Mihai Dan and Marc Ohlmann
12/4/2014
CS261-002
Assignment 7

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

There are a few problems with Amy simply adding a new slot for Alan to hash into. In order to add values past the capacity of a hash table, a new table has to be created with new index values creating by modding by the new capacity. Then, all the values will be hashed into the new has table, evenly, assuming the hash function was done well. At this point, Alan has enough room to be indexed to a new slot and the load capacity also stays below the desired 75%.

2.) <u>Amy's club has grown…</u>

a. Value for each key BEFORE rehashing:

Abel: 4	Abigal: 0	Abraham: 1	Ada: 0
Adam: 0	Adrian: 1	Adrienne: 1	Agnes: 5
Albert: 1	Alex: 4	Alfred: 5	Alice: 0

b. <u>Chain Hashing with 6 Spots:</u>
Spot 0: Alan, Ada, Adam (3)
Spot 1: Agnes, Anne (2)
Spot 2: Amina, Abigail, Alice (3)
Spot 3: Andy, Aspen (2)
Spot 4: Alessia, Abel, Alex (3)
Spot 5: Alfred, Abraham, Adrian (3)
c. <u>Load Factor:</u>
16 keys/ 6 spots = 2.667 keys/Spot

b. <u>Chain Hashing with 13 Spots:</u>
Spot 0: Alan, Ada, Adam, Agnes, Anne (5)
Spot 1: (0)
Spot 2: Aspen (1)
Spot 3: Andy (1)
Spot 4: Alessia, Abel, Alex, Abraham, Adrian (5)
Spot 5: Alfred (1)
Spot 6: (0)
Spot 7: (0)
Spot 8: Amina, Abigail, Alice (3)
Spot 9: (0)
Spot 10: (0)
Spot 11: (0)
Spot 12: (0)
c. <u>Load Factor:</u>
16 keys/ 13 spots = 1.23 keys/spots

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?

There are some major issues with using cosine as a hash function. First of all, Cos(x) can return -1 which is not a valid index number. Hash functions are meant to return index values so hash functions should only return a valid set of index values. Secondly, Cos(x) is a periodic function which means it repeats its values for every 2π interval.

$$\text{Cos}(x) = \text{Cos}(x + 2\pi n) \text{ for any integer } n$$

Cos(x) returns all real values between -1 and 1, in fact, the only time Cos(x) will return an integer is for the following inputs (between -2π and 2π):

x	Cos(x)
- π, π	-1
0, 2π, -2π	1
-π/2, π/2	0

When typecasting Cos(x) to an integer there are only 3 possible return values (-1, 1, 0). Since typecasting a real number to an integer truncates the fractional portion of the real value, and since Cos(x) cannot return a value larger than 1 or less than -1, this hash function will only be able to return zero unless the value happens to be – π, π, 0, or 2π **exactly**. If value is just slightly less than π, then Cos(value) will return a value less than 1, which will be truncated down to zero. This means it is very unlikely that this hash function will ever return anything but zero unless value is a multiple of π.

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).

a. Table for Days of the Week:

I assumed a standard week starts with Sunday. To implement my hashing index, I divided the table into 10 different spots and used Ami's algorithm to find out how many letters of the alphabet I can distribute to each spot (3 for 0-5 and 2 for 6-9), and then filled them going through the alphabet starting with the letter a. Each day is hashed into the table by using the second letter (i.e. Wednesday = 'e'). The table looks as follows:

Table size: 10
Table Load: .7
Spot 0 (abc): Saturday
Spot 1 (def): Wednesday
Spot 2 (ghi): Thursday
Spot 3 (jkl):
Spot 4 (mno): Monday
Spot 5 (pqr): Friday

Spot 6 (st):
Spot 7 (uv): Sunday, ~~Tuesday~~
Spot 8 (wx): Tuesday
Spot 9 (yz):
When hashing ‘Tuesday’ into the function, it returned the same index as ‘Sunday’ causing a collision between the two, so ‘Tuesday’ was probed to the next Spot, Spot 8.

b. Table for Months of the Year:

I assumed a standard year starts with January. To implement the hashing index, I divided the table into 15 different spots and hashed the different letters of the alphabet iterating through the table and filling every other with a letter. The letters are hashed by their third letter because that is the one the hashed the best for months of the year. The table looks as follows:
Table size: 15
Table Load: .8
Spot 0 (ap): September
Spot 1 (ix):
Spot 2 (bq): February
Spot 3 (jy): May
Spot 4 (cr): March, ~~April~~, ~~December~~
Spot 5 (kz): April
Spot 6 (ds): December
Spot 7 (l): July
Spot 8 (et): October
Spot 9 (m):
Spot 10 (fv): November
Spot 11 (n): January, ~~June~~
Spot 12 (gv): June, ~~August~~
Spot 13 (o): August
Spot 14 (nw):
I was not able to construct a hashing index that yielded less than three collisions; the best I was able to do was a hashing index that leads to this table with only four collisions. The probing is shown in red on the table above.

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?

In a dictionary, there is a simple procedure to access a value if a key is entered. The key is entered and hashed out producing a given index value. The index value accesses the index at said point and checks whether or not there is a value within. If there is a value, it returns it, if not it will simply return 0;. The complexity of this procedure is O(1) because you are only required to access only memory point instead of searching through a whole array.

