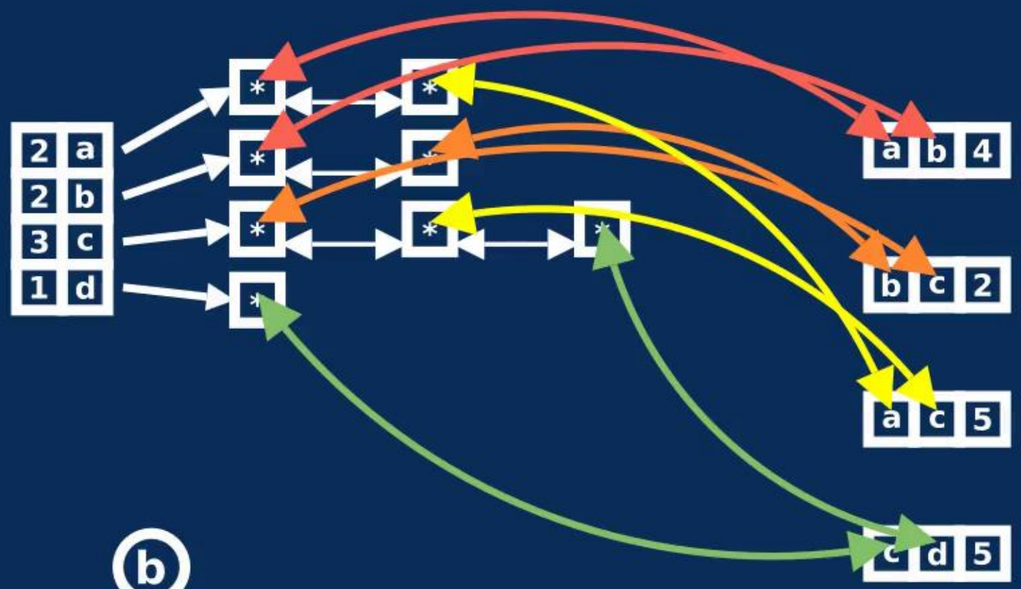
Timing

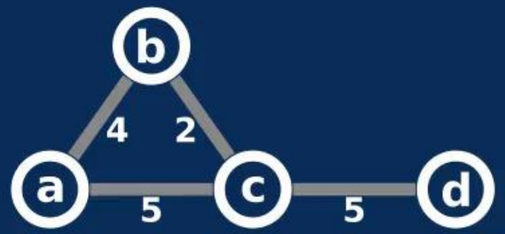





```
areAdjacent(v1,v2)
```

Timing

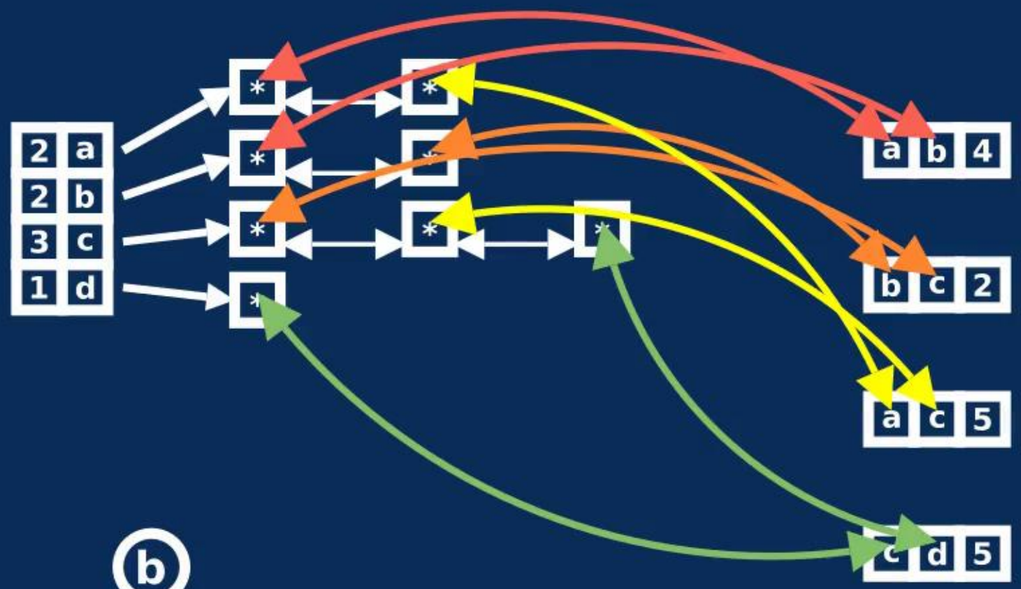


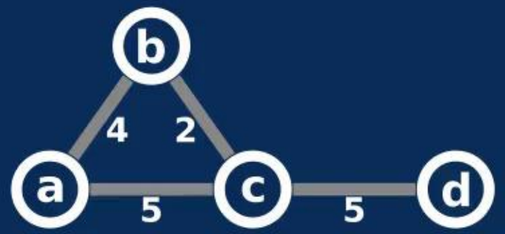


`areAdjacent(v1,v2)`

- Look up min degree `v1` and `v2`
- Assume WLOG `v1` is min
- Search `v1`'s list for `v2`
- $\mathcal{O}(deg(v_1))$

