Homework 5

Yousef Suleiman | Due: Apr 16

Question 1

Notice that both the upper bound of increment and decrement is $O(k)$ as in the worst case you'd need to flip all bits for both operations.

- for example, incrementing `0111` would be `1000`
- and decrementing `1000` would be `0111`

A worst case sequence of n operations would then be alternating between incrementing `0111` and then decrementing `1000`. This would be $\Theta(nk)$.

Question 2

Consider performing the operation 10 times such that the cost would be

Operation:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Cost:	1	2	1	4	1	1	1	4	1	1

Notice that the for every i^{th} operation that is a power of 2, the next j^{th} operation is always i operations away. For example, $i = 2$ is $j = i + i = 2i = 4$. Therefore, if we save an extra 2 "coins" between the operations that are powers of 2, we'll be able to cover for it without ever having a negative balance. This means that if our amortized cost $\hat{c} = 3$ then $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c} = \sum_{i=1}^{n} 3 = 3n$ such that our average complexity is $O(1)$.

Question 2

The complexity of `enqueue` is always $\Theta(1)$. The complexity of `dequeue` is at worst case $O(n)$ where `s2` is empty and n is the number of elements in `s1` (i.e. we need to pop and push everything from `s1` to `s2` which a linear operation).

Suppose the we have n operations (which can either be `enqueue` or `dequeue`). This means that `s1` must have at most n elements (in the case all n operations are `enqueue`). Because of this, we know that the upper bound of `dequeue` will need to be $O(n)$. Using the aggregate method, $O(n)/n = O(1)$.

Question 4

```
/*
where M is the maze
(a, b) is the start cross
(c, d) is the goal cross
*/
function mazeSolver(M, a, b, c, d) {
    /* Q will will be used for our BFS */
    initialize new queue Q
    /* V will keep track of visited crosses */
    initialize new matrix V with false
    /* enqueue the first cross in the path */
    Q.enqueue((a, b))
    /* C will keep track of costs of getting to certain cross */
    initialize new matrix C with -1
    C[a, b] = 0
    return BFS(c, d, M, Q, C, V)
}

/*
BFS checks if Q is empty such that the goal was never reached
because the goal was not reached, C[c, d] = -1
otherwise it checks if the dequeued cross from Q is the goal (c, d)
if not, it tries to move in all 4 directions
it returns the cost of getting to (c, d)
where (c, d) is the goal cross
M is the maze
Q is BFS queue
C is the cost matrix
V is the visited matrix
*/
function BFS(c, d, M, Q, C, V) {
    if Q is empty then return C[c, d]
    (a, b) = Q.dequeue()
    if (a, b) == (c, d) then return C[c, d]
    /* move left */
    goto(a, b, a - 1, b)
    /* move right */
    goto(a, b, a + 1, b)
    /* move up */
    goto(a, b, a, b + 1)
    /* move down */
    goto(a, b, a, b - 1)
    return BFS(c, d, M, Q, C, V)
}

/*
goto is an auxiliary function that checks if we can go to cross (x, y) from (a, b)
where (a, b) is the cross we are going from
(x, y) is the cross we are try to go to
M is the maze
Q is the BFS queue
C is the cost matrix
V is the visited matrix
*/
function goto(a, b, x, y, M, Q, C, V) {
    if ((x, y) is in M and is not * and is not V[x, y]) then {
        Q.enqueue((x, y))
        V[x, y] = true
        C[x, y] = C[a, b] + 1
    }
}
```