Task 3.1 Using the language given and the key phrase "LA CHI CIA", the plaintext "NSA IS CIA IS IRA" encodes to cyphertext "LA PW IRA IRA WIFI"

Task 3.2 For convenience's sake, I omit the w for each word in the language, and just refence them by index (starting at 0). Then the plaintext looks like a sequence of integers, $m_0, m_1, ..., m_{d-1}$, the key as $k_0, k_1, ..., k_{l-1}$, and ciphertext $c_0, c_1, ..., c_{d-1}$. Then directly from the cipher algorithm, we have that each $c_i = (m_i + k_{i \mod l}) \mod n$, where the $i \mod l$ loops the key if the message is longer than the key, and the final modulus n wraps the resulting word around the end of language.

Task 3.3 We could try to derive an expression by solving the above for m_i, but we can more easily take advantage of the symmetry in the underlying Ceasar cypher. Since the encryption modified each word only according to its position in the plaintext, we can simply apply the inverse transformation on the word currently in that position. The additive inverse of any k_j modulo n is simply $n - k_j$. Thus we have $m_i = (c_i + (n - k_{i \mod l})) \mod n$

Task 5.2 One number will be the root, and $n - 1$ nodes will be split between the left and right subtrees

```
int num_trees(int n) {
    if (n == 0) return 1;
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += num_trees(n - i - 1) * num_trees(i);
    }
    return total;
}
```

Task 5.3 We'll also hold onto two ints, which will hold how many nodes are in the left and right subtrees of each given node. Updating this value is trivial, we just increment the proper counter at each step of insert when we move down the tree.

We'll also keep track of the min and max price of all the nodes in each subtree (4 integers). This still won't increase algorithmic complexity since at each level of each insert, we can compare the price of the inserted node to the proper subtree and update accordingly.

Task 5.4

```
tree *insert(tree *T, char *addr, int dist, int price) {
    if (T == NULL) {
        // leaf; make a new node
        T = malloc(sizeof(struct node));
        T->addr = addr;
        T->dist = dist;
        T->price = price;
        rebalance(T);
        return T;
    } else if (dist < T->distance) {
        // insert into left subtree
        T->left = insert(T->left, addr, dist, price);
```

```
        return T;
    } else if (dist > T->distance) {
        // insert into right subtree
        T->right = insert(T->right, addr, dist, price);
        return T;
    } else {
    /   / duplicate; not possible
        printf("Error: Properties with same distance     encountered.\n");
    }
    return T;
}
```

This certainly runs in $O(\log(n))$ since each recursive step progresses one layer down the tree, and the depth is always within $\log(n)$.

Task 5.5 In proto-pseudo-code:

1) Find the largest node still lower than the lower bound $O(\log n)$

2) Initialize a counter at the number of nodes in the current node's right-hand tree and then climb back up the tree to the root and at each point: $O(\log n)$

-If we came up from a right branch, do nothing.

-If we came up from a left branch, add to the counter the number of nodes in the right branch (values greater than it) + 1 (for the current node)

3) The counter now holds the number of nodes in the tree greater than the lower bound.

4) Do the same to find the number of nodes greater than the upper bound and return the difference of these two values.

Each step takes at most $O(\log n)$, so the whole algorithm is $O(\log n)$

Task 5.6 1) Find the nodes just below and above the lower and upper bounds respectively.

2) Initialize iterators at each of these nodes, max and min prices associated with each

3) Do until the lower iterator is at the root of a subtree that also contains the upper iterator (could be the root of the tree)

. i) set the lower max and min to be the max and min respectively of: the current node's price, and the max/min stored for the right sub-tree. . ii) move the iterator back up the tree. If we came from a right-hand branch, skip i) in the next loop.

4) Do until the upper iterator is at the same node as the lower iterator:

. i) set the upper max and min to be the max and min respectively of: the current node's price, and the max/min stored for the left sub-tree.

. ii) move the iterator back up the tree. If we came from a left-hand branch, skip i) in the next loop.

5) return the global max and min between the four values stored in the iterators.

This takes $O(\log(n))$ time since we search the tree twice and then traverse back up, which each take logarithmic time, and all other operations are constant-time conditionals and assignments.

