Amortized Analysis

Operations don't stand alone!

Often, we perform a sequence of operations on data structures and we are interested in the time complexity of the entire sequence

Definition: worst-case sequence complexity (WCSC) on m operations
max. time over all sequences of m operations

WCSC ≤ m * worst-case time complexity of any one operation in a sequence of m operations

Amortized Sequence Complexity

Definition: amortized sequence complexity

$$ASC = \frac{WCSC \text{ on sequence of } m \text{ operations}}{m}$$

- Represents the "average" worst-case complexity of each operation
- Different than average-case time (no probability!)
- Often makes more sense in real situation

Approaches to Calculating Amortized Complexity

- Aggregate (add together)

  Add together costs from m operations / m
  → works well when there is only 1 operation
  → also can work when there are multiple operations

- Accounting → charge each operation a "fee" and pay real cost. Then bank the overcharge on a specific piece of the data structure

- Potential → charge every operation but now the overcharges are stored as "potential" energy in the data structure overall

Prep review

Binary Counters

Suppose we are counting up with a binary counter:

when does the 1's digit get flipped? every single time

when does the 2's digit get flipped? every 2 times

what about the 4's digit? every 4 times

k's digit gets flipped every k times

Chalk Talk

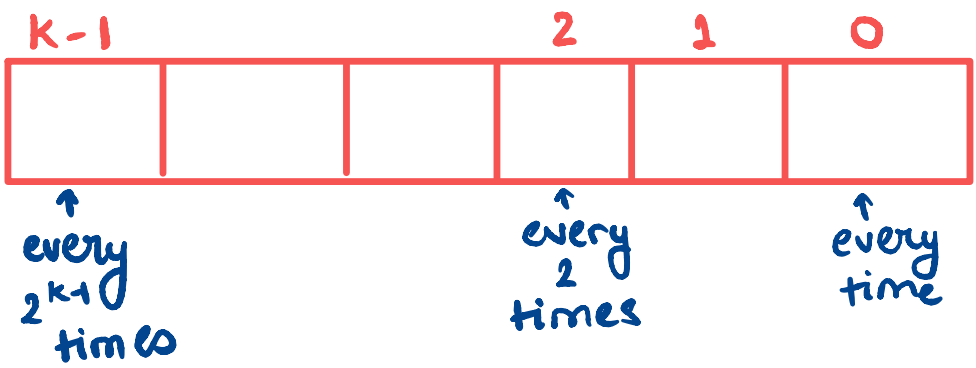
Binary Counter: Aggregate Approach

Binary Counter = sequence of k bits (k fixed) with single operation: INCREMENT

```
INCREMENT(A)
1   i ← 0
2   while i < length[A] and A[i] = 1
3       do A[i] ← 0
4          i ← i + 1
5   if i < length[A]
6       then A[i] ← 1
```

Initial counter	00000	Value = 0	
After INCREMENT	00001	Value = 1	Cost = 1
After INCREMENT	00010	Value = 2	Cost = 2
After INCREMENT	00011	Value = 3	Cost = 1
After INCREMENT	00100	Value = 4	Cost = 3
After INCREMENT	00101	Value = 5	Cost =
After INCREMENT	00110		Cost =
...			
After INCREMENT	11101	29	Cost = 1
After INCREMENT	11110	30	Cost = 2
After INCREMENT	11111	31	Cost = 1
After INCREMENT	00000	0	Cost = 5

While i have 1s, make them all 0s and make first 0 a 1.



Chalk Talk

bit #	changes	total # of changes in an operation
0	every time	m
1	every 2 ops	$\lfloor m/2 \rfloor$
2	every 4 ops	$\lfloor m/4 \rfloor$
⋮		
i	every 2^i ops	$\lfloor m/2^i \rfloor$
$K-1$	every 2^{K-1} ops	$\left\lfloor \frac{m}{2^{K-1}} \right\rfloor$

Sum up over all bits =

$$\sum_{i=0}^{K-1} \left\lfloor \frac{m}{2^i} \right\rfloor < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

total running time for m operations $O(2m) = O(m)$

Amortized cost per operation $= \frac{O(m)}{m} = O(1)$

Multipop Stack

- A stack with the regular PUSH and POP operations
- Each take O(1) time
-

```
MULTIPOP(S, k)
1   while not STACK-EMPTY(S) and k ≠ 0
2       do POP(S)
3       k ← k − 1
```

} cost of multipop is 1 * no. of pops

Worksheet → Aggregate Approach Stack

1. Using the steps below, work through an amortized analysis of a stack that in addtion to PUSH and POP, has a MULTIPOP operation.

Q1.

```
MULTIPOP(S, k)
1   while not STACK-EMPTY(S) and k ≠ 0
2       do POP(S)
3       k ← k − 1
```

(a) What is the worst-case time complexity of each of the operations PUSH, POP and MULTIPOP on a stack of n items? Assume that STACK-EMPTY is a constant time function.

Push → O(1)
Pop → O(1)
Multipop → O(min(|S|, k))

(b) Calculate an upper bound on the worst-case time for n operations by taking the time of the slowest operation (from part a above) and multiplying by n.

Slowest ~ Multipop
n operations ⇒ each worst one takes n
= n^2

(c) Would it really be possible to have a sequence of n of that operation? If it is possible, would each of the n calls be able to take the worst-case time? Either give an example (use a small n) or explain why not.

Nope! In order to get n things popd, and then nothing else to pop off. So can call on the $(n-1)^{th}$ item then n^{th}. But can't call pop on n operations n times

(d) Using aggregate analysis, argue for a better bound for a sequence of n operations.

