David Merrick

CS 261, Winter 2012

3/14/12

# Assignment 7

1.

```
//Dynamic array implementation
//Function that removes the key before inserting new one.

void putMap (struct hashMap * ht, KeyType k, ValueType v){
    int idx;

    //check to see if map contains key. If so, remove it.
    if(containsKey(ht, k)){
        removeKey(ht, k);
    }

    idx = stringHash1(k) % ht->tableSize;

    if(idx < 0){
        idx += ht->tableSize;
    }

    //Iterate through values in the loop to find the correct spot.
    while(ht->table[idx] != 0){
        idx++;
    }

    ht->table[idx] = v;
    ht->count++;
}

//Function that replaces value

void putMap (struct hashMap * ht, KeyType k, ValueType v){
    if(!containsKey(ht, k)){ //If it doesn't contain the key, find the correct spot
        int idx = stringHash1(k) % ht->tableSize;

        if(idx < 0){
            idx += ht->tableSize;
        }
        while(ht->table[idx] != 0)
            idx++;
        ht->table[idx] = v;
        ht->count++;
    } else {
        int* value = atMap(&hashTable, word);
        *value = v;
    }
}
```

The first implementation is easier to understand, but the second will be faster because it potentially doesn't iterate through the table as many times.

2. In a table with a vector size of 6, Alan's name would hash to the same key (0) as Amy's, causing a hash collision. If the size were increased to 7, "Amy" and "Andy" would both hash to 3.

3. a)

| Name | Third letter | Hash value (mod 6) |
|---|---|---|
| Abel | E | 4 |
| Adam | A | 0 |
| Albert | B | 1 |
| Amanda | A | 0 |
| Angela | G | 0 |
| Arnold | N | 1 |
| Abigail | I | 2 |
| Adrian | R | 5 |
| Alex | E | 4 |
| Amy | Y | 0 |
| Anita | I | 2 |
| Arthur | T | 1 |
| Abraham | R | 5 |
| Adrienne | R | 5 |
| Alfred | F | 5 |
| Andrew | D | 3 |
| Anne | N | 1 |
| Audrey | D | 3 |
| Ada | A | 0 |
| Agnes | N | 1 |
| Alice | I | 2 |
| Andy | D | 3 |
| Antonia | T | 1 |

b) Table with 5 Buckets:

| Index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Assigned values | afkpuz | bglqv | chmrw | dinsx | Eqoty |
| Number of elements assigned to bucket | 4 | 2 | 3 | 9 | 5 |

Table with 11 Buckets

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Assigned values | alw | bmx | cny | doz | ep | fq | gr | hs | it | ju | kv |
| Number of elements assigned to bucket | 3 | 1 | 4 | 3 | 2 | 1 | 4 | | 5 | | |

c) Load factor (table with size 5) = 5/5 = 1. Load factor (table with size 11) = 8/11 = .73.

4. int index = (int) Math.sin(value) is a bad choice because, due to the constraints of being cast as an integer, “index” will always be either 0, -1, or 1 (but will usually be 0). This will most definitely cause hash collisions (and segmentation faults in the event of trying to access a negative integer position in a table after the function returns -1).

5. (Almost) Perfect hash functions for days of week and months of year:

| Day of week | First letter value (x) | Second letter value (y) | First letter value (z) | Word length (len) | Hash function: (1x + 2y + 3z + len) % 10 |
|---|---|---|---|---|---|
| Sunday | 18 | 20 | 13 | 6 | 3 |
| Monday | 12 | 14 | 13 | 6 | 5 |
| Tuesday | 19 | 20 | 4 | 7 | 8 |
| Wednesday | 22 | 4 | 3 | 9 | 8 |
| Thursday | 19 | 7 | 20 | 8 | 1 |
| Friday | 5 | 17 | 8 | 6 | 9 |
| Saturday | 18 | 0 | 19 | 8 | 3 |

| Month | First letter value (x) | Second letter value (y) | Third letter value (z) | Hash function: (1x + 2y + 3z) % 15 |
|---|---|---|---|---|
| January | 9 | 0 | 13 | 3 |
| February | 5 | 4 | 1 | 1 |
| March | 12 | 0 | 17 | 3 |
| April | 0 | 15 | 17 | 6 |
| May | 12 | 0 | 24 | 9 |
| June | 9 | 20 | 13 | 13 |
| July | 9 | 20 | 11 | 7 |
| August | 0 | 20 | 6 | 13 |
| September | 18 | 4 | 15 | 11 |
| October | 14 | 2 | 19 | 0 |
| November | 13 | 14 | 21 | 14 |
| December | 3 | 4 | 2 | 2 |

