I pledge my honor that I will not violate the policies of this examination.

(Print Your First Name and Last Name) Ella Grady

Ella Grady
CS 160
12/9/21

1. (Sorting, 9 points) Design a random, typical array of at least 7 elements (numbers, characters, strings… your choice) including one duplicate element that occurs at least twice in your array. Use this array as the starting point in the tracing of each of the algorithms in this question. Also, use subscripts to identify duplicate elements, such as A_1 and A_2, to keep track of these elements and show the algorithms' stability properties.

   For each of the sorting algorithms covered in chapter 2, except 3-way quick sort (since it is for arrays mostly consisting of duplicate elements, not the scope here)
   a. give a high-level description of how it work in a few sentences, but not code or pseudo-code.
   b. State the characteristics of best-case inputs and the algorithm's best-case time complexity, and the characteristics of worst-case inputs the algorithm's worst-case time complexity.
      If the algorithm works the same for all arrays and doesn't really have best-case or worst-case inputs, explain why that is so, and give the algorithm's time complexity.
   c. Trace the sorting algorithm on your random, typical array, and show all the steps from the start to the sorted array. Also, choose an intermediate step to explain the invariants of the sorting algorithm, such as which part of the sub-array is sort, what about the other elements, etc.

1. arr = [O W N A_1 R_1 R_2 A_2 Y]

selection sort

a. selection sort works by moving through the array, finding the smallest element and exchanging it with the first element repeating this process for the entire array, each time looking at the next element in the array (left to right) and exchanging it with the next smallest element.

b. selection sort works the same for all arrays, even if already sorted, because it will go through the process for every element, and will recognize an element as the smallest element and will exchange it even if it is already in the right position.
time complexity: $O(n^2)$ — n = number of elements in array

c.

i	min	0	1	2	3	4	5	6	7
		O	W	N	A_1	R_1	R_2	A_2	Y
0	3	O	W	N	A_1	R_1	R_2	A_2	Y
1	6	A_1	W	N	O	R_1	R_2	A_2	Y
2	2	A_1	A_2	N	O	R_1	R_2	W	Y
3	3	A_1	A_2	N	O	R_1	R_2	W	Y
4	4	A_1	A_2	N	O	R_1	R_2	W	Y
5	5	A_1	A_2	N	O	R_1	R_2	W	Y
6	6	A_1	A_2	N	O	R_1	R_2	W	Y
7	7	A_1	A_2	N	O	R_1	R_2	W	Y
		A_1	A_2	N	O	R_1	R_2	W	Y

anything in gray is in final position
A_1 W N O R_1 R_2 A_2 Y
↑
invariants: elements left of ↑ are fixed and sorted in ascending order (A_1)
no elements to right of ↑ are smaller than those to left of ↑
(W N O R_1 R_2 A_2 Y) ≥ (A_1)
the ↑ is moved through array as i pointer is incremented, ensuring all to the left is sorted and all to the right is not smaller than the left.

1. insertion sort

a. insertion sort works by considering each element and comparing it to the elements before it in the array, shifting all the elements it is less than right one index to place the current element in order.

b. Best case inputs: already sorted arrays
best case time complexity: $O(n)$ — n = num. elements in array
worst case inputs: reverse sorted arrays
worst case time complexity: $O(n^2)$ — n = num. elements in array

c.

i	j	0	1	2	3	4	5	6	7
		O	W	N	A_1	R_1	R_2	A_2	Y
1	0	O	W	N	A_1	R_1	R_2	A_2	Y
2	0	O	W	N	A_1	R_1	R_2	A_2	Y
3	0	N	O	W	A_1	R_1	R_2	A_2	Y
4	3	A_1	N	O	W	R_1	R_2	A_2	Y
5	4	A_1	N	O	R_1	W	R_2	A_2	Y
6	1	A_1	N	O	R_1	R_2	W	A_2	Y
7	7	A_1	A_2	N	O	R_1	R_2	W	Y
		A_1	A_2	N	O	R_1	R_2	W	Y

→ anything in gray do not move
N O W A_1 R_1 R_2 A_2 Y
↑
invariants: elements to left of ↑ are in ascending order
elements to right of ↑ haven't been seen yet

natural merge sort

a. natural merge sort splits the array in half, sorts both halves separately and then merges them so they are in increasing order

b. For any array, merge sort will take $O(n \log n)$ time because, regardless of how the array is currently sorted, it will still split the array in half recursively to sort everything

c.