In n operations, at most n calls to push (cost = n)
$|S| \leq n$; if |S| is at most n, in total we can have at most n pops (solos or inside multipop)
total cost ≤ 2n O(n)

(e) Then divide by n to obtain the amortized cost of a single operation.

$$\frac{O(n)}{n} = O(1)$$

Accounting Approach

Charge each operation an amortized cost "fee"

- amortized costs could be more or less than actual cost
- can be different for diff. types of operations

When amortized cost > actual cost when $\hat{c}_i > c_i$

- credit is assigned to particular element in data structure

When amortized cost < actual cost when $\hat{c}_i < c_i$

- difference MUST be paid for by existing credit

Can NEVER be in debt

$$\tilde{c}_3 + \hat{c}_2 + \hat{c}_1 \geq c_1 + c_2 + c_3$$

$$\sum_{i=1}^{k} \hat{c}_i \geq \sum_{i=1}^{k} c_i$$

→ invariant is something that is always true

Chalk Talk

Accounting Approach: Multipop Stack

	PUSH	POP	MULTIPOP
actual cost	$1	$1	$ min(|S|, K)
charge	$2	0	0

CREDIT INVARIANT: Each item in stack has $1 credit

Base Case: (empty stack)
every item has $ no items - trivially true

Assume credit invariant is true for an existing stack and any operation I do will maintain CI

Argument abt. how credit invariant holds after each operation

PUSH → charge $2 and costs $1 so leave $1 on the new element

POP → charge $0 but pay for the pop from the dollar that was on the item

Since credit invariant says credit ≥ 0, we are never in debt.

Each operation's amortized cost ≤ 2

∴ O(1) per operation

Worksheet

Binary Counter: Accounting Approach

Binary Counter = sequence of k bits (k fixed) with single operation: INCREMENT

```
INCREMENT(A)
1   i ← 0
2   while i < length[A] and A[i] = 1
3       do A[i] ← 0
4       i ← i + 1
5   if i < length[A]
6       then A[i] ← 1
```

Initial counter	00000	Value = 0	
After INCREMENT	00001	Value = 1	Cost = 1
After INCREMENT	00010	Value = 2	Cost = 2
After INCREMENT	00011	Value = 3	Cost = 1
After INCREMENT	00100	Value = 4	Cost = 3
After INCREMENT	00101	Value = 5	Cost =
After INCREMENT	00110		Cost =
…			
After INCREMENT	11101	29	Cost = 1
After INCREMENT	11110	30	Cost = 2
After INCREMENT	11111	31	Cost = 1
After INCREMENT	00000	0	Cost = 5

Worksheet

Accounting Approach: Binary Counter

Charge $2 for INCREMENT actual cost is $ per bit that flips

Credit Invariant → At any step in the sequence, each bit in the counter that is equal to 1 has $1 credit

Proof By induction

Initially counter is 0 and no credit → no 1 bits so trivially true

Assume invariant holds, call Increment.

Cost of flipping leading 1s to 0s paid for $1 credit on each 1.

Cost of flipping only 0 that goes to 1 paid for by the first $ in the fee. The second $ in the fee stored as a $1 credit on that 1.

No other bits change

∴ Invariant holds

This shows data structure never goes in debt

So amortized cost per operation ≤ 2

O(1)

allocated → amt. of space available
size → no. of elements in array

arrays have to be contiguous memory

Dynamic Arrays: Reminder

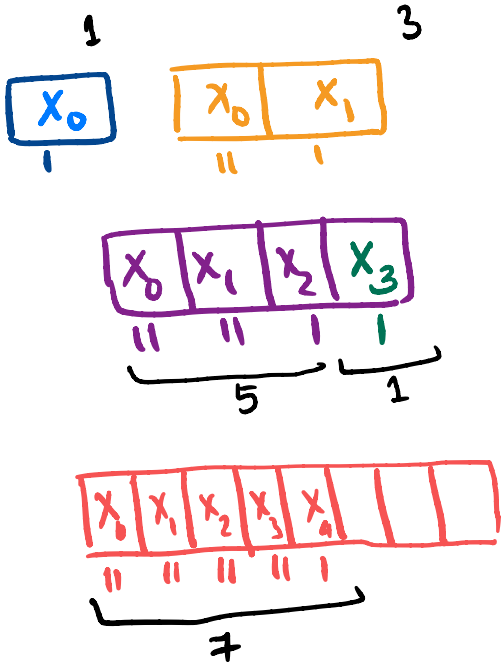
```
1  def insert(A, x):
2      # Check whether current array is full
3      if A.size == A.allocated:
4          new_array = new array of length (A.allocated * 2)
5          copy all elements of A.array into new_array
6          A.array = new_array
7          A.allocated = A.allocated * 2
8
9      # insert the new element into the first empty spot
10     A.array[A.size] = x
11     A.size = A.size + 1
```

2 * size
= 2 * allocated

→ 1 write operation

size
~~0~~ 1 ~~2~~ ~~3~~ ~~4~~ 5

allocated
~~1~~ ~~2~~ ~~4~~ 8



$$\sum_{i=0}^{b} 2^i \rightarrow 2^{b+1} - 1$$

Chalk Talk

Dynamic Arrays: Copying Elements

Time for n operations

$$T(n) = \sum_{k=0}^{n-1} t_k$$

$$t_k = \begin{cases} 2k+1, & k \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

Let $t_k' = t_k - 1$

$$t_k = t_k' + 1$$

$$t_k' = \begin{cases} 2k, & k \text{ is a power of 2} \\ 0, & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{n-1} t_k' + n$$

$$= n + \sum_{i=0}^{\lfloor \log(n-1) \rfloor} t_{(2^i)}$$

$$= n + 2 \sum_{i=0}^{\lfloor \log(n-1) \rfloor} 2^i$$

$$= n + 2\left(2^{\lfloor \log(n-1) \rfloor + 1} - 1\right)$$

$$\leq n + 2\left[2 \cdot 2^{\log(n-1)} - 1\right]$$

$$= n + 2\left[2(n-1) - 1\right]$$

$$= 5n - 6$$

$$T(n) \in O(n)$$

Single operation amortized complexity is $O(1)$

1. Using the steps below, work through an amortized analysis of a stack that in addtion to PUSH and POP, has a MULTIPOP operation.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k ≠ 0
2      do POP(S)
3      k ← k − 1
```

   (a) What is the worst-case time complexity of each of the operations PUSH, POP and MULTIPOP on a stack of n items? Assume that STACK-EMPTY is a constant time function.

   (b) Calculate an upper bound on the worst-case time for n operations by taking the time of the slowest operation (from part a above) and multiplying by n.

   (c) Would it really be possible to have a sequence of n of that operation? If it is possible, would each of the n calls be able to take the worst-case time? Either give an example (use a small n) or explain why not.

   (d) Using aggregate analysis, argue for a better bound for a sequence of n operations.

   (e) Then divide by n to obtain the amortized cost of a single operation.

2. Analyze the INCREMENT operation on a binary counter that starts at zero using the accounting method. Use a cost of \$1 to represent the real cost of flipping a bit. Here is a reminder of the INCREMENT algorithm.