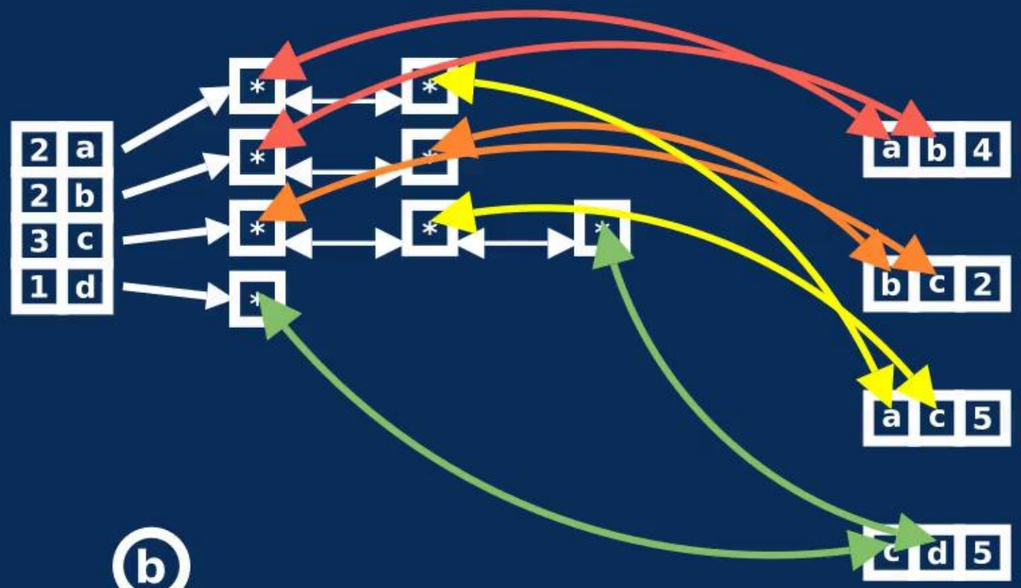
Timing

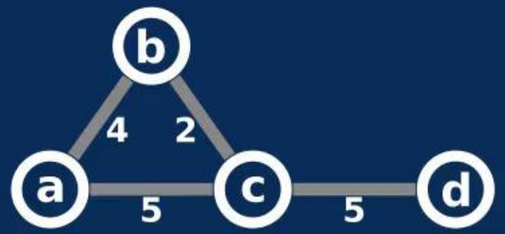





```
insertEdge(v1,v2,k)
```

Timing

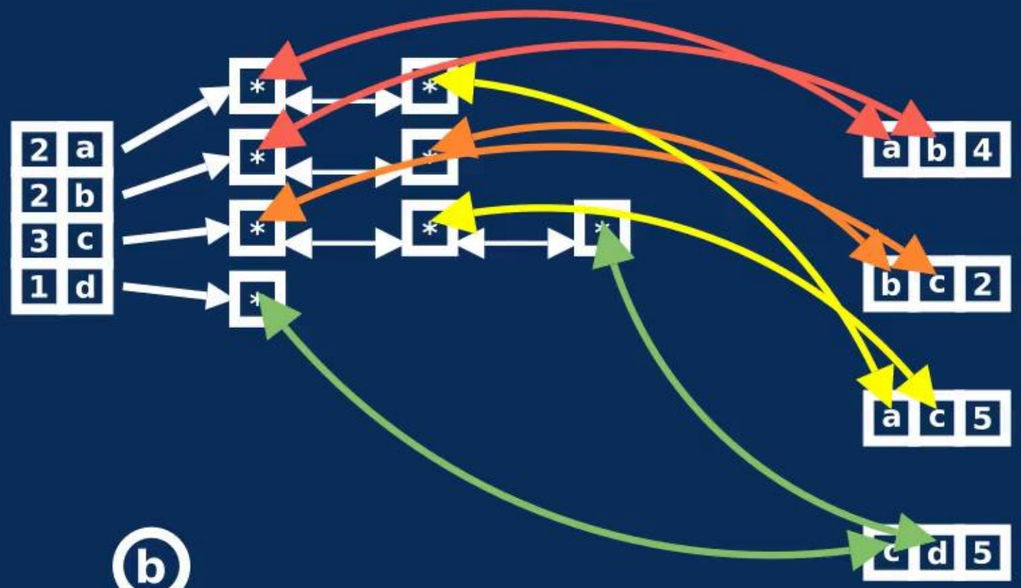


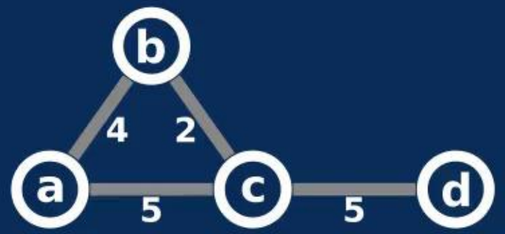


`insertEdge(v1,v2,k)`

- Add edge to edge list $= \mathcal{O}(1)$
- Add node to adj. lists $= \mathcal{O}(1)$
- Link pointers $= \mathcal{O}(1)$