	0	1	2	3	4	5	6	7
	O	W	N	A_1	R_1	R_2	A_2	Y
	O	W	N	A_1	R_1	R_2	A_2	Y
sort left	O	W	N	A_1	R_1	R_2	A_2	Y
sort left	O	W	N	A_1	R_1	R_2	A_2	Y
sort right	O	W	A_1	N	R_1	R_2	A_2	Y
merge	A_1	N	O	W	R_1	R_2	A_2	Y
sort right	A_1	N	O	W	R_1	R_2	A_2	Y
sort left	A_1	N	O	W	R_1	R_2	A_2	Y
sort right	A_1	N	O	W	R_1	R_2	A_2	Y
merge	A_1	N	O	W	A_2	R_1	R_2	Y
merge	A_1	A_2	N	O	R_1	R_2	W	Y

→ in this step the right half subarray of the left half subarray is being sorted within itself. The preceding subarray is also sorted within itself, but anything in the full array's right half subarray has not yet been sorted.

1. mergesort:

top-down:

a. mergesort splits array into 2 sorted subarrays: arr[lo ... mid] and arr[mid+1 ... hi] and merges them into one array moving through full array recursively

b. For any array, mergesort will take O(n log n) time because, regardless of how the array is currently sorted, it will still split the array in half recursively to sort everything

c.

	0	1	2	3	4	5	6	7
	O	W	N	A_1	R_1	R_2	A_2	Y
merge(a,0,0,1)	O	W	N	A_1	R_1	R_2	A_2	Y
merge(a,2,2,3)	O	W	A_1	N	R_1	R_2	A_2	Y
merge(a,0,1,3)	A_1	N	O	W	R_1	R_2	A_2	Y
merge(4,4,5)	A_1	N	O	W	R_1	R_2	A_2	Y
merge(6,6,7)	A_1	N	O	W	R_1	R_2	A_2	Y
merge(4,5,7)	A_1	N	O	W	A_2	R_1	R_2	Y
merge(0,3,7)	A_1	A_2	N	O	R_1	R_2	W	Y
	A_1	A_2	N	O	R_1	R_2	W	Y

anything in gray does not move

In this step, the first half of the array's two sub arrays ([O,W] and [A_1,N]) are being merged into one array that is sorted([A_1,N,O,W]). At this step's completion, this subarray of the whole array is sorted within itself but not within the whole array, and anything in the second half of the whole array is not yet sorted.

bottom-up:

a. bottom-up mergesort works similarly to top-down but differs slightly by sorting all sub arrays of the same size first, then moving up in subarray sizes in merging and sorting.

b. For any array, mergesort will take O(n log n) time because, regardless of how the array is currently sorted, it will still split the array in half recursively to sort everything

c.

	0	1	2	3	4	5	6	7
sz = 2	O	W	N	A_1	R_1	R_2	A_2	Y
merge(a,0,0,1)	O	W	N	A_1	R_1	R_2	A_2	Y
merge(a,2,2,3)	O	W	A_1	N	R_1	R_2	A_2	Y
merge(a,4,4,5)	O	W	A_1	N	R_1	R_2	A_2	Y
merge(a,6,6,7)	O	W	A_1	N	R_1	R_2	A_2	Y
sz = 4								
merge(a,0,1,3)	A_1	N	O	W	R_1	R_2	A_2	Y
merge(a,4,5,7)	A_1	N	O	W	A_2	R_1	R_2	Y
sz = 8								
merge(a,0,3,7)	A_1	A_2	N	O	R_1	R_2	W	Y

gray do not move at this step, each subarray of size 2 ([O,W], [A_1,N], [R_1,R_2], [A_2,Y]) is sorted within itself but not within any broader sub array or whole array.