```
INCREMENT(A)
1   i ← 0
2   while i < length[A] and A[i] = 1
3       do A[i] ← 0
4       i ← i + 1
5   if i < length[A]
6       then A[i] ← 1
```

3. Conduct an amortized analysis of insertions into a dynamic array where the expansion factor is 2. Use the code from the W7 Quercus module (also in the W7 slides). When we calculated the worst-case performance we decided to count array accesses. We determined that these only happened on lines 10 and 5. When the underlying array had room for a new element, insert only required one access on line 10. But when the underlying array was full with n elements, inserting one more required $2n$ accesses on line 5 and then 1 more on line 10.

   (a) Initially do the analysis using a charge of \$2 for each insert. Is this enough of a charge?

   (b) Try again with \$3 per insert. That will leave us a \$2 credit on the first item inserted. Intuitively, that feels like it will be enough to the do the copy when we have to copy this item to the larger array. Can you prove that this will work and the data structure will never be in debt?

   (c) Try again with \$5 as the charge.

1. Using the steps below, work through an amortized analysis of a stack that in addtion to PUSH and POP, has a MULTIPOP operation.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k ≠ 0
2      do POP(S)
3      k ← k − 1
```

   (a) What is the worst-case time complexity of each of the operations PUSH, POP and MULTIPOP on a stack of n items? Assume that STACK-EMPTY is a constant time function.

   SOLUTION: PUSH and POP are constant time. MULTIPOP does minimum(k, s) POP operations where s is the size of the stack with MULTIPOP is called. This is $\mathcal{O}(n)$

   (b) Calculate an upper bound on the worst-case time for n operations by taking the time of the slowest operation (from part a above) and multiplying by n.
   SOLUTION: Worst case is n MULTIPOP operations which gives an upper bound of $\mathcal{O}(n^2)$

   (c) Would it really be possible to have a sequence of n of that operation? If it is possible, would each of the n calls be able to take the worst-case time? Either give an example (use a small n) or explain why not.
   SOLUTION: You can call MULTIPOP n times but if you did, each operation would have the stack empty (because you did no inserts) and so the cost of each would be 0 not $\mathcal{O}(n)$.

   (d) Using aggregate analysis, argue for a better bound for a sequence of n operations.
   SOLUTION: In n operations, an upper bound on the number of calls to insert is n. That takes $\mathcal{O}(n)$ time. Each element can only be popped once and so the combined number of calls to POP (either directly to POP or within MULTIPOP) is $\mathcal{O}(n)$ and each takes 1. So in total in the n operations and upper bound on the calls to POP is $\mathcal{O}(n)$. So combined, the n operations are bounded by $\mathcal{O}(n)$.

   (e) Then divide by n to obtain the amortized cost of a single operation.
   SOLUTION: $\mathcal{O}(n)$ / $n = \mathcal{O}(1)$ amortized cost per operation.

2. Analyze the INCREMENT operation on a binary counter that starts at zero using the accounting method. Use a cost of \$1 to represent the real cost of flipping a bit. Here is a reminder of the INCREMENT algorithm.