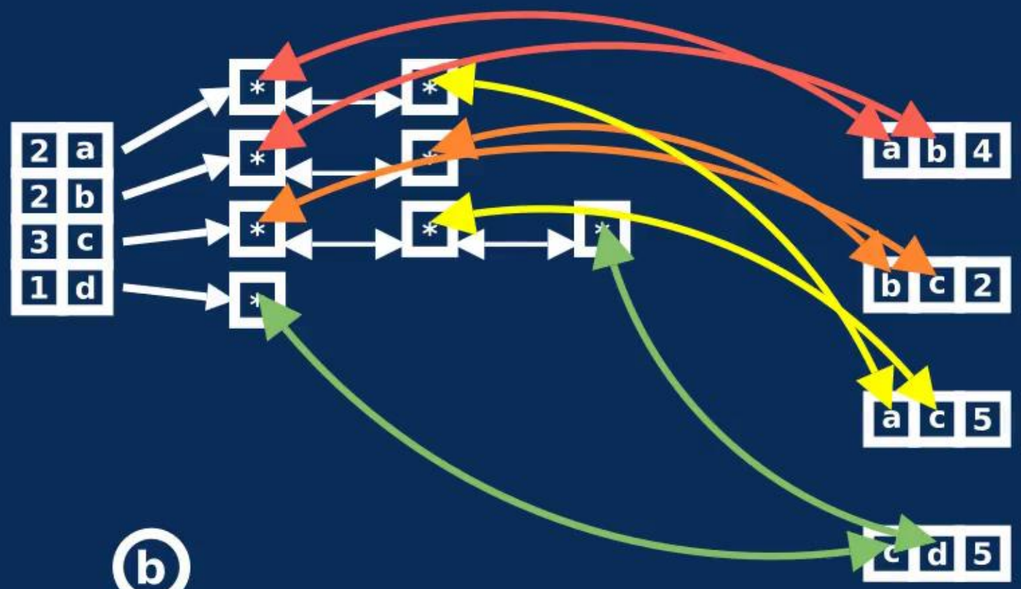
Timing

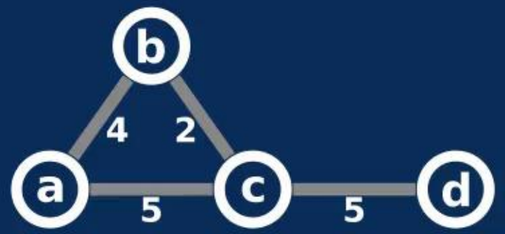




`removeEdge(v1,v2)`

Timing

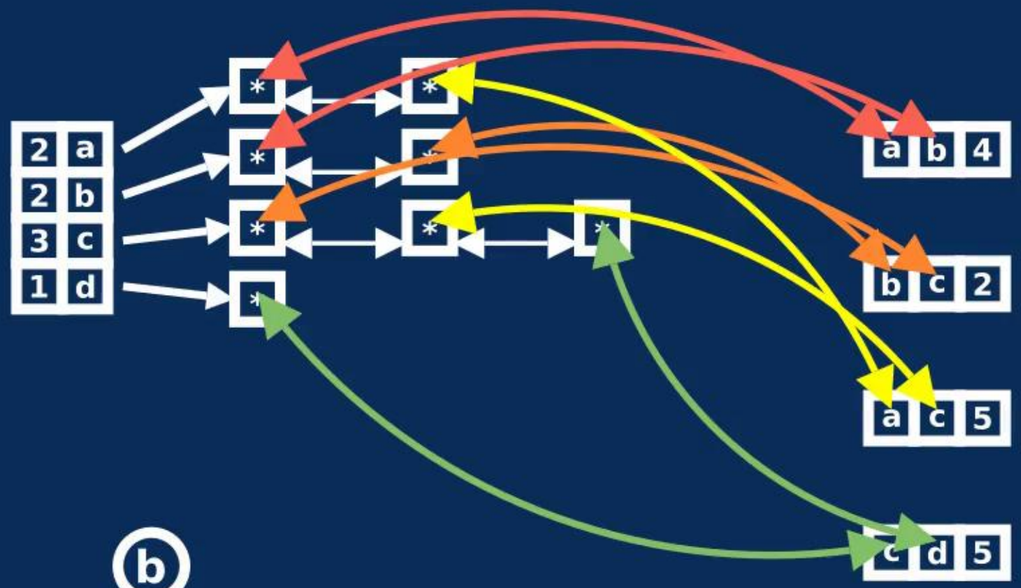


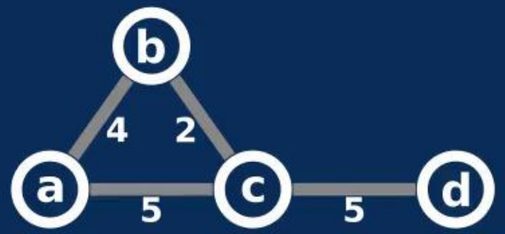


`removeEdge(v1,v2)`

- Get v1 and v2 = $\mathcal{O}(1)$
- Assume WLOG v1 is min deg
- Get edge: $\mathcal{O}(deg(v_1))$