1. quicksort

a. quicksort works by first randomly shuffling the array then partitioning the array, by scanning through the array from both ends, exchanging elements if they are not < the first element from left to right or > first element from right to left, repeating until the pointers cross, at which point the array's first element is moved so everything to the left of it is not greater than it, and everything to the right is not less than it. At that point, both subarrays can be sorted recursively.

b. best-case input: when each partition divides each array exactly in half
best-case time complexity: $O(n \log n)$
worst-case input: already sorted array
worst-case time: $O(n^2)$

c.

l	j	hi	0	1	2	3	4	5	6	7
			0	W	N	A_1	R_1	R_2	A_2	Y
	shuffle		0	Y	R_2	W	A_2	N	R_1	A_1
0	3	7	A_2	A_1	N	0	W	R_2	R_1	Y
0	1	2	A_1	A_2	N	0	W	R_2	R_1	Y
4	6	7	A_1	A_2	N	0	R_1	R_2	W	Y
			A_1	A_2	N	0	R_1	R_2	W	Y

gray does not move.

A_1 A_2 N 0 ↑ W R_2 R_1 Y

at this step, all elements to the left have been sorted and are in their final positions, while all elements to the right have yet to be sorted.

partition:

i	j	0	1	2	3	4	5	6	7
0-7		0	Y	R_2	W	A_2	N	R_1	A_1
1	7	0	A_1	R_2	W	A_2	N	R_1	Y
2	5	0	A_1	N	W	A_2	R_2	R_1	Y
3	4	0	A_1	N	A_2	N	R_2	R_1	Y
4	3	0	A_1	N	A_2	W	R_2	R_1	Y
0-2		A_2	A_1	N	0	W	R_2	R_1	Y
1	1	A_2	A_1	N					
4-7		A	A_2	N					
7	6							R_1	R_2 W Y

heapsort

a. heapsort works by first organizing the array into a heap, then pulling items out of the heap in decreasing order to generate the sorted result array

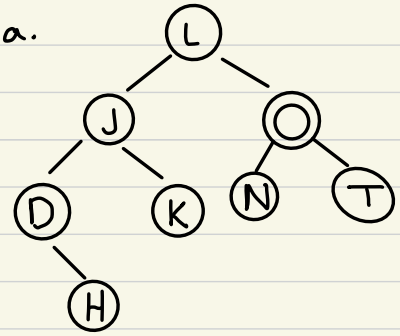
b. No matter the input, the heapsort algorithm will take $O(n \log n)$ time because it creates a heap of the whole array and repeatedly removes the maximum regardless of its initial sort status

c.

N	k	0	1	2	3	4	5	6	7
initial values		0	W	N	A_1	R_1	R_2	A_2	Y
7	3	0	W	N	Y	R_1	R_2	A_2	A_1
7	2	0	W	R_2	Y	R_1	N	A_2	A_1
7	1	0	Y	R_2	W	R_1	N	A_2	A_1
7	2	Y	W	R_2	0	R_1	N	A_2	A_1
heap ordered		Y	W	R_2	0	R_1	N	A_2	A_1
7	1	W	R_1	R_2	0	A_1	N	A_2	Y
6	1	R_2	R_1	N	0	A_1	A_2	W	Y
5	1	R_1	0	N	A_2	A_1	R_2	W	Y
4	1	0	A_2	N	A_1	R_1	R_2	W	Y
3	1	N	A_2	A_1	0	R_1	R_2	W	Y
2	1	A_2	A_1	N	0	R_1	R_2	W	Y
1	1	A_1	A_2	W	0	R_1	R_2	W	Y