6. Adjacency Matrix

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **2** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| **4** | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| **7** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Edge List:

1: {2, 4}

2: {3}

3: {5, 6}

4: {5}

5: {}

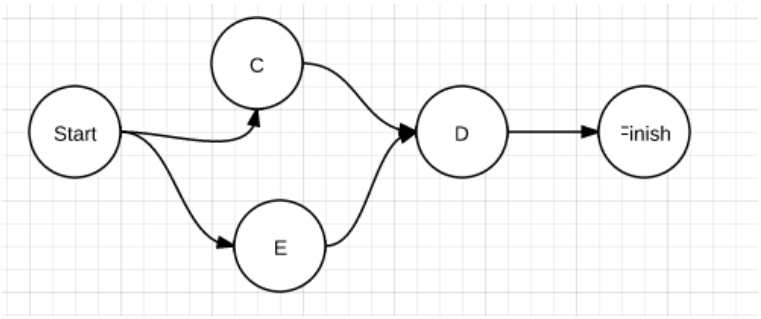
6: {7, 8}

7: {5}

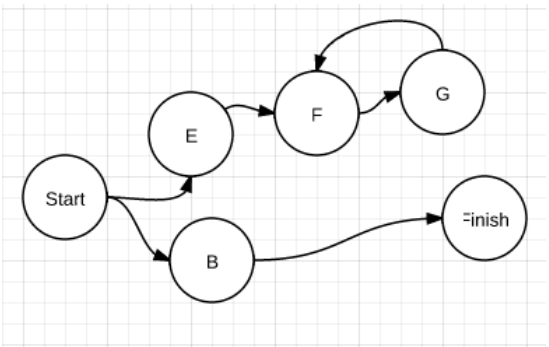
8: {}

7. In these graphs, assume DFS visits next vertex in order of greatest alphabetical position.

DFS will finish before BFS:




BFS will finish before DFS:



8. DFS

| Step | Stack | Reachable |
|---|---|---|
| 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 | Continues this way… | Continues this way… |

BFS

| Step | Queue | 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, 14 | 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 |
| 14 | Continues this way… | Continues this way… |

9.

| **Queue** | **Reachable** |
|---|---|
| Pensacola: 0 | |
| Phoenix: 5 | Pensacola: 0 |
| Pueblo: 8, Peoria: 9, Pittsburgh: 15 | Phoenix: 5 |
| Peoria: 9, Pierre: 11, Pittsburgh: 15 | Pueblo: 8 |
| Pierre: 11, Pueblo: 12, Pittsburgh: 14, Pittsburgh: 15 | Peoria: 9 |
| Pueblo: 12, Pendleton: 13, Pittsburgh: 14, Pittsburgh: 15 | Pierre: 11 |
| Pendleton: 13, Pittsburgh: 14, Pittsburgh: 15 | |
| Pittsburgh: 14, Pittsburgh: 15, Phoenix: 19, Pueblo: 21 | Pendleton: 13 |
| Pittsburgh: 15, Pensacola: 18, Phoenix: 19, Pueblo: 21 | Pittsburgh: 14 |
| Pensacola: 18, Phoenix: 19, Pensacola: 19, Pueblo: 21 | |
| Phoenix: 19, Pensacola: 19, Pueblo: 21 | |
| Pensacola: 19, Pueblo: 21 | |
| Pueblo: 21 | |

10. The purpose of Djikstra's algorithm is to find the shortest path to various vertices. This can only be done if the next-shortest path is chosen from a prioritized queue that orders path lengths from least-to-greatest, not in any arbitrary stack or queue.

11. Assuming e, the number of edges, is greater than the number of vertices, an edge-list representation of a graph requires O(e) storage.

12. An adjacency matrix of a graph with V vertices requires $O(V^2)$ spaces.

13. A breadth-first search is guaranteed to find the solution. A depth-first search will run until it reaches a dead-end. In this case, that may mean that it will run through an infinite path toward that dead end. A breadth-first search, on the other hand, explores all paths simultaneously. Accordingly, if there is a finite path to the end, the breadth-first search will find it.