Timing

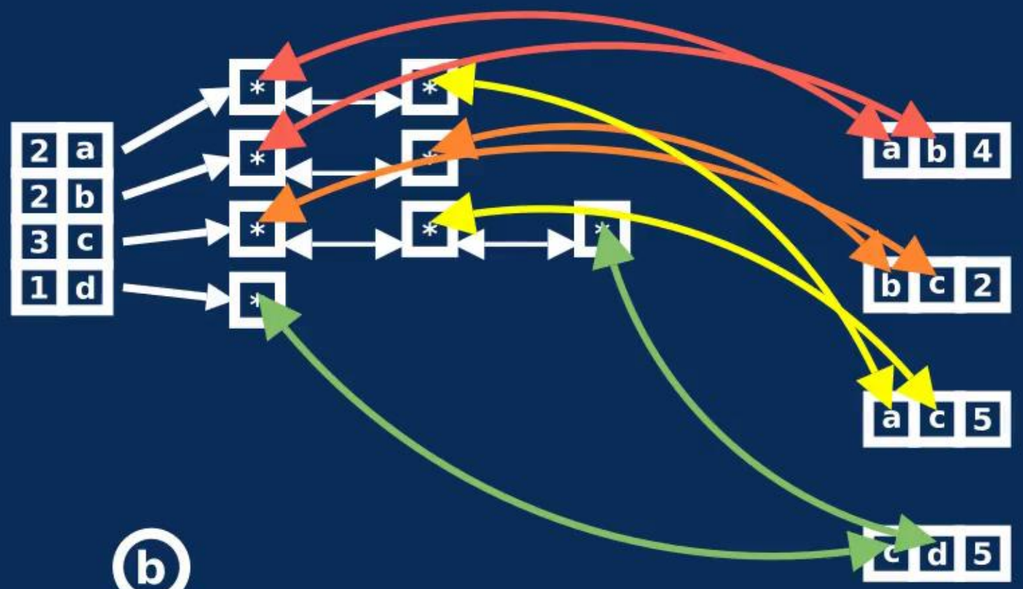


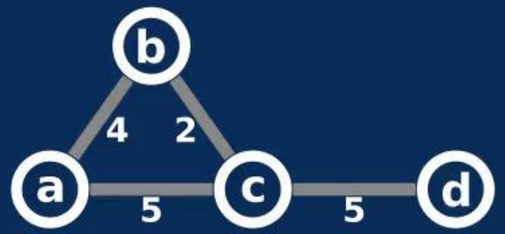


`removeEdge(v1,v2)`

- Get `v1` and `v2` = $\mathcal{O}(1)$
- Assume WLOG `v1` is min deg
- Get edge: $\mathcal{O}(deg(v_1))$
- Remove `v1`'s adj. node = $\mathcal{O}(1)$