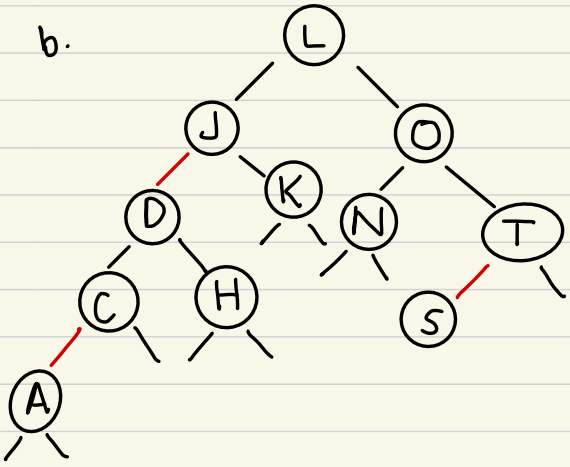
→ at this point, the second half of the array is sorted [R_1, R_2, W, Y] but the first half is not. Everything in the first half, however, is not greater than anything in the second part

2. a.



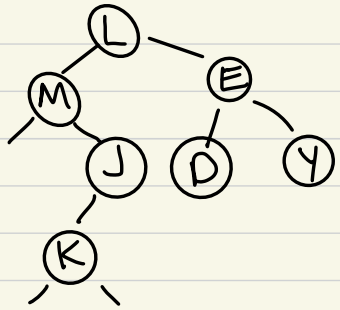
isBST() will return true for this tree because starting with the root of L: leftchild = J, rightchild = O & leftchild < root, rightchild > root, so okHere = true, and repeating that process down each side of the tree, all left childs are less than their parents, grandparents, and so on, up the tree, while the right childs are all greater than their parents, grandparents and so on up the tree, following the definition of a BST.

b.



isBST() will return true for this tree because starting with the root of L: leftchild = J, rightchild = O & leftchild < root, rightchild > root, so okHere = true, and repeating that process down each side of the tree, all left childs are less than their parents, grandparents, and so on, up the tree, while the right childs are all greater than their parents, grandparents and so on up the tree, following the definition of a BST.

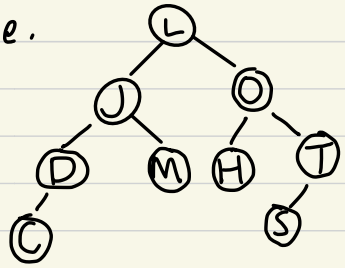
c.



isBST() will return false immediately because the left child is not less than the root & the right child is not greater than the root, so okHere will be false, making isBST() return false as it should

d. No such binary tree that isBST() returns false incorrectly for exists. While isBST() is sometimes faulty, it will always correctly recognize when a tree is a BST as it will recognize when a leftchild is null or less than it's parent and when a rightchild is null or greater than its parent and apply that to determine if it is a BST correctly.

e.



isBST() will incorrectly return true because it does not check that all grandchildren and on are still less than the root, it only checks if a parent and its two children make a valid BST. In the case of this tree, each parent and its tree make a valid BST on their own, however, in the case of M and H, they are on the wrong branches of the root L, so it is not a BST

3. a. word = "Aa" A = 65 a = 97

i = 0 word[i] = A

hash = 0

hash = (0 * 31) + charAt(0)

= 0 + 65

= 65

i = 1 word[i] = a

hash = 65

hash = (65 * 31) + charAt(1)

= 2015 + 97

= 2112

hash = 2112

b. word = "BB" B = 66

i = 0 word[i] = B

hash = 0

hash = (0 * 31) + charAt(0)

= 0 + 66

= 66

i = 1 word[i] = B

hash = 66

hash = (66 * 31) + 66

= 2046 + 66

= 2112

hash = 2112

```
public int hashCode() {
   int hash = 0;
   for (int i = 0; i < length(); i++)
      hash = (hash * 31) + charAt(i);
   return hash;
}
```

c. Aa BB BB Aa A aBB

BB BB BB BB BB BB

The hashcodes should be equal because the hashcodes of Aa and BB and both have equal length of 6 Aa/BBs

d. hash attack is the idea that there are multiple strings with the same hashcode, using substrings Aa and BB to build 2^N strings that have length 2N with the same hashvalue