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] and A[i] = 1
3      do A[i] ← 0
4      i ← i + 1
5  if i < length[A]
6      then A[i] ← 1
```

   SOLUTION: Taken up during lecture. For a written description see Chapter 17.2 from CLRS textbook.

3. Conduct an amortized analysis of insertions into a dynamic array where the expansion factor is 2. Use the code from the W7 Quercus module (also in the W7 slides). When we calculated the worst-case performance we decided to count array accesses. We determined that these only happened on lines 10 and 5. When the underlying array had room for a new element, insert only required one access on line 10. But when the underlying array was full with n elements, inserting one more required $2n$ accesses on line 5 and then 1 more on line 10.

   (a) Initially do the analysis using a charge of \$2 for each insert. Is this enough of a charge?

   (b) Try again with \$3 per insert. That will leave us a \$2 credit on the first item inserted. Intuitively, that feels like it will be enough to the do the copy when we have to copy this item to the larger array. Can you prove that this will work and the data structure will never be in debt?

   (c) Try again with \$5 as the charge.

   SOLUTION: Taken up during lecture. There is a discussion of this in CLRS as well but they use a different unit of analysis where they consider only the real cost of the array write operations.

WEEK 1 PREP

5. Aggregate Method

$$= \frac{\sum_{i=1}^{\log_5 n} 5^i + (n - \log_5 n)}{n}$$

$$= \frac{\sum_{i=1}^{\log_5 n} 5^i}{n} + 1 - \frac{\log_5 n}{n}$$

$$= \frac{5^{\log_5 n + 1} - 1}{(5-1)n} + \frac{\log_5 n}{n} + 1$$

$$= \left(\frac{5^{\log_5 n}}{n}\right)\frac{5}{4} - \frac{1}{4n} + \frac{\log_5 n}{n} + 1$$

As $n \to \infty$, $\frac{\log n}{n} \to 0$

$$= \left(\frac{n}{n}\right)\frac{5}{4} - \frac{1}{4n} + 1 \qquad \left(5^{\log_5 n} = n\right)$$

$$= \frac{5}{4} - \frac{1}{4n} + 1$$

$$= 2.25$$

Graphs

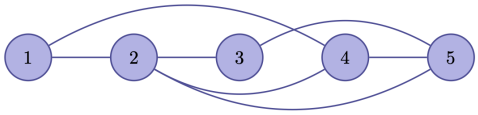
This week: Representations and Breadth First Search

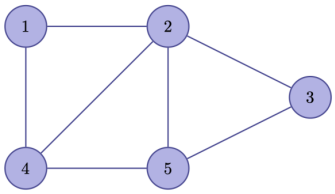
Announcements

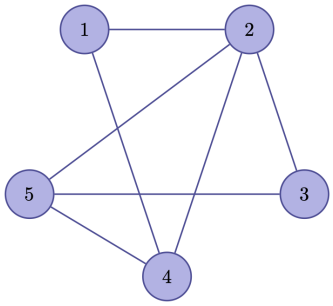
- Don't discuss your PS2 answers quite yet

- Test 2 in-person on Friday March 18
  During your "TUT" time on ACORN
  Rooms TBA

Are these graphs the same or different?



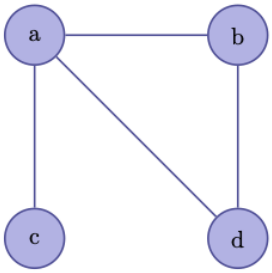




In CS, all are same.

Chalk Talk

Adjacency Lists



list of all vertices

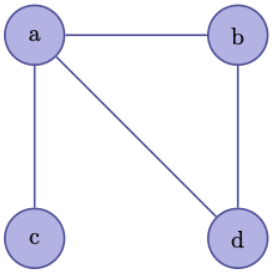
a : b, c, d
b : a, d
c : a
d : a, b

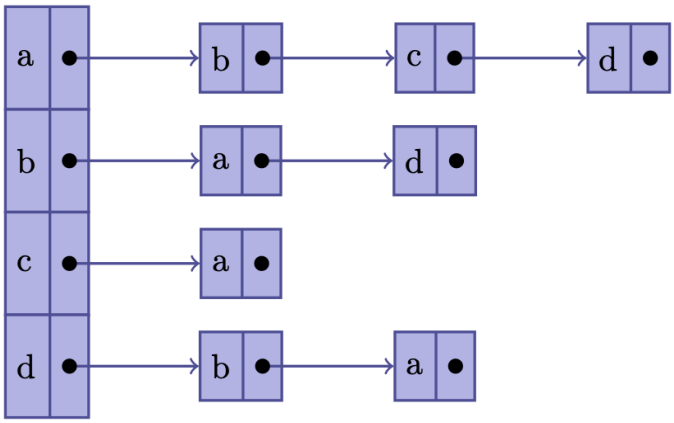
sublist of adjacent vertices

Chalk Talk

Adjacency Lists

every node is there twice



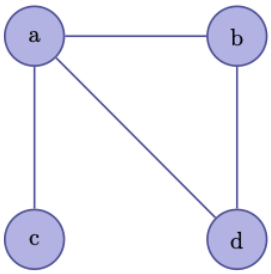


* UNDIRECTED GRAPH *

↳ $2^{\text{no. of edges}}$ = no. of 1s

Chalk Talk

Adjacency Matrix



	a	b	c	d
a	0	1	1	1
b	1	0	0	1
c	1	0	0	0
d	1	1	0	0

0 → not connected

1 → connected

diagonal is vertex connecting to itself (not allowed in undirected graph)

which representation is better?

Space

Adj list → $\Theta(|V| + |E|)$

Adj matrix → $\Theta(|V|^2)$

Time

Operations?

- Add / Remove vertex
- Add / Remove edge
- Edge Query - given vertices u, v is (u, v) in E?

Adj list → $\Theta(|V|)$

- Neighbourhood Query

In-neighborhood: given vertex u, return set of vertices such that (v, u) in G

Out-neighbourhood: given vertex u, return set of vertices such that (u, v) in G

↳ easier with adj list

Worksheet

Q1 - Q6

Breadth First Search

- Starting from source s in V, BFS visit every vertex v reachable from s.
- In the process, we find paths from s to each reachable v
  paths make BFS tree rooted at s
- Works on directed and undirected graphs

- Keeping track of progress by colouring each vertex
  - White → all vertices start white (not discovered)
  - Grey → discovered but not explored
  - Black → fully explored
    ↳ immediate edges explored for BFS

Breadth First Search

- Keep track of
  - parent of v in BFS tree π
  - distance from s to v d – depth
- Grey vertices stored in Queue so they are handled in FIFO order
- Use adjacency list order

Breadth First Search