Timing



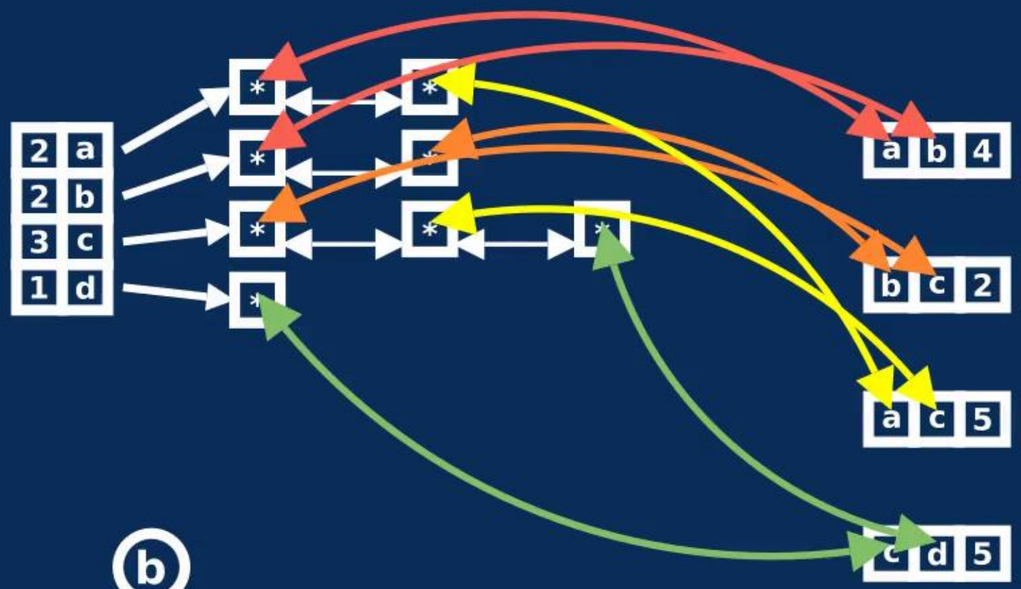


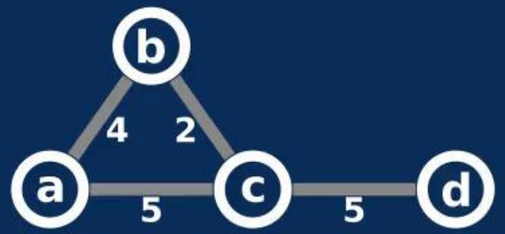
`removeEdge(v1,v2)`

- Get `v1` and `v2` = $\mathcal{O}(1)$
- Assume WLOG `v1` is min deg
- Get edge: $\mathcal{O}(deg(v_1))$