4. a.

nodes = 119429
movies = 4188
actors = 115241
edges = 202927

Movies.txt separated each line out so each line was a different movie and the actors from that movie. To find the total number of nodes, I used the built in graph method V() to find the total number of verticies in the graph. Then to find the total number of movies I incremented a counter for each file line read. Because the graph nodes are all movies and actors, to find the number of actors, I subtracted the number of movies from the total number of nodes. Finally, to find the number of edges, I used the built in graph method E() that returns the total number of edges in the graph.

b. My bfs algorithm is similar to the previous dfs, however, I added a queue to add the verticies too. As long as the queue wasn't empty, the algorithm would remove the top vertex in the queue and go through that vertex's adjacents. As long as that vertex hadn't already been marked, it was added to the queue, marked, and a counter was incremented. For each adjacent, the corresponding element in the size array was set to the counter total.

Results:

number of connected components : 33

results in format of:

movie title int id cc.size(id)

Breaker' Morant (1980) 0 5
Aleksandr Nevskiy (1938) 5191 31
Babam Ve Oglum (2005) 13954 41
Bacheha-Ye aseman (1997) 14317 46
Bin-jip (2004) 20327 36
Bronenosets Potyomkin (1925) 25544 28
Buena Vista Social Club (1999) 26047 37
C'est arrivé près de chez vous (1992) 26783 36
Crumb (1994) 35827 35
Day of the Woman (1978) 37681 34
Dersu Uzala (1975) 38958 26
Filantropica (2002) 47928 32
Flåklypa Grand Prix (1975) 49278 5
For the Birds (2000) 49599 30
Gin gwai (2002) 53260 29
Hababam sinifi (1975) 56347 28
Idi i smotri (1985) 62208 27
Jetée, La (1962) 65755 26
Krámpack (2000) 69827 25
Manos: The Hands of Fate (1966) 76940 46
Murderball (2005) 82409 23
Mystery Science Theater 3000: The Movie (1996) 83349 12
Osama (2003) 87761 28
Primer (2004) 92651 19
Samaria (2004) 98879 15
Seom (2000) 100395 18
Touching the Void (2003) 111161 16
Undead (2003) 112798 16
Voyage dans la lune, Le (1902) 114570 15
Vozvrashcheniye (2003) 114580 14
Yi ge dou bu neng shao (1999) 118365 13
Yi yi (2000) 118390 39
Être et avoir (2002) 119378 8

5.

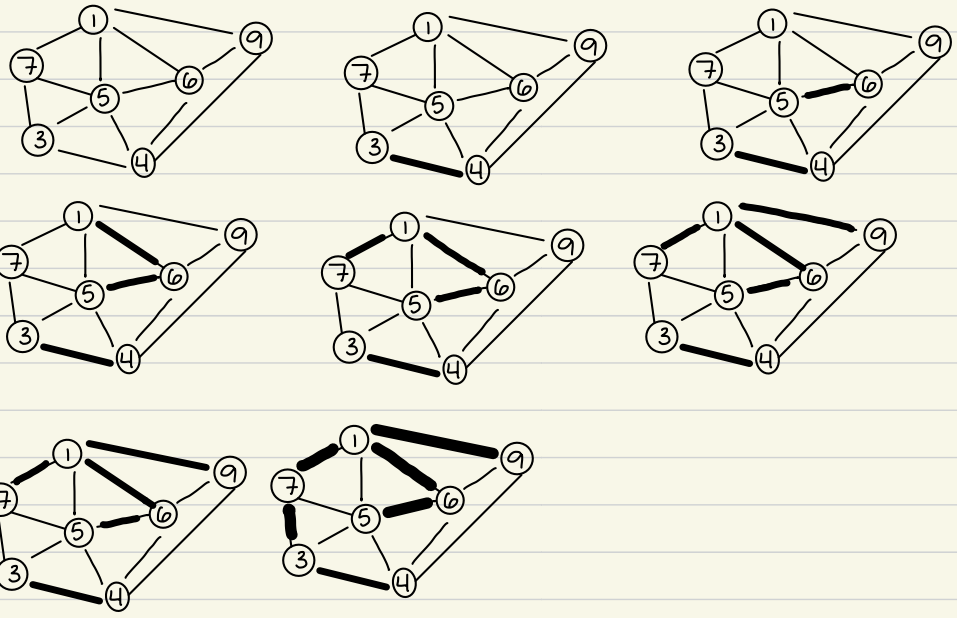
```
7
13
7  1  0.26
5  7  0.71
7  3  0.43
6  5  0.19
9  6  0.83
5  3  0.57
3  4  0.09
1  6  0.23
4  6  0.91
9  1  0.36
4  9  0.47
5  4  0.62
1  5  0.58
```