```
BFS(G=(V,E, s):
1    for all v in V:
2        colour[v] ← white
3        d[v] ← inf
4        π[v] ← NIL
5    colour[s] ← grey
6    d[s] ← 0
7    initialize empty queue Q
8    ENQUEUE(Q, s) #loop invariant Q contains exactly the grey vertices
9    while Q not empty:
10       u ←DEQUEUE(Q)
11       for each edge (u,v) in E:
12           if colour[v] == white:
13               colour[v] ← grey
14               d[v] ← d[u] + 1
15               π[v] ← u
16               ENQUEUE(Q,v)
17       colour[u] ← black
```

initialise

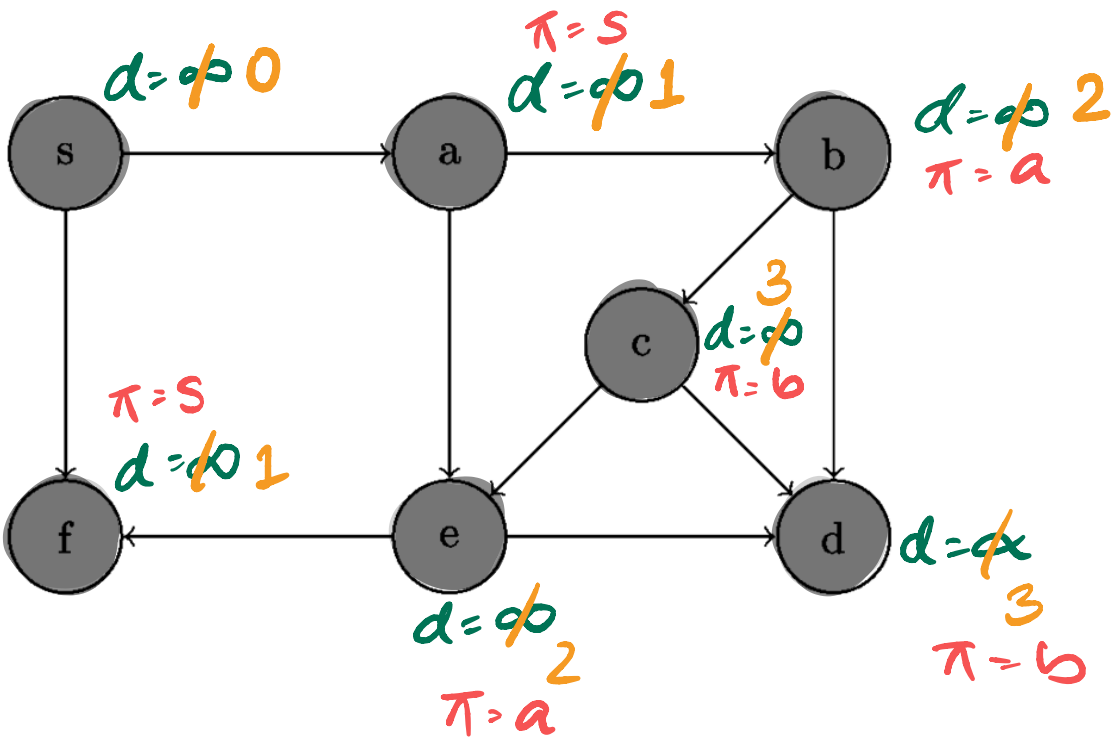
starting vertex

intialise s

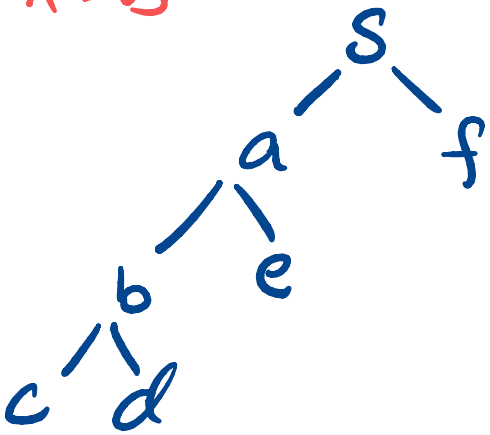
woohoo!! found new vertex

Chalk Talk

Breadth First Search



- s: a, f
- a: b, e
- b: c, d
- c: d, e
- d:
- e: d, f
- f:



BFS: Complexity

Each vertex is enqueued at most once when it is white.

So, while loop iterates at most $O(|V|)$ times.

But, the adj list of each node is examined at at most once in total.

So total running time is $O(|V| + |E|)$

BFS: Claim to find shortest paths

Define $\delta(s, v)$ = minimum distance from s to v

smallest no. of edges on any path from s to v.

Claim: At the end of BFS, d[v] = $\delta(s, v)$

Aside: later when we have weighted graphs we will define shortest path differently. For now, all edges in the path count as 1.

Chalk Talk

Lemma $\delta(s, v) \leq \delta(s, u) + 1$ for all (u,v) in E.

Proof Idea

Lemma $d[v] \geq \delta(s, v)$ for all v in V, at any point during BFS

Proof Idea

Lemma if $Q = [v_1, \ldots v_r]$, then $d[v_i] \leq d[v_{i+1}]$ for each i and $d[v_r] \leq d[v_1] + 1$

Proof Idea

Chalk Talk

Theorem: At the end of BFS, d[v] = δ(s,v) for all v

means for v
$d[v] \leq \delta(s,v)$
$d[v] = \delta(s,v)$

Assume: d[v] > δ(s,v) for some v with minimum δ(s,v)

v is the first vertex on some path when $\delta(s,v)$ is shorter.
Let u be the first vertex just before v on that path. So when we discovered v and and set d[v] we were assuming (u,v)

$d[v] > \delta(s,v)$

$\delta(s,v) = \delta(s,u) + 1$

$\delta(s,u) + 1 = d[u] + 1$

$d[v] > d[u] + 1$

Now lets consider the point when we dequeue vertex u in line 10. Then we explore edge (u,v) and v is coloured.

white: woohoo! Set $d[v] = d[u] + 1$ Contradiction
Enqueue v
colour it
never change d[v]

black: means? vertex v was in the Queue earlier before u.
$d[v] \leq d[u]$ from 3rd lemma ∴ contradiction

grey: it was painted grey by some other vertex w.
$d[v] = d[w] + 1$
$\leq d[u] + 1$
$d[w] \leq d[u]$ looking at edge (u,v)

Contradiction

Chalk Talk

Pages 235-236 CLRS

Lemma

$\delta(s, v) \leq \delta(s, u) + 1$ for all (u,v) in E. (directed or undirected)

Proof Idea

either ① u is reachable from s, so v is also reachable. In that case, $\delta(s,v)$ cannot be longer than $\delta(s,u)+1 \Rightarrow \delta(s,v) \leq \delta(s,u)+1$

② u is not reachable from s: then $\delta(s,u) = \infty$ and the inequality still holds

Lemma

$d[v] \geq \delta(s, v)$ for all v in V, at any point during BFS ← Inductive hypothesis

Proof Idea

↓ Induction on Enqueue operations

→ when s is enqueued, $d[s] = 0 = \delta(s,s)$

$d[v] = \infty \geq \delta(s,v)$ $\forall v \in V - \{s\}$

Line 14 in code → when v is discovered from a parent u, then $d[u] \geq \delta(s,u)$ (by inductive hypothesis)

$\Rightarrow d[v] = d[u] + 1 \geq \delta(s,u) + 1 \geq \delta(s,v)$ (by lemma 1)

→ v is enqueued & $d[v]$ never changes again so the inductive hypothesis is maintained

Lemma

↓ queue has at most two values of d at any point

Proof Idea

if $Q = [v_1, \ldots v_r]$, then $d[v_i] \leq d[v_{i+1}]$ for each i and ← Inductive hypothesis

$d[v_r] \leq d[v_1] + 1$ ← increasing distance in queue → $d[v_r] = d[v_1] + 1$ or less

→ when s is enqueued, lemma holds

→ when v_1 is dequeued, v_2 becomes the head. By inductive hypothesis, $d[v_1] \leq d[v_2]$ & $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$. Lemma holds!

→ when v is enqueued, it becomes v_{r+1}. The parent of v is u, and it has just been dequeued. v_1 is the new head. So $d[v_1] \geq d[u]$.

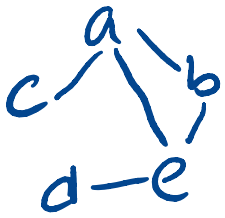
$\Rightarrow d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Also, $d[v_r] \leq d[u] + 1$

→ $d[v_r] \leq d[v] = d[v_{r+1}]$ so lemma holds!

1. Draw the undirected graph represented by this adjacency list.