Task 6.4 Note we have to check in the worst case every value in the heap, which will take order n. We assume this is a min-heap.

```
bool heap_search(heap *H, int p, int idx) {
    if (idx > heap->size || heap->priorities[idx] > p) {
```

```
        // fill in the condition above that indicates ``p is not in the heap''.
        // Hint: heaporder property
        return false;
    } else if (heap->priorities == idx) {
        // fill in the condition above that indicates ``found p in the heap''
        return true;
    } else {
        return heap_search(H, p, idx * 2) || heap_search(H, p, 1 + idx * 2;
        // fill in the body above that recursively solve the problem
    }
}
```

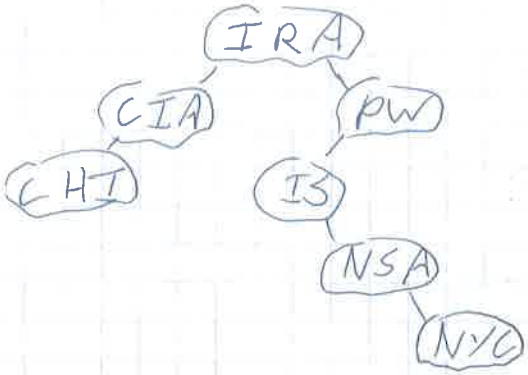
Task 6.5 To access quickly the median value instead of the max or min, we will keep two heaps: One max-heap that holds all the values below the median and a min-heap that holds the values above the median. We can envision a tree with the median as a root, and these two heaps as its left and right children.

Accessing the median is trivially $O(1)$ since we mandate its position at the root of this hypothetical tree.

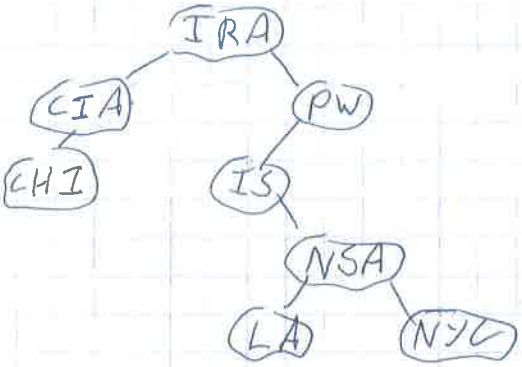
Deleting the median is also trivially $O(\log(n))$ as follows: We remove the median as above. Then compare the sizes of each heap: Remove the max or min from the larger one or, if they're even, remove from the min-heap (since we use $\lfloor n/2 \rfloor$ instead of $\lceil n/2 \rceil$). This new value is certainly the new median, so place it in its spot and restore the heap property. Our logic takes $O(1)$ since the heap sizes are stored, and the removal takes $O(\log(n))$ since we're using a standard heap.

Inserting (though not mandated by the question) is also $O(\log(n))$ at least by the following method–we can optimize, but not asymptotically. If the new value is larger than the current median, insert into min-heap, if smaller, than into the max-heap. Then compare sizes and if they're not equal within one (according to proper division rounding) then insert the median into the smaller heap and remove the root of the larger heap into the median. Each of these operations takes $O(\log(n))$, so our asymptotic behavior is the same.

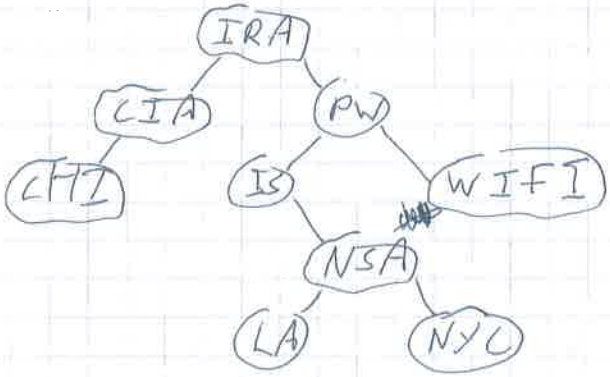
Queue: NYC, LA, WIFI



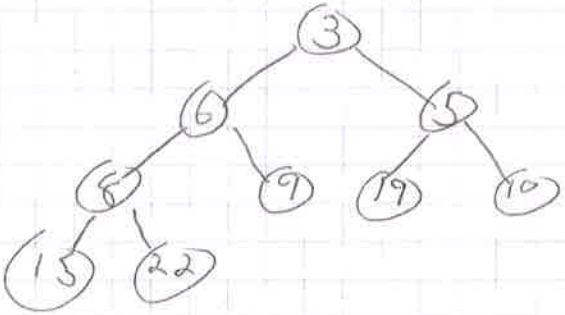
Queue: LA, WIFI



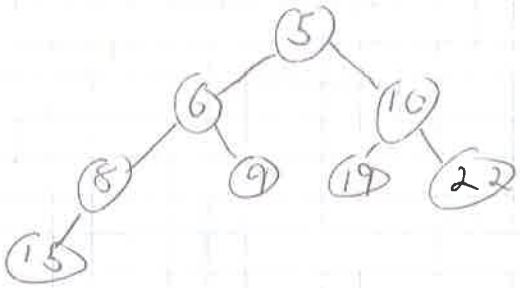
Queue: WIFI



Task 6.1



Task 6.2



Task 6.3

priorities =

index	value (size = 8)
0	5
1	6
2	10
3	8
4	9
5	19
6	22
7	15
8	—
9	—
10	—
11	—
12	—
13	—
14	—
15	—