1.

First, we need an variable keeping track of the first and last place of the array. We also need a mid value, which allows us to recurse on half of the list within a loop. Next, we create a loop where we check the first, last, and mid variables. If it isn't any of these, it has to be in the other parts of the array, and we recurse depending on how the key relates to the mid value. If the mid value is higher than the key, we recurse on the first half of the array. Since the array is sorted, we just search the lower half of the array because it is less than the middle value. Otherwise, we search the upper half. Then, we return -1 if we weren't able to find the value.

Algorithm runs in log(n) time because the maximum runtime consists of
$T(n) = T(n/2) + O(1) + O(1) + O(1) + O(1) + O(1)$
$T(n/2) = 2T(n/4) + 1$
$T(n/4) = 2T(n/8) + 1$
And in the worst runtime, this runs in O(log(n)) time.

Runs in log(k) time otherwise, because T(n/2) means that it returns a letter depending on location, and if n=k,
runtime = $T(k/2) + O(1) + O(1) + O(1) + O(1) + O(1)$
=log(k)

2.

First, we create a node class in order to be able to initialize nodes. Next we create a tree with buildTree which uses the first, last, and mid in order to create a node as a root, and then creates the left and right subtrees (where we place additional nodes according to double value 'd'). We also need a newNode function in order to instantiate the nodes. I created a showTree based off of the tree we coded together in class, where we recurse on the left subtree, show the current node and then recurse on the right subtree.

Algorithm runs in O(n) time because

```
  root->left = buildTree(arr, start, middle - 1);
  root->right = buildTree(arr, middle + 1, end);
```

Are both running in T(n/2) time, and
$T(n) = 2T(n/2)$
which solves to O(n) time. This is because 2T(n/2) solves to (n^($\log_2 2$)) which solves to O(n).