order by weight:

3-4 0.09
6-5 0.19
1-6 0.23
7-1 0.26
9-1 0.36
7-3 0.43
4-9 0.47 X cycle
5-3 0.57 X cycle
1-5 0.58 X cycle
5-4 0.62 X cycle
5-7 0.71 X cycle
9-6 0.83 X cycle
4-6 0.91 X cycle

Kruskal's algorithm:

Kruskal's algorithms process edges in weight order (smallest to largest), only adding edges that will not create a cycle out of the previously added edges in the MST, and ends after V-1 edges.

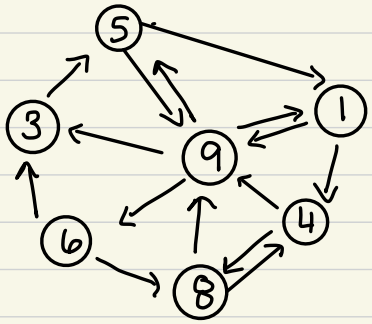


6.

1 → 4	0.23
1 → 9	0.32
5 → 1	0.13
5 → 9	0.41
3 → 5	0.24
6 → 8	0.39
4 → 8	0.17
8 → 4	0.17
8 → 9	0.42
4 → 9	0.37
9 → 6	0.29
9 → 3	0.21
9 → 1	0.32
9 → 5	0.41
6 → 3	0.09

Bellman-Ford:

the algorithm considers all edges in any order, and relaxes them all, testing and picking the best path between 2 vertices



pass 1 (s = 3)

v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

pass 2

v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

pass 3

v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

pass 4

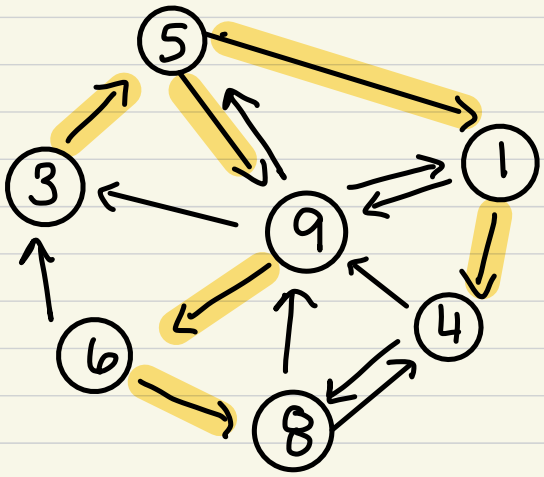
v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

pass 5

v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

pass 6

v	distTo[]	edgeTo[]
3	0.0	—
5	0.24	3 → 5
1	0.37	5 → 1
9	0.65	5 → 9
4	0.60	1 → 4
6	0.94	9 → 6
8	1.33	6 → 8

