Rhea Mae Edwards
Student ID # 932-389-303

<u>Hash Tables and Graphs (Final Assignment)</u>

<u>Hash Tables</u>

From chapter 12, this is a description of Amy's hash function assuming an initial table size of 6:

"Amy uses an interesting fact. If she selects the third letter of each name, treating the letter as a number from 0 to 25, and then Mods the number by 6, each name yields a different number"

1. When Alan wishes to join the circle of six friends, why can't Amy simply increase the size of the table to seven?

Because by doing so, the indexes of the hashes by previously modding by 6 will become inconsistent with the table. Every element would need to be rehashed is Amy decides to change the size of the table, she just cannot simply increase the size.

2. Amy's club has grown, and now includes the following members:

Abel Abigail Abraham Ada

Adam Adrian Adrienne Agnes

Albert Alex Alfred Alice

a. Find what value would be computed by Amy's hash function for each member of the group, BEFORE modding by the table size.

Abel	**e = 4**	**Adrienne**	**r = 17**
Abigail	**i = 8**	**Agnes**	**n = 13**
Abraham	**r = 17**	**Albert**	**b = 1**
Ada	**a = 0**	**Alex**	**e = 4**
Adam	**a = 0**	**Alfred**	**f = 5**
Adrian	**r = 17**	**Alice**	**i = 8**

b. Now, assume we use Amy's hash function and assign each member to a bucket by simply modding the hash value (obtained from part a) by the number of buckets. Determine how many elements would be assigned to each bucket (assume hashing with chaining) for a hash table of size 6. Do the same for a hash table of size 13.

Abel	**e = 4 % 6 = 4**
Abigail	**i = 8 % 6 = 2**
Abraham	**r = 17 % 6 = 5**
Ada	**a = 0 % 6 = 0**
Adam	**a = 0 % 6 = 0**
Adrian	**r = 17 % 6 = 5**
Adrienne	**r = 17 % 6 = 5**
Agnes	**n = 13 % 6 = 1**
Albert	**b = 1 % 6 = 1**
Alex	**e = 4 % 6 = 4**
Alfred	**f = 5 % 6 = 5**
Alice	**i = 8 % 6 = 2**

Along with the elements given in the example or the people who initially were in the group (the original first six people Amy, Anne, Amina, Andy, Alessia, Alfred), each bucket will hold…

Bucket 0: 3 Elements
Bucket 1: 3 Elements
Bucket 2: 3 Elements
Bucket 3: 1 Element
Bucket 4: 3 Elements
Bucket 5: 5 Elements

Rhea Mae Edwards
Student ID # 932-389-303

Abel	**e = 4 % 13 = 4**
Abigail	**i = 8 % 13 = 8**
Abraham	**r = 17 % 13 = 4**
Ada	**a = 0 % 13 = 0**
Adam	**a = 0 % 13 = 0**
Adrian	**r = 17 % 13 = 4**
Adrienne	**r = 17 % 13 = 4**
Agnes	**n = 13 % 13 = 0**
Albert	**b = 1 % 13 = 1**
Alex	**e = 4 % 13 = 4**
Alfred	**f = 5 % 13 = 5**
Alice	**i = 8 % 13 = 8**

Alfred	**f = 5 % 13 = 5**
Alessia	**e = 4 % 13 = 4**
Amina	**i = 8 % 13 = 8**
Amy	**y = 24 % 13 = 11**
Andy	**d = 3 % 13 = 3**
Anne	**n = 13 % 13 = 8**

Along with the elements given in the example or the people who initially were in the group (the original first six people Amy, Anne, Amina, Andy, Alessia, Alfred), each bucket will hold…

Bucket 0: 3 Elements
Bucket 1: 1 Element
Bucket 2: 0 Elements
Bucket 3: 1 Element
Bucket 4: 6 Elements
Bucket 5: 2 Elements
Bucket 6: 0 Elements
Bucket 7: 0 Elements
Bucket 8: 4 Elements
Bucket 9: 0 Elements
Bucket 10: 0 Elements
Bucket 11: 1 Element
Bucket 12: 0 Elements

c. What are the load factors of these two tables?

