Kevin Sliker
December 4, 2015
CS261 Data Structures: Assignment 7

Hash Tables

1. Amy cannot simply increase the table size to 7 because the modulo in her algorithm would not return unique values given the third letter in each persons name. Specifically, the names Amy and Andy, with third letter values 24 and 3, respectively, modulo 7 gives the same remainder of 3.

2. (a) Each name passed through Amy's hash function before modding:

$$
\begin{aligned}
Abel - 4\\
Abigail - 8\\
Abraham - 17\\
Ada - 0\\
Adam - 0\\
Adrian - 17\\
Adrienne - 17\\
Agnes - 13\\
Albert - 1\\
Alex - 4\\
Alfred - 5\\
Alice - 8.
\end{aligned}
$$

(b) Elements found in each bucket after being hashed:

Table Size == 6	Table Size == 13
$table[0] = 2$	$table[0] = 3$
$table[1] = 2$	$table[1] = 1$
$table[2] = 2$	$table[2] = 0$
$table[3] = 0$	$table[3] = 0$
$table[4] = 2$	$table[4] = 5$
$table[5] = 4$	$table[5] = 1$
	$table[6] = 0$
	$table[7] = 0$
	$table[8] = 2$
	$table[9] = 0$
	$table[10] = 0$
	$table[11] = 0$
	$table[12] = 0$

(c) The load factors for a tablesize of 6 and 13, are 2 and .92, respectively.

3. Using $cos(val)$ as a hash function over the integer values is a bad idea because the cosine function is sinusoidal. Therefore, the values -1 and 1 (both proper integer values) both return $\approx$ .999848.

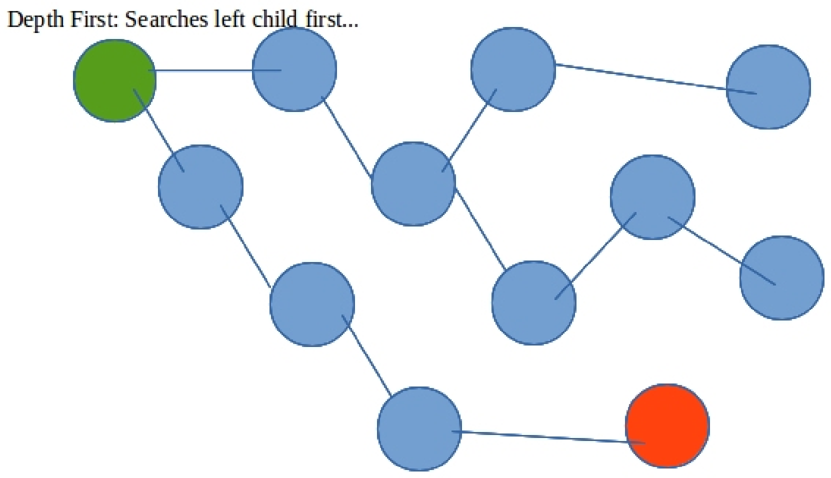
4. (a) Days of the week hash function with a table size of 10: Assign each letter of the word a value between 0-25, add up all of the letters values, divide that sum by the number of letters in the word, then mod that quotient by 10. This algorithm results in a near perfect hash with only 2 collisions.

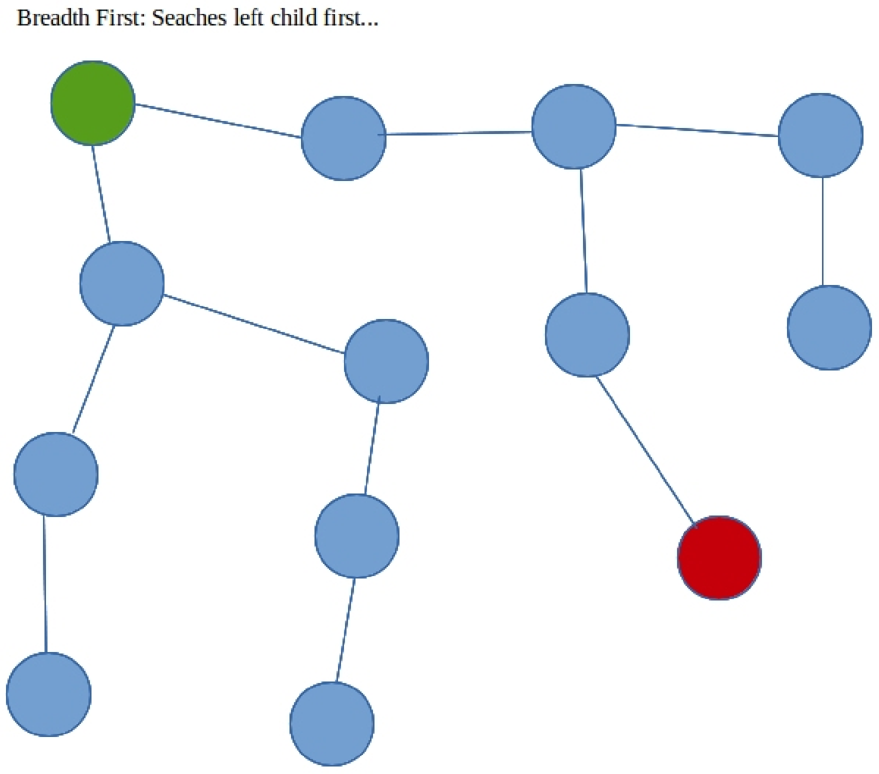
   (b) Months of the year hash function with a table size of 15:

5. One way to search for a given value would be to keep the buckets in a dynamic array but make the links or chaining a balanced binary tree. The reason for using a tree instead of a linked list would be to quickly search through our chains. We know that if the load factor is set correctly we could achieve a best case $O(1)$ time complexity for finding keys. We also know that we can achieve a $O(logn)$ time complexity for a balanced binary tree. So I would suggest using an AVL tree instead of a linked list to do our chaining by either ranking the values in the tree based on their character values, time they were entered in the tree, or some other distinguishing factor.

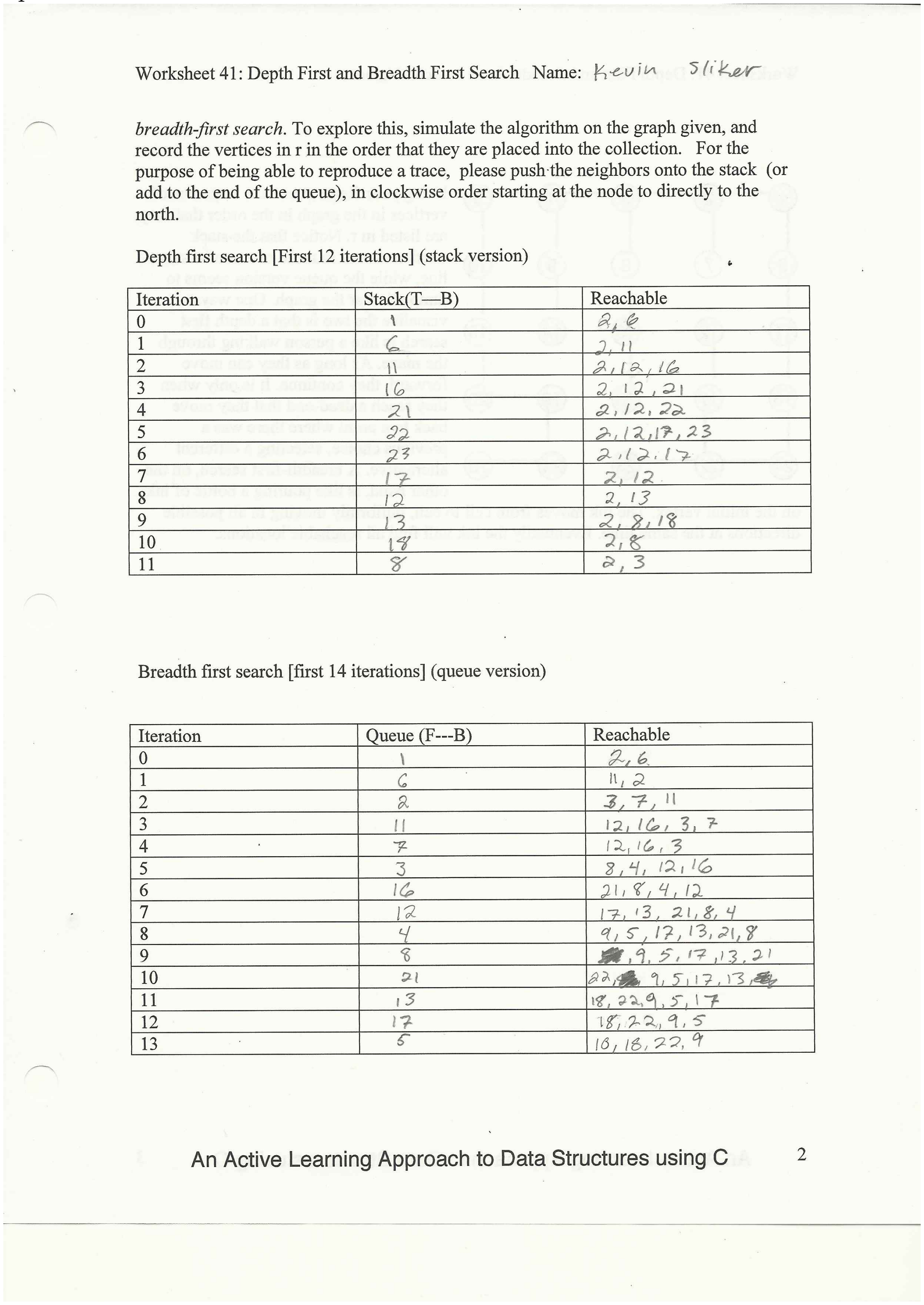
Graphs

Construct a graph in which a depth first search will uncover a solution in fewer steps that will a breadth first search.