- Remove `v1`'s adj. node = $\mathcal{O}(1)$
- Remove edge from list = $\mathcal{O}(1)$

Timing



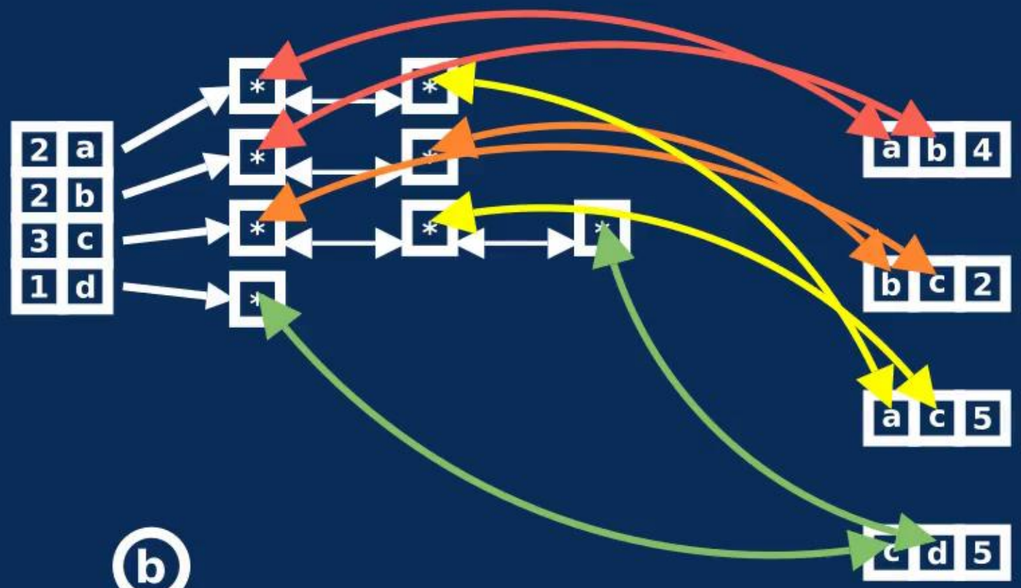


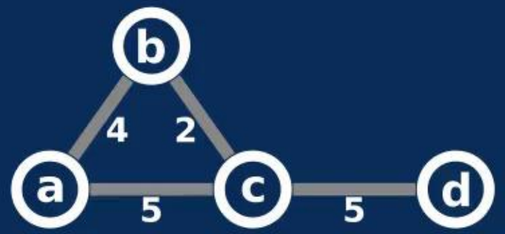
`removeEdge(v1,v2)`

- Get `v1` and `v2` = $\mathcal{O}(1)$
- Assume WLOG `v1` is min deg
- Get edge: $\mathcal{O}(deg(v_1))$

- Remove `v1`'s adj. node = $\mathcal{O}(1)$
- Remove edge from list = $\mathcal{O}(1)$
- Remove `v2`'s adj. node = $\mathcal{O}(1)$

