Problem 3

1.

```
function modify(x, v)
    if (v > x.key)
        x.key = v
        min_child = minNode(x.l, x.r)
        while (min_child != NIL)
            if x.key <= min_child.key
                break
            else
                swapNode(x, min_child)
                min_child = minNode(x.l, x.r)
    else if (v < x.key)
        x.key = v
        parent = x.p
        while (parent != NIL)
            if x.key >= parent.key
                break
            else
                swapNode(x, parent)
                parent = x.p
```

When `v > x.key`, in the `while` loop we always try to move `x` down, and a node can be moved down for at most $\lg |h|$ times (from top to bottom). `swapNode` and `minNode` can both be done in $O(1)$ time. Therefore time complexity is $\lg |h| \cdot O(1) = O(\lg |h|)$.

When `v < x.key`, instead of moving down, we are trying to move `x` up. The same analogy from above applies, therefore time complexity is also $O(\lg |h|)$.

2.

Empty locations are left blank

(a)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & 1 \\ 4 & & & 2 \end{bmatrix}$$

(b)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & 1 \\ 4 & & & \end{bmatrix}$$

(c)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & 3 \\ & & & 1 \\ 4 & & & \end{bmatrix}$$

(d)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & 3 \\ & & & \\ 4 & & & \end{bmatrix}$$

(e)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ 4 & & & \end{bmatrix}$$

3.

Elements are stored in a $N \times M$ array `A`. Each row and column has a corresponding heap `row[i]` / `col[j]`. Each elements has these extra attributes: `col_l`, `col_r`, `row_l`, `row_r`, representing its left/right child in the row/column heap. Each element also has `i`, `j` representing its index.

4.

```
/* Using heap operations from P3-1 */
/* Assmuing that "extract" operations return the extracted element */
function add(i, j, v)
    A[i][j] = v
    row[i].insert(A[i][j])
    col[j].insert(A[i][j])
```

```
function extractMinRow(i)
    row_min = row[i].extractMin()
    col[row_min.j].delete(row_min)

function extractMinCol(j)
    col_min = col[j].extractMin()
    row[col_min.i].delete(col_min)

function delete(i, j)
    row[i].delete(A[i][j])
    col[i].delete(A[i][j])
```

Maximum size of heap `row[i]` is the number of columns M. And the maximum size of heap `col[j]` is the number of rows N.

`add()`

`A[i][j] = v` is $O(1)$.
`row[i].insert()` is $O(\lg M)$. `col[j].insert()` is $O(\lg N)$. (heap operation)
Total time complexity is $O(1) + O(\lg M) + O(\lg N) = O(\lg(MN))$.

`extractMinRow()`

`row_min = row[i].extractMin()` is $O(\lg M)$. (heap operation)
`col[row_min.j].delete(row_min)` is $O(\lg N)$. (also heap operation)
Total time complexity is $O(\lg M) + O(\lg N) = O(\lg(MN))$.

`extractMinCol()`

`col_min = col[j].extractMin()` is $O(\lg N)$. (heap operation)
`row[col_min.i].delete(col_min)` is $O(\lg M)$. (also heap operation)
Total time complexity is $O(\lg N) + O(\lg M) = O(\lg(MN))$.

`delete()`

`row[i].delete(A[i][j])` is $O(\lg M)$. (heap operation)
`col[i].delete(A[i][j])` is $O(\lg N)$. (also heap operation)
Total time complexity is $O(\lg M) + O(\lg N) = O(\lg(MN))$.