6.) Describe the graph as both an adjacency matrix and an edge list:

Adjacency Matrix:

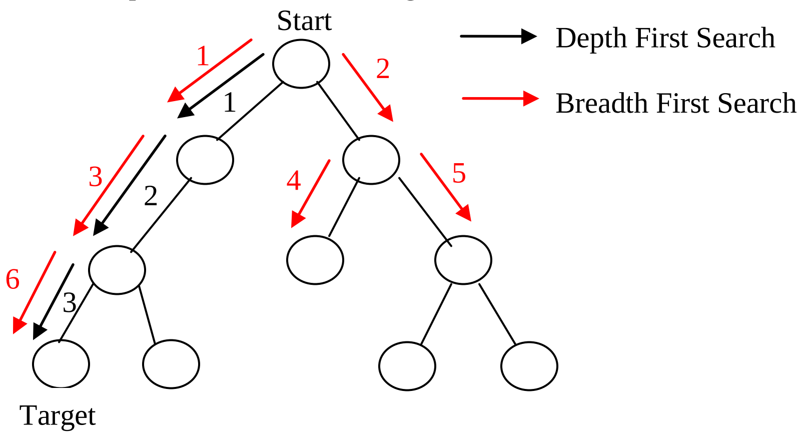
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	?	1	0	1	0	0	0	0
2	0	?	1	0	0	0	0	0
3	0	0	?	0	1	1	0	0
4	0	0	0	?	1	0	0	0
5	0	0	0	0	?	0	0	0
6	0	0	0	0	0	?	1	1
7	0	0	0	0	1	0	?	0
8	0	0	0	0	0	0	0	?

Edge List:

1: {2, 4}
2: {3}
3: {5, 6}
4: {5}
5: {}
6: {7, 8}
7: {5}
8: {}

7.) 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.

a. Depth First Search yielding faster results:

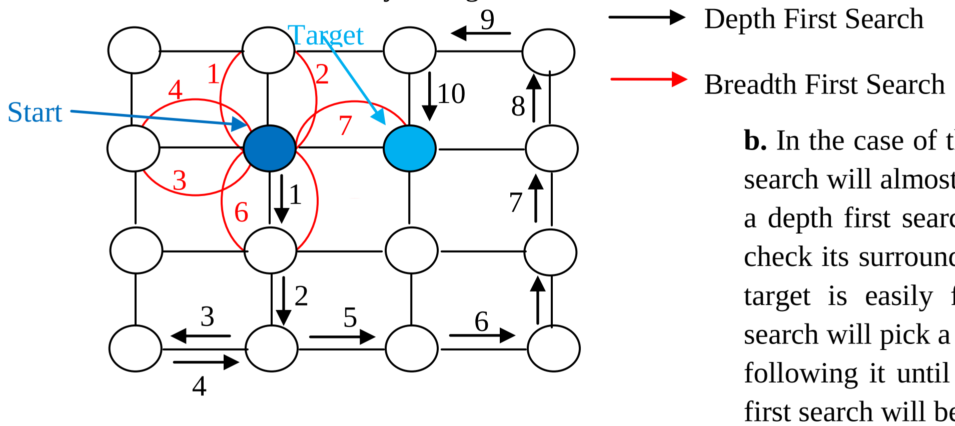


a. As shown by the arrows, this graph represents a graph where depth first search will find the result faster than breadth first search. Breadth search first will search every possible location as it ascends through the levels, whereas depth first search will go straight to the target.

Total steps: 3

Total steps: 6

b. Breadth First Search yielding faster results:



b. In the case of this kind of graph, a breadth first search will almost always yield a result faster than a depth first search. The breadth first search will check its surroundings before proceeding, and the target is easily found that way. A depth first search will pick a path and try to find the target by following it until the end. In this case, a breadth first search will be more efficient.

Total steps: 10

Total steps: 7

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

a. Depth First Search [First 12 Iterations]:

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

b. Breadth First Search [First 14 Iterations]:

Iteration	**Queue (F—B)**	**Reachable**
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, 21	1, 2, 6, 3, 7, 11, 4, 8, 12, 16
11	9, 13, 13, 17, 21, 10	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5
12	13, 13, 17, 21, 10, 14	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9
13	13, 17, 21, 10, 14, 18	1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 9, 13

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

Iteration	Priority Queue	Reachable with Cost
0	Pensacola(0)	{}
1	Pheonix(5)	Pensacola(0)
2	Pueblo(8), Peoria(9), Pittsburg(15)	Pheonix(5)
3	Peoria(9), Pierre(11), Pittsburg(15)	Pueblo(8)
4	Pierre(11), Pittsburg(14), Pittburrg(15)	Peoria(9)
5	Pendleton(13), Pittsburg(14), Pittsburg(15)	Pierre(11)
6	Pittsburg(14), Pittsburg(15)	Pendleton(13)
7	Pittsburg(15)	Pittsburg(14)
8	{}	--

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

It’s important that Dijkstra’s algorithm stores intermediate results in a priority queues simply because a shorter path will be placed earlier in the queue and subsequently be removed from the queue sooner. Unlike using a stack, which works on a first in first out basis, a priority queue takes into account which item is more important, moving it to the front of the queue, meaning it will be removed sooner.

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

An edge-list representation of a graph requires O(n+e) space.

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

The adjacency matrix representation always requires $O(v^2)$ space to store a matrix with v vertices, regardless of the number of arcs.

13.) 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.

Depth first search will pick a route and follow it until it finds a dead end. This means if the maze is infinitely long, depth first search may never find a dead end, so if it doesn’t go down the correct path it will get stuck in an infinite loop and never find the path. Breadth first search on the other hand slowly creeps out along all adjacent paths almost simultaneously. Breadth first search will eventually be able to find the path, even though it may take a very long time. As long as the distance from the start to the end point is finite, breadth first search will be able to find a path even if the size of the maze is infinite, while depth first search has a chance to get stuck in an infinite loop.