Load Factor = Number of Elements / Capacity of Table

Table (Size 6) Load Factor: 18 / 6 = 3

Table (Size 13) Load Factor: 18 / 13 = 1.38

3. In searching for a good hash function over the set of integer values, one student thought he could use the following:

int index = (int) cos(value); // Cosine of 'value'

What was wrong with this choice?

Before even modding, if the student even decides to mod, every given integer of the set will be either given the value of -1, 0, or 1. There is no other possible value that can be given, since the cosine function only outputs values between -1 and 1 no matter how small or large the integer value is given. Also you can't have a negative index value in a table if the student doesn't decide to mod afterwards.

Rhea Mae Edwards
Student ID # 932-389-303

4. Can you come up with a perfect hash function for the names of days of the week? The names of the months of the year? Assume a table size of 10 for days of the week and 15 for names of the months. In case you cannot find any perfect hash functions, we will accept solutions that produce a small number of collisions (<3).

Hash Function for the Names of Days of the Week:

Add the values of the first letter and the third letter of the names (0-25) then modding by 10.
There are two collisions with two different indexes. I couldn't figure out any perfect hash functions…

Sunday	**s + n = (18 + 13) = 31 % 10 = 1**
Monday	**m + n = (12 + 13) = 25 % 10 = 5**
Tuesday	**t + e = (19 + 4) = 23 % 10 = 3**
Wednesday	**w + d = (22 + 3) = 25 % 10 = 5**
Thursday	**t + u = (19 + 20) = 39 % 10 = 9**
Friday	**f + i = (5 + 8) = 13 % 10 = 3**
Saturday	**s + t = (18 + 19) = 37 % 10 = 7**

Hash Function for the Names of the Months:

Add the values of the first letter and the last letter of the names (0-25) then modding by 15.
There are three collisions with three different indexes. I couldn't figure out any perfect hash functions…

January	**j + y = (9 + 24) = 33 % 15 = 3**
February	**f + y = (5 + 24) = 29 % 15 = 14**
March	**m + h = (12 + 7) = 19 % 15 = 4**
April	**a + l = (0 + 11) = 11 % 15 = 11**
May	**m + y = (12 + 24) = 36 % 15 = 6**
June	**j + e = (9 + 4) = 13 % 15 = 13**
July	**j + y = (9 + 24) = 33 % 15 = 3**
August	**a + t = (0 + 19) = 19 % 15 = 4**
September	**s + r = (18 + 17) = 35 % 15 = 5**
October	**o + r = (14 + 17) = 31 % 15 = 1**
November	**n + r = (13 + 17) = 30 % 15 = 0**
December	**d + r = (3 + 17) = 20 % 15 = 5**

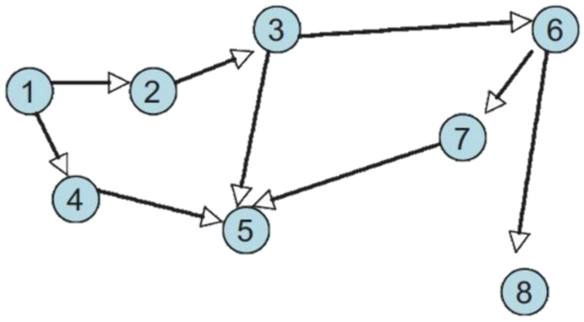
5. The function containsKey() can be used to see if a dictionary contains a given key. How could you determine if a dictionary contains a given value? What is the complexity of your procedure?

