Hashing and Hash Tables

1. Suppose our hash function simply returns 17 every time it is called. Is such a hash function legal? If so, describe the effect of using it. If not, explain why.

Solution: Yes, such a function would be legal. However, it wouldn't be very useful because all elements would hash to the same index within an array. Effectively, every single insertion after the first would be a collision.

2. Describe an algorithm to delete an element in a hash table using linear probing.

Solution: Hash the key to find the index where the element should be. If that array index contains the element, set it to `null`. If the array index does not contain the element, continue searching using the linear probing algorithm. If a `null` element is found during the search, stop (the element is not in the hash table). Once the element is found, set it to `null`. Rehash any elements of the array which come after the now deleted element into the array. Stop rehashing once we reach a `null` entry.

3. Describe the algorithm used to delete an element in a hash table using separate chaining.

Solution: Hash the key to find the index of the linked list where the element should be. Linearly scan through the linked list, checking if the element's key matches the key we are trying to delete. If so, perform standard linked list deletion on the node which contains the element. Keep checking until we either find the element we are trying to delete or we reach the end of the list.

4. Insert the keys `E A S Y Q U T I O N` in that order into an initially empty table of $M = 5$ lists, using separate chaining. Use the hash function $11k\%M$ to transform the kth letter of the alphabet into a table index. You do not need to resize the hash table as a part of your solution. Show only the final hash table after all insertions have been completed.

Solution:

0	O → T → Y → E
1	U → A
2	Q
3	
4	N → I → S

5. Insert the keys `E A S Y Q U T I O N` in that order into an initially empty table of size $M = 16$ using linear probing. Use the hash function $11k\%M$ to transform the kth letter of the alphabet into a table index. You do not need to resize the hash table as a part of your solution. Show only the final hash table after all insertions have been completed.

Solution:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	S		Y	I	O		E	U		N	A	Q	T		

6 – 2	5 – 2	5 – 10
2 – 0	2 – 3	4 – 1
3 – 6	8 – 1	1 – 11
11 – 8	7 – 11	0 – 6
8 – 4	7 – 8	4 – 8
10 – 3	3 – 10	

Figure 1: A set of vertices connected within a graph

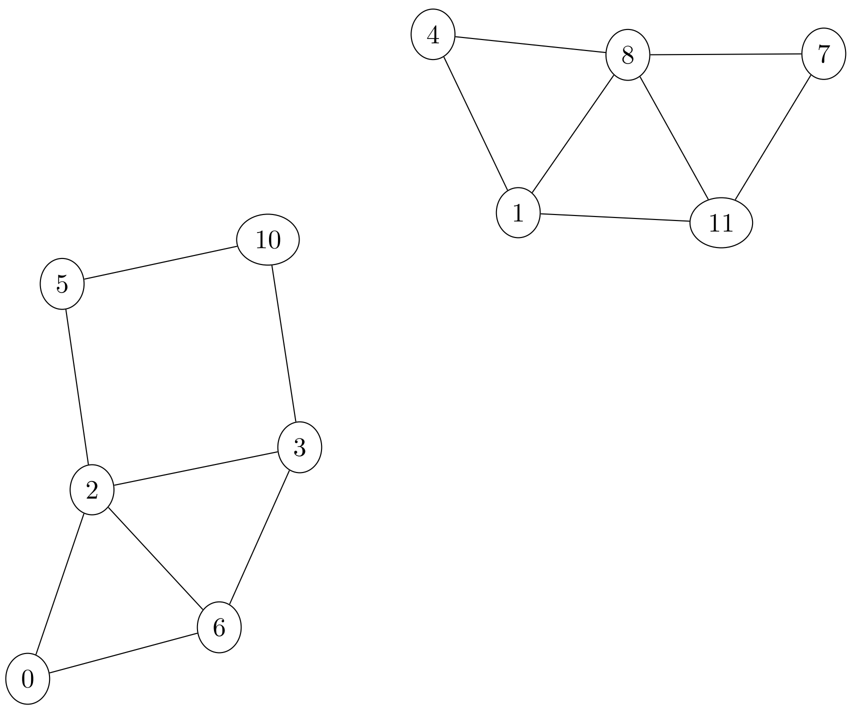
Undirected Graphs

6. When representing a graph in memory, when is it better to use an adjacency list? When is it better to use an adjacency matrix?

Solution: Adjacency lists are more space efficient for sparse graphs. However, adjacency matrices are more space efficient for dense graphs.

7. Given the list of vertices and edges in Figure 1, draw the resulting graph.

Solution:



8. Given the list of vertices and edges in Figure 1, write out the resulting adjacency list once the graph is loaded into memory.

Solution:

0	6 → 2
1	11 → 4 → 8
2	3 → 5 → 0 → 6
3	2 → 10 → 6
4	1 → 8
5	10 → 2
6	0 → 3 → 2
7	8 → 11
8	7 → 1 → 4 → 11
9	
10	5 → 3
11	1 → 7 → 8

9. Does breadth first search tell us anything about the distance from node v to node w when neither is at the root of the search?

Solution: Breadth first search can effectively give us the the distance between vertices v and w even when they are not at the root of the search. If v is an ancestor of w in the breadth first search (i.e. we pass through v while following the `edgeTo[]` links), or vice versa, the distance between v and w would be `abs(distTo[v] - distTo[w])`. If neither v is an ancestor of w nor w is an ancestor of v, we can still determine the distance between v and w. We first find a common ancestor on the way to the source node from both v and w. The distance between v and w is the sum of the distances to the common ancestor. In the worst case, the common ancestor would be the source node of the BFS, giving us a maximum distance of `distTo[v] + distTo[w]`.

10. A colleague suggests you to use a stack instead of a queue when running breadth first search. Would your colleague's suggestion still compute shortest paths in a non-edge-weighted undirected graph? Why or why not?

Solution: No, using stacks would no longer compute shortest paths. Using a stack would effectively convert the algorithm into depth first search. We know that depth first search does not compute shortest paths.