```
a: e, c, b
b: a, e
c: a
d: e
e: d, a, b
```

→ connected
→ undirected

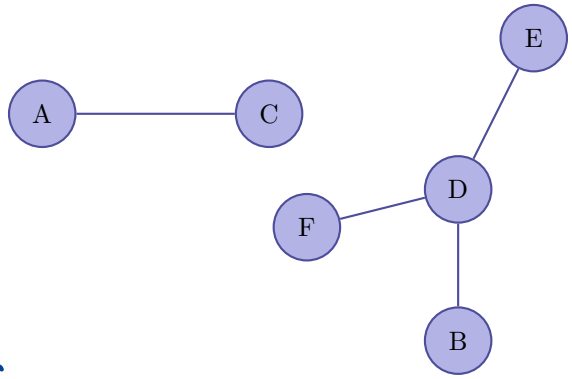


2. Write out the adjacency matrix representing this same graph.

	a	b	c	d	e
a	0	1	1	0	1
b	1	0	0	0	1
c	1	0	0	0	0
d	0	0	0	0	1
e	1	1	0	1	0

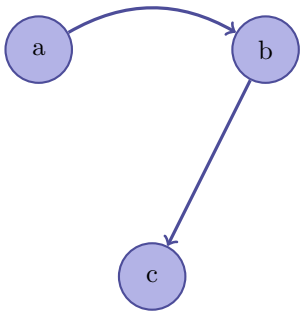
0 → not connected
1 → connected

3. Write the adjacency list representation and the adjacency matrix representation of this unconnected undirected graph.

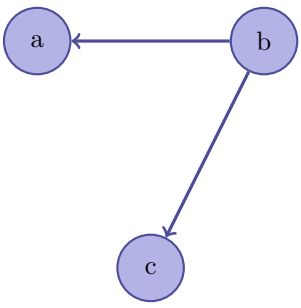


A : C
B : D
C : A
D : B, F, E
E : D
F : D

4. Now, we will use an adjacency list to represent a directed graph. Recall that in a directed graph, v is adjacent to u if the graph contains edge (u,v), but that edge (u,v) does not make u adjacent to v. Give the adjacency lists for these two graphs.



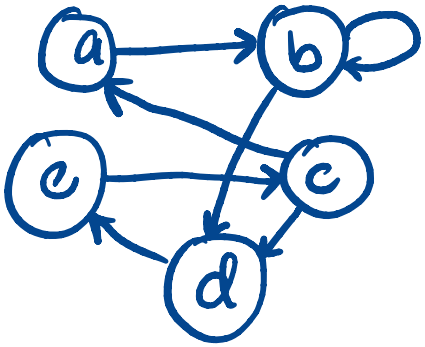
a : b
b : c
c : []



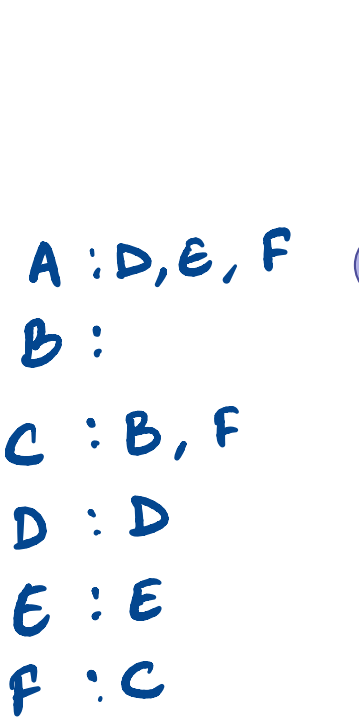
a : []
b : a, c
c : []

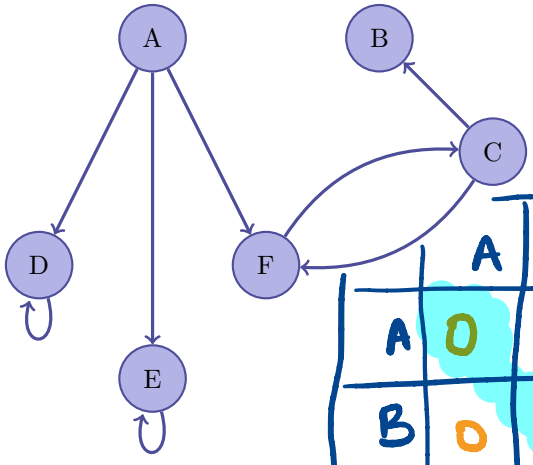
5. Draw the directed graph represented by this adjacency list.

```
a: b
b: b, d
c: a, d
d: e
e: c
```



6. Give the adjacency matrix representation for this directed graph.





TO

FROM	A	B	C	D	E	F
A	0	0	0	1	1	1
B	0	0	0	0	0	0
C	0	1	0	0	0	1
D	0	0	0	1	0	0
E	0	0	0	0	1	0
F	0	0	0	0	0	1

7. Here are some questions to discuss with your neighbours.

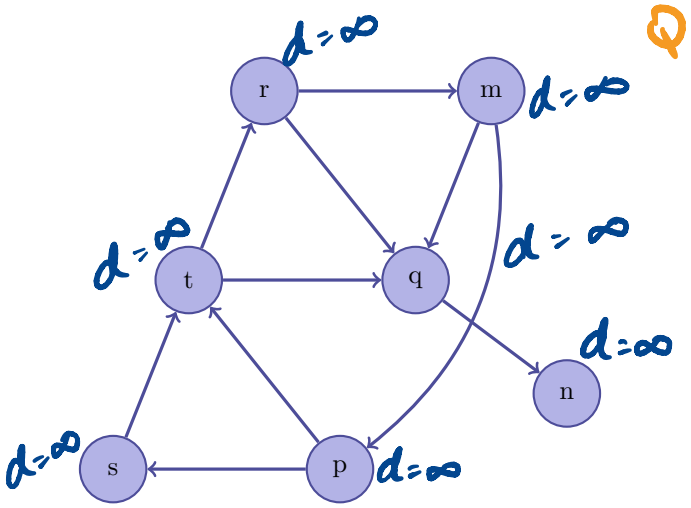
   (a) Which edges are on the diagonal of an adjacency matrix? → (D,D), (E,E), (F,F)

   (b) How do we determine the out-neighbourhood of a given vertex in each representation? What is the complexity of this operation in each representation?

   (c) How do we determine the in-neighbourhood of a given vertex? What is the complexity of this operation in each representation?

SOLVE (b) and (c)

8. Run BFS(G,s) on the directed graph G shown here using vertex s as the starting vertex. Draw the BFS tree that results. What is its height? How many edges are in it? How many vertices?



9. Run BFS(G,t) on the same graph. (Use vertex t as the starting vertex.) What is the height of the resulting BFS tree? How many edges are in it? How many vertices?

10. Discuss with your group the following questions. What will be the resulting tree when we start from vertex q? What will be its height and how many vertices will be in it?

11. Draw the directed graph that is represented by the following adjacency list.