Complete worksheet 41



Worksheet 41: Depth First and Breadth First Search Name: Kevin Sliker

breadth-first search. To explore this, simulate the algorithm on the graph given, and record the vertices in r in the order that they are placed into the collection. For the purpose of being able to reproduce a trace, please push the neighbors onto the stack (or add to the end of the queue), in clockwise order starting at the node to directly to the north.

Depth first search [First 12 iterations] (stack version)

Iteration	Stack(T—B)	Reachable
0	1	2, 6
1	6	2, 11
2	11	2, 12, 16
3	16	2, 12, 21
4	21	2, 12, 22
5	22	2, 12, 17, 23
6	23	2, 12, 17
7	17	2, 12
8	12	2, 13
9	13	2, 8, 18
10	18	2, 8
11	8	2, 3

Breadth first search [first 14 iterations] (queue version)

Iteration	Queue (F---B)	Reachable
0	1	2, 6
1	6	11, 2
2	2	3, 7, 11
3	11	12, 16, 3, 7
4	7	12, 16, 3
5	3	8, 4, 12, 16
6	16	21, 8, 4, 12
7	12	17, 13, 21, 8, 4
8	4	9, 5, 17, 13, 21, 8
9	8	[illegible], 9, 5, 17, 13, 21
10	21	22, [illegible], 9, 5, 17, 13, [illegible]
11	13	18, 22, 9, 5, 17
12	17	18, 22, 9, 5
13	5	10, 18, 22, 9

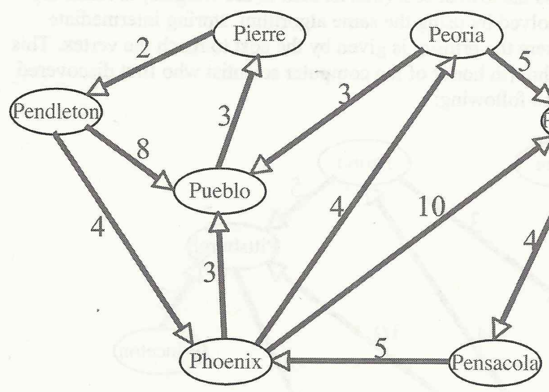
Complete worksheet 42

Worksheet 42: Dijkstra's Algorithm Name: Kevin Sliker

6	Pittsburgh:15, Pittsburgh: 16	--
7	Pittsburgh: 16, Pensacola: 19	Pittsburgh: 15
8	Pensacold:19	--.
9	{}	Pensacola: 19

Notice how duplicates are removed only when pulled from the pqueue.

Simulate Dijkstra's algorithm, only this time using Pensacola as the starting city:



Reachable	Costs
Pensacola	5 - Phoenix
Phoenix: 5	Pueblo: 8, Peoria: 9, Pittsburgh: 15
Pueblo: 8	Pierre: 11, Peoria: 9, Pittsburgh: 15
Peoria: 9	Pueblo: 12, Pittsburg: 14 Pierre: 11
~~Pittsburg: 14~~	~~Pensacola: 18~~
Pierre: 11	Pendelton: 13, Pittsburg: 14
Pendleton: 13	~~Pueblo~~: 21, Pheonix: 17

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

The importance in storing the intermediate results is that dijkstra's algorithm doesn't just want to find a path. It wants to find the path with the lowest cost. Therefore, when the algorithm is looking for potential paths it must hold the cost associated with each possible path in a priority to keep a preference. The algorithm is particular in which path it takes and the priority queue gives preference to those possible paths. The stack or the queue only give a "backtracked" possible path, which is not necessarily a path of lowest cost.

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

The edge-list would require $O(E)$ space for any number of edges - E.

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

The space for an adjacency matrix is $O(V^2)$ for any number of vertices - V.

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.

The depth first search is not guaranteed to find the path to finish. The breadth first search is guaranteed to find a path to finish. The reason for this is due to the fact that if the depth first search chose a path that was infinite, it would never "backtrack" to another path choice. That being said, it would carry out it's algorithm indefinitely. Now the depth first search would indeed find a path to finish even if there were an arbitrary amount of infinite paths as well. This method would search every possible path both the infinite ones and the finite ones all at the same time ensuring a completion in finite time.