To determine is a given value is contained in a dictionary is to first put the given value through the same hash function that is associated with the dictionary. Then look through that location/index to see if that currently key in the dictionary matches the value in question. And for as long as the key location doesn't equal null, search through the rest of the keys in that bucket/index of the dictionary/table to see if a given value is contained in the dictionary.
The complexity of this procedure is O(the number of elements contained in that hashed to bucket), or on average O(load factor of the table of the dictionary).

Rhea Mae Edwards
Student ID # 932-389-303

<u>Graphs</u>

Describe the following graph as both an adjacency matrix and an edge list:



Adjacency Matrix:

Numbers	1	2	3	4	5	6	7	8
1	?	1	0	1	0	0	0	0
2	0	?	1	0	0	0	0	0
3	0	0	?	0	1	1	0	0
4	0	0	0	?	1	0	0	0
5	0	0	0	0	?	0	0	0
6	0	0	0	0	0	?	1	1
7	0	0	0	0	1	0	?	0
8	0	0	0	0	0	0	0	?

Edge List:

1: {2, 4}

2: {3}

3: {5, 6}

4: {5}

5: { }

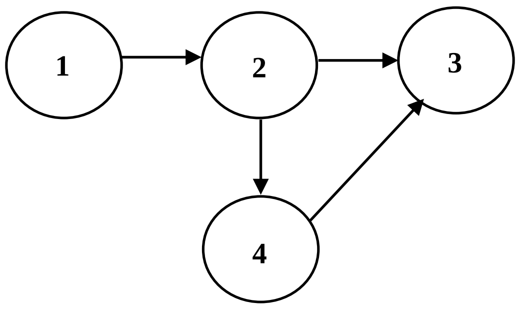
6: {7, 8}

7: {5}

8: { }

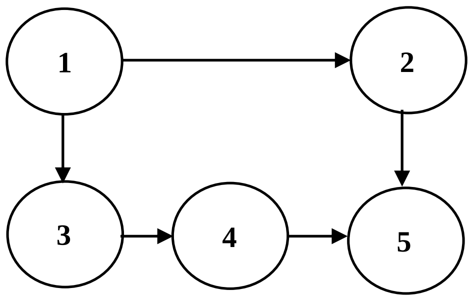
Construct a graph in which a depth-first search will uncover a solution (discover reachability from one vertex to another) in fewer steps than will a breadth first search. You may need to specify an order in which neighbor vertices are visited. Construct another graph in which a breadth-first search will uncover a solution in fewer steps.

Fewer Steps with Depth-First Search:



From 1 (Start) to 4 (Solution)

Fewer Steps with Breadth-First Search:



From 1 (Start) to 5 (Solution)

Rhea Mae Edwards
Student ID # 932-389-303

Complete Worksheet 41 (2 simulations). Show the content of the stack, queue, and the set of reachable nodes.

Depth First Search (Stack Version)

Breadth First Search (Queue Version)

Stack:

Iteration	Known to be Reachable (c) (Top → Bottom)	Reachable Vertices (r)
0	1	
1	6, 2	1
2	11, 2	1, 6
3	16, 12, 2	1, 6, 11
4	21, 12, 2	1, 6, 11, 16
5	22, 12, 2	1, 6, 11, 16, 21
6	23, 17, 12, 2	1, 6, 11, 16, 21, 22
7	17, 12, 2	1, 6, 11, 16, 21, 22, 23
8	12, 12, 2	1, 6, 11, 16, 21, 22, 23, 17
9	13, 12, 2	1, 6, 11, 16, 21, 22, 23, 17, 12
10	18, 8, 12, 2	1, 6, 11, 16, 21, 22, 23, 17, 12, 13
11	8, 12, 2	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18
12	3, 12, 2	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8
13	12, 2, 4	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3
14	7, 4	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2
15	4	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7
16	9, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4
17	14, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9
18	15, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14
19	20, 10, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15
20	19, 10, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20
21	24, 10, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19
22	25, 10, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19, 24
23	10, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19, 24, 25
24	5, 5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19, 24, 25, 10
25	5	1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19, 24, 25, 10, 5
26		1, 6, 11, 16, 21, 22, 23, 17, 12, 13, 18, 8, 3, 2, 7, 4, 9, 14, 15, 20, 19, 24, 25, 10, 5