```
s: a, b
a: s, c
b: s, c
c: b, a
```

Run BFS on this graph starting with vertex s (using the adjacency list representation) and highlight the resulting tree. For each vertex indicate its distance from s.

Repeat this same exercise with the graph represented by this adjacency list.

```
s: b, a
a: s, c
b: s, c
c: b, a
```

What do you notice about the graphs, the trees, and the distances?

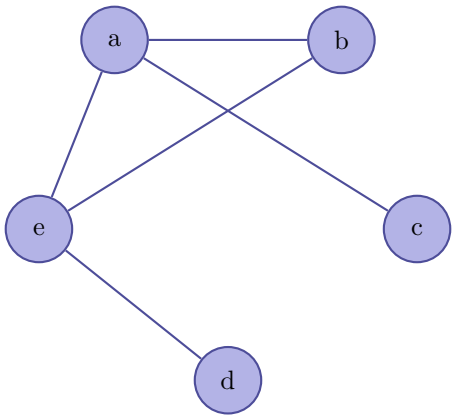
12. Assume that we have a group of students who have worked on many assignments in various partnerships. We have a list of all these past partnerships and a list of all the students. We want to know if it is possible to divide the group into two classes so that nobody is in the same class with anyone with whom they have previously been a partner.

    Devise an algorithm to determine if this is possible and if it is possible, to generate the two classlists.

1. Draw the undirected graph represented by this adjacency list.

```
a: e, c, b
b: a, e
c: a
d: e
e: d, a, b
```

SOLUTION:

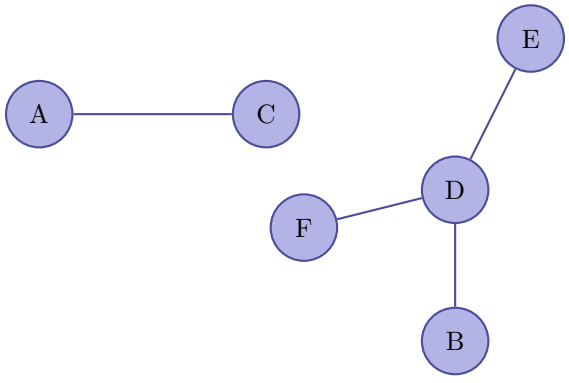


2. Write out the adjacency matrix representing this same graph.

SOLUTION:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

3. Write the adjacency list representation and the adjacency matrix representation of this unconnected undirected graph.

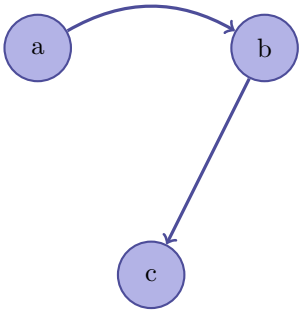


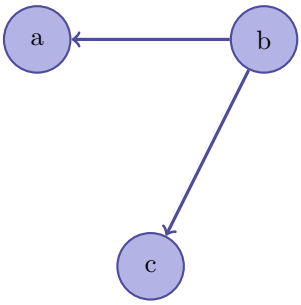
SOLUTION:

```
A: C
B: D
C: A
D: E, F, B
E: D
F: D
```

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

4. Now, we will use an adjacency list to represent a directed graph. Recall that in a directed graph, v is adjacent to u if the graph contains edge (u,v), but that edge (u,v) does not make u adjacent to v. Give the adjacency lists for these two graphs.

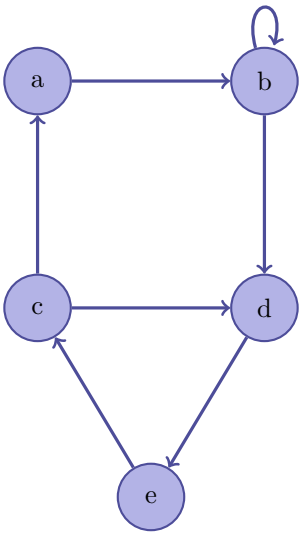




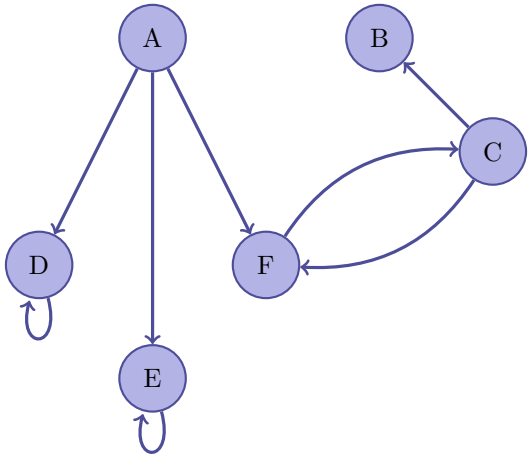
5. Draw the directed graph represented by this adjacency list.