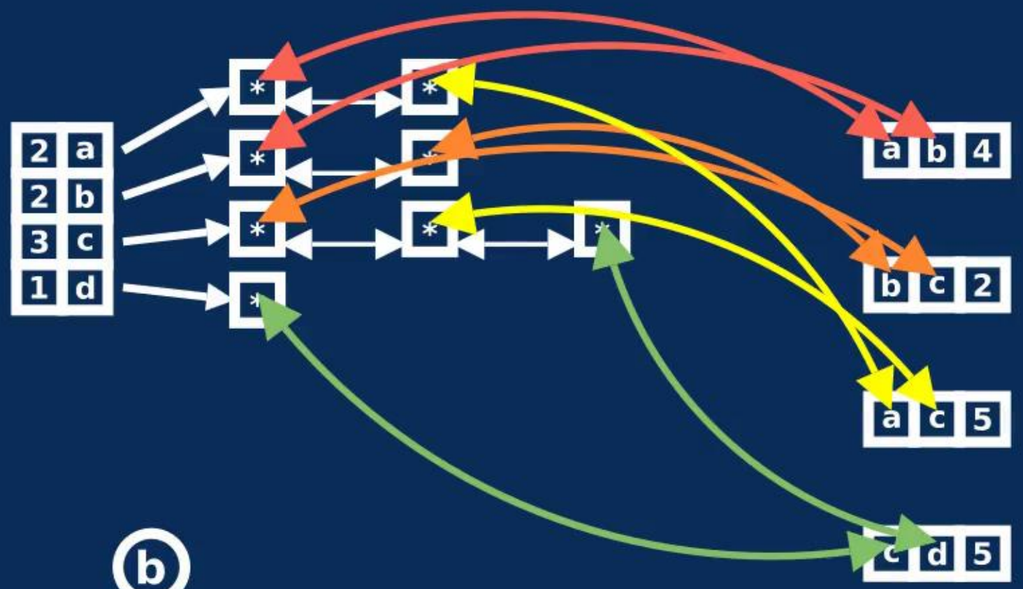
Timing

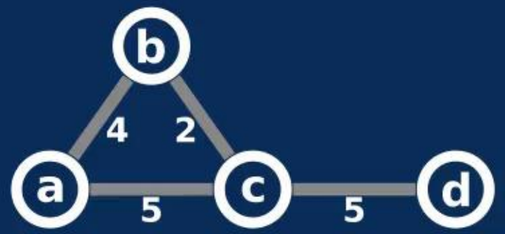





```
insertVertex(v1,key)
```

Timing

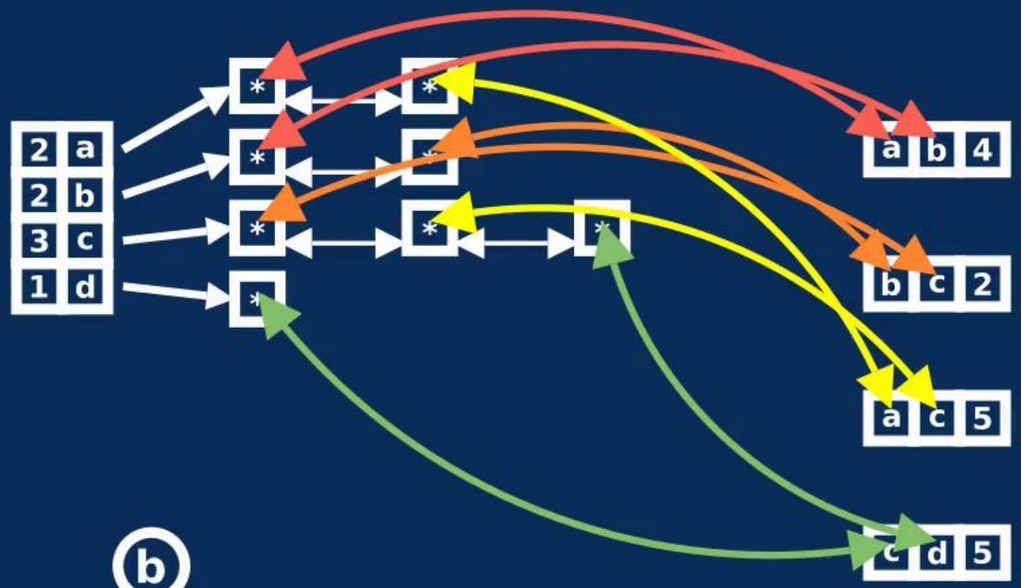


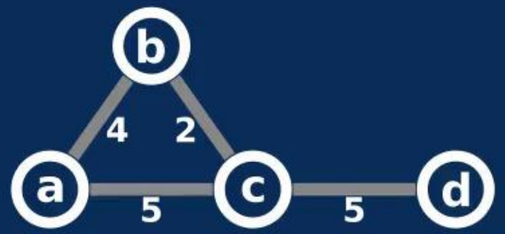


`insertVertex(v1,key)`

- Add new vertex to hash table.
- Create empty adj. list.
- Cost: $\mathcal{O}(1)$