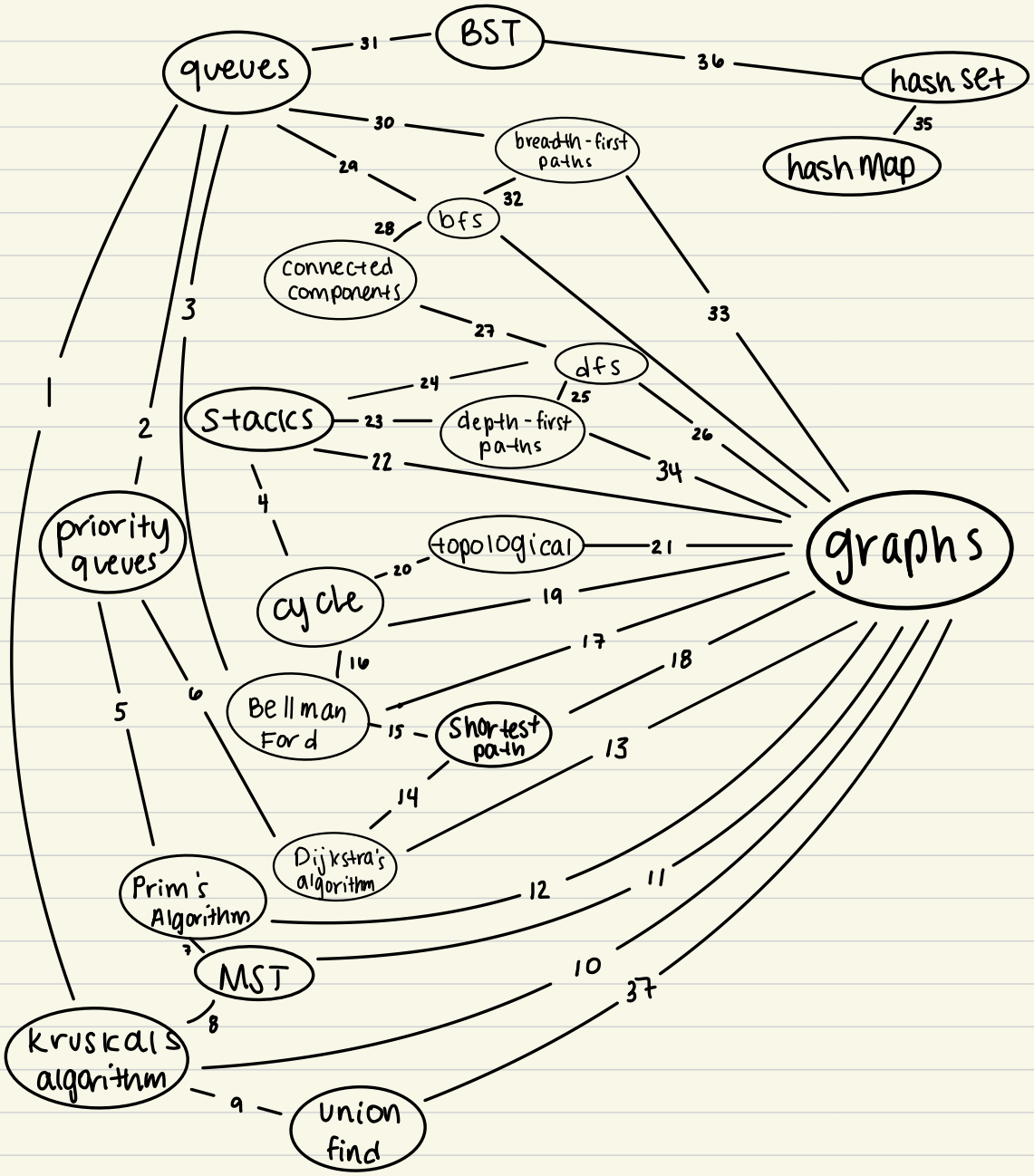


shortest path tree from s

7. A real world problem I am interested in that could be represented as a graph problem is cloud database management. For example, in business management, customer data and commerce data could be stored in two separate databases, and rather than combaning the two, a graph, focusing on connected components, could be used to keep track of the relationships between the databases. for example a customer's information could be connected to their personal commerce data in the other database.
In this example, customers as well as products could be the nodes, while the edges would represent purchases, connecting the customer nodes to the products they purchased, and other edges could represent connections between customers, if, for example, they came from the same industry. seeing the connected components could allow for the combination of 2 datasets to analyze the connections between customers and their purchases.