```
a: b
b: b, d
c: a, d
d: e
e: c
```

Solution:



6. Give the adjacency matrix representation for this directed graph.



Solution:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

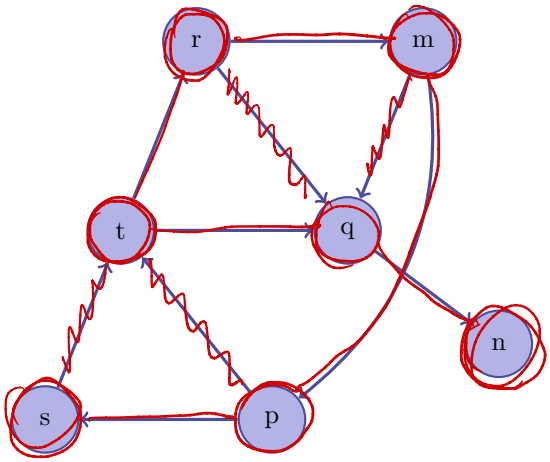
7. Here are some questions to discuss with your neighbours.

   (a) Which edges are on the diagonal of an adjacency matrix?

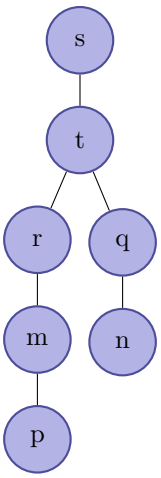
   (b) How do we determine the out-neighbourhood of a given vertex in each representation? What is the complexity of this operation in each representation?

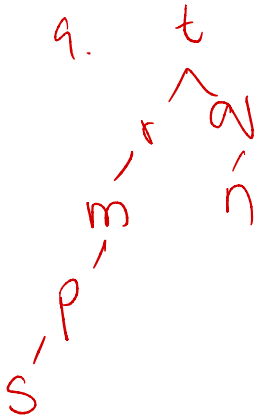
   (c) How do we determine the in-neighbourhood of a given vertex? What is the complexity of this operation in each representation?

8. Run BFS(G,s) on the directed graph G shown here using vertex s as the starting vertex. Draw the BFS tree that results. What is its height? How many edges are in it? How many vertices?



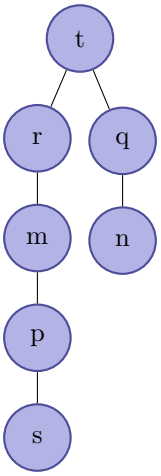
SOLUTION:





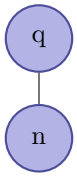
This BFS tree has **7 vertices**, **6 edges**, and **height 4**.

9. Run BFS(G,t) on the same graph. (Use vertex t as the starting vertex.) What is the height of the resulting BFS tree? How many edges are in it? How many vertices?
SOLUTION:



This BFS tree has **7 vertices**, **6 edges**, and **height 4**.

10. Discuss with your group the following questions. What will be the resulting tree when we start from vertex q? What will be its height and how many vertices will be in it?
SOLUTION:



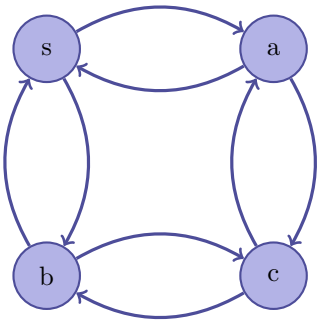
This BFS tree has **2 vertices**, **1 edge**, and **height 1**.

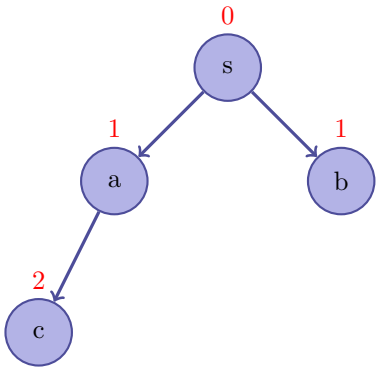
11. Draw the directed graph that is represented by the following adjacency list.

```
s: a, b
a: s, c
b: s, c
c: b, a
```

Run BFS on this graph starting with vertex s (using the adjacency list representation) and highlight the resulting tree. For each vertex indicate its distance from s.

SOLUTION:

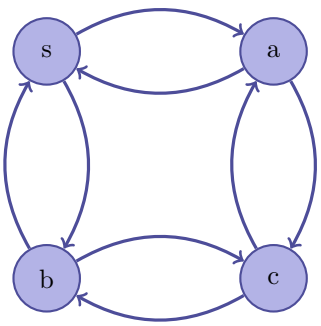


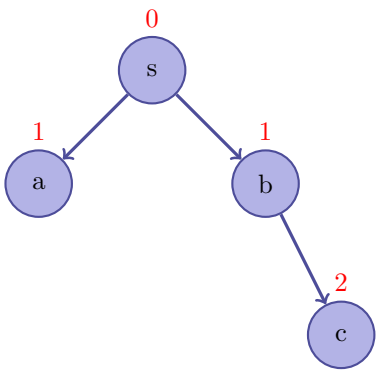


Repeat this same exercise with the graph represented by this adjacency list.

```
s: b, a
a: s, c
b: s, c
c: b, a
```

SOLUTION:





What do you notice about the graphs, the trees, and the distances?

SOLUTION: The graphs are the same. The trees are different, but the distance from s to each vertex is the same on both graphs.

12. Assume that we have a group of students who have worked on many assignments in various partnerships. We have a list of all these past partnerships and a list of all the students. We want to know if it is possible to divide the group into two classes so that nobody is in the same class with anyone with whom they have previously been a partner.

    Devise an algorithm to determine if this is possible and if it is possible, to generate the two classlists.

    SOLUTION: Discussed in lecture: The vertices are students and the edges represent a partnership between two students. The graph is undirected. In the evening class we noted that the depth of each vertex in the BFS tree is either odd or even. In the day class we talked about assigning colours (red and blue) to the two groups. The basic idea is to run BFS and whenever you encounter a grey or black vertex, check if its d value (depth or distance) is the parity that we would need (odd if the parent is even and even if the parent is odd). Using the colours approach, a vertex needs to be red if its parent is blue and vice versa. If we ever encounter a vertex that needs to be both odd and even (or needs to be both red and blue) then we can't divide the class. If we don't encounter a problem before we finish the full graph, then the class can be divided into red and blue students (or odd and even students.) Finally, we noted that this kind of graph has a name – a *bipartite* graph.