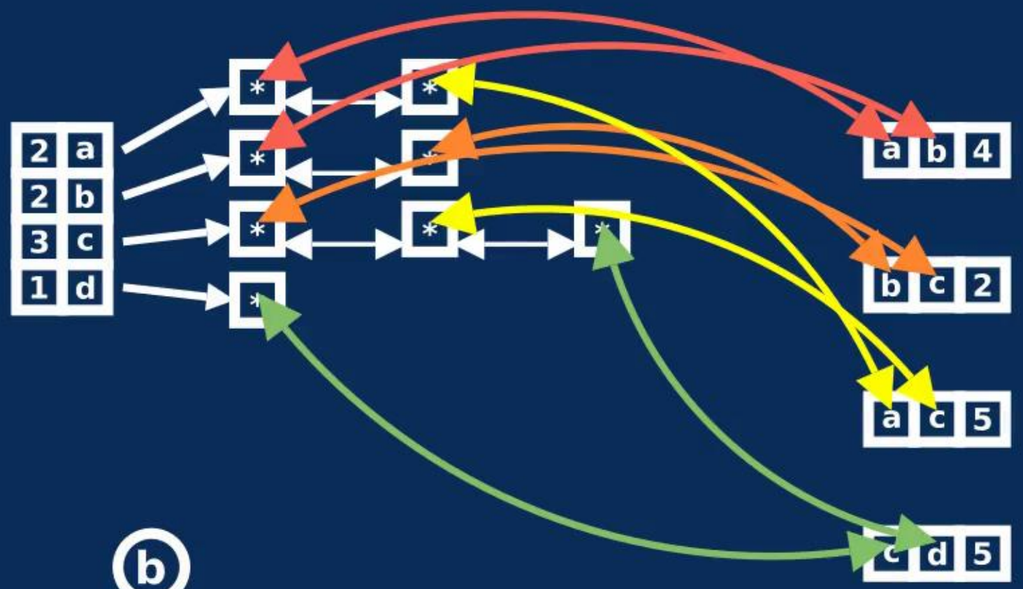
Timing

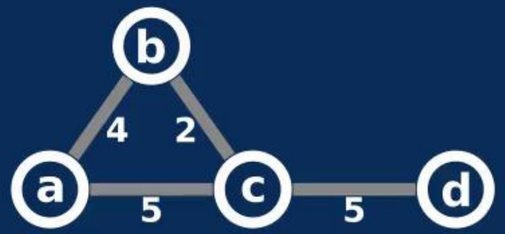




`removeVertex(v1)`

Timing





`removeVertex(v1)`

- Find Vertex $= \mathcal{O}(1)$
- Traverse adj. list $= \mathcal{O}(deg(v))$

For each adj. node:

- Remove node $= \mathcal{O}(1)$
- Remove edge $= \mathcal{O}(1)$
- Remove mirror node $= \mathcal{O}(1)$

Summary

$n = \|V\|, m = \|E\|$	**Edge List**	**Adjacency Matrix**	**Adjacency List**
Space	$\mathcal{O}(n+m)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n+m)$
`insertVertex(v)`	$\mathcal{O}(1*)$	$\mathcal{O}(n*)$	$\mathcal{O}(1*)$
`removeVertex(v)`	$\mathcal{O}(n+m)$	$\mathcal{O}(n)$	$\mathcal{O}(deg(v))$
`insertEdge(v1,v2)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1*)$
`removeEdge(v1,v2)`	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(min(deg(v_1), deg(v_2)))$
`incidentEdges(v1)`	$\mathcal{O}(m)$	$\mathcal{O}(n)$	$\mathcal{O}(deg(v))$
`areAdjacent(v1,v2)`	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(min(deg(v_1), deg(v_2)))$

Next Time

Big Idea: You can trade time for memory!

Next time: traversals!