Rhea Mae Edwards
Student ID # 932-389-303

Queue:

Iteration	Known to be Reachable (c) (Front → Back)	Reachable Vertices (r)
0	1	
1	2, 6	1
2	6, 3, 7	1, 2
3	3, 7, 11	1, 2, 6
4	7, 11, 4, 8	1, 2, 6, 3
5	11, 4, 8	1, 2, 6, 3, 7
6	4, 8, 12, 16	1, 2, 6, 3, 7, 11
7	8, 12, 16, 5, 9	1, 2, 6, 3, 7, 11, 4
8	12, 16, 5, 9, 13	1, 2, 6, 3, 7, 11, 4, 8
9	16, 5, 9, 13, 13, 17	1, 2, 6, 3, 7, 11, 4, 8, 12
10	5, 9, 13, 13, 17	1, 2, 6, 3, 7, 11, 4, 8, 12, 16
11	9, 13, 13, 17, 10	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5
12	13, 13, 17, 10, 14	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9
13	13, 17, 10, 14, 18	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13
14	10, 14, 18, 22	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17
15	14, 18, 22, 15	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10
16	18, 22, 15	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14
17	22, 15	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18
18	15, 23, 21	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22
19	23, 21, 20	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15
20	21, 20	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23
21	20	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23, 21
22	19	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23, 21, 20
23	24	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23, 21, 20, 19
24	25	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23, 21, 20, 19, 24
25		1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13, 17, 10, 14, 18, 22, 15, 23, 21, 20, 19, 24, 25

Complete Worksheet 42 (1 simulation). Show the content of the priority queue and the cities visited at each step.

Iteration	Reachable	Priority Queue
0		Pensacola: 0
1	Pensacola: 0	Phoenix: 5
2	Phoenix: 5	Pueblo: 8, Peoria: 9, Pittsburgh: 15
3	Pueblo: 8	Peoria: 9, Pierre: 11, Pittsburgh: 15
4	Peoria: 9	Pierre: 11, Pueblo: 12, Pittsburgh: 14, Pittsburgh: 15
5	Pierre: 11	Pueblo: 12, Pendleton: 13, Pittsburgh: 14, Pittsburgh: 15
6	Pendleton: 13	Pittsburgh: 14, Pittsburgh: 15, Phoenix: 17, Pueblo: 21
7	Pittsburgh: 14	Pittsburgh: 15, Phoenix: 17, Pensacola: 18, Pueblo: 21
8		

Rhea Mae Edwards
Student ID # 932-389-303

Why is it important that Dijkstra's algorithm stores intermediate results in a priority queue, rather than in an ordinary stack or queue?

Because Dijkstra’s algorithm is a cost-first search, and the values stored along with the name of the element matters of which intermediate result is stored in the priority queue, whereas in an ordinary stack or queue that importance is not taken into account and just depends on which node was reached first, last, etc., not really taking into account the costs/weighs given by a graph.

How much space (in big-O notation) does an edge-list representation of a graph require?

O(V + E) space where V is the number of vertices and E is the number of edges of the graph.

For a graph with V vertices, how much space (in big-O notation) will an adjacency matrix require?

$O(V^2)$ space

Suppose you have a graph representing a maze that is infinite in size, but there is a finite path from the start to the finish. Is a depth-first search guaranteed to find the path? Is a breadth-first search? Explain why or why not.

**Neither search will guarantee a path for a finite solution of an infinite sized maze, because one way can be the solution for the finite path of the maze, but the opposing path could’ve been taken/chosen instead.
For example, the solution could’ve been found by the use of a depth-first search but the breadth-first search was taken/chosen instead, and vice versa.**