8. stacks, queues, union-find, priority queues - basic, MinPQ, MaxPQ, binary trees, hash tables, hash sets, graphs - undirected, directed, unweighted, weighted

algorithms: bfs, dfs, connected components, topological, cycle, directed cycle, MST, Prim's, Kruskal's, Dijkstra's, Bellman-ford



8.

Edge 1: queues are used as a supporting data structure in Kruskal's algorithm
Edge 2: priority queues are an extension of queues with assigned priorities (min or max or given)
Edge 3: queues are used as a supporting data structure in Bellman-Ford's algorithm
Edge 4: stacks are used as a supporting data structure in Cycle and by inference also directed cycle
Edge 5: minimum priority queues are used as a supporting data structure in Prim's algorithm
Edge 6: minimum priority queues are used as a supporting data structure in Dijkstra's algorithm
Edge 7: Prim's algorithm computes MSTs
Edge 8: Kruskal's algorithm computes MSTs
Edge 9: union find is used to check if a graph is acyclic in Kruskal's algorithm
Edge 10: edge weighted graphs, grouped in under graphs, are used as a supporting data structure to construct MSTs in Kruskal's algorithm
Edge 11: edge weighted graphs, grouped in under graphs, are used as a supporting data structure in MSTs
Edge 12: edge weighted graphs, grouped in under graphs, are used as a supporting data structure to construct MSTs in Prim's algorithm
Edge 13: edge weighted digraphs, grouped in under graphs, are used as a supporting data structure to construct shortest paths in Dijkstra's algorithm, as the algorithm is looking for shortest paths in a graph
Edge 14: Dijkstra's algorithm computes shortest paths
Edge 15: Bellman-Ford's algorithm computes shortest paths
Edge 16: Bellman-Ford's algorithm checks if a graph has negative cycles
Edge 17: edge weighted digraphs, grouped in under graphs, are used as a supporting data structure to construct shortest paths in Bellman-Ford's algorithm, as the algorithm is looking for shortest paths in a graph
Edge 18: shortest paths are found between given vertices in a graph
Edge 19: cycles are paths in graphs that start and end at the same vertex
Edge 20: directed cycles, and by inference cycles, are used as a supporting data structure to determine whether a digraph has a topological order
Edge 21: topological orders are found in digraphs, grouped in under graphs
Edge 22: stacks are used as a supporting data structure in graphs
Edge 23: stacks are used as a supporting data structure in depth first paths
Edge 24: stacks are used as a supporting data structure in dfs
Edge 25: dfs is used in depth-first paths
Edge 26: dfs finds what vertices are connected to a given vertex in graphs
Edge 27: dfs is used in connected components
Edge 28: bfs is used in connected components
Edge 29: queues are used as a supporting data structure in bfs
Edge 30: queues are used as a supporting data structure in breadth-first paths
Edge 31: queues are used as a supporting data structure in BST
Edge 32: bfs is used in breadth-first paths
Edge 33: breadth-first paths finds paths from a given vertex to other vertices in a graph
Edge 34: depth-first paths finds paths from a given vertex to other vertices in a graph
Edge 35: hashmaps are used as a supporting data structure in hashsets, and both use hashcode
Edge 36: BSTs can be implemented with hashtables, something that can be represented as a hashset, and both use keys
Edge 37: union find is used in Kruskal's algorithm to determine if a graph is acyclic