1. There are 6 primary classes in total:

- BinaryNode: Represents a node stored in an unbalanced binary search tree. It contains the references to a city record, the left and the right subtrees.
- City: Represents a city record in the database. A city record contains the information about it, which are its name, population and payload. Every city record is referenced by one of the node in the binary search trees.
- DatabaseBST: Represent a generic Binary Search Tree that references to the database in an efficient way for later searching.
- NameComp: Represent a comparator for the Name Binary Search Tree. It will be used for making decisions in directions (left or right) while going from the top to the bottom of the tree.
- PopulationComp: Represent a comparator for the Name Binary Search Tree. It will be used to make decisions in directions (left of right) while going from the top to the bottom of the tree.
- bst: The main program that reads the inputs and return the outputs accordingly.

2. In order to support the findKth method:

- I created one more field in the BinaryNode class: int leftSize, which counts the number of nodes in the left subtree of the current node. The size of the left subtree is increased by 1 if a node is inserted onto the left of the current node. Similarly, the size of the left subtree is decreased by 1 if a node is deleted from the left of the current node. Some small modifications in the insert( ) and delete( ) methods helps me to achieve that without affecting the complexity of those 2 methods. Finally, in the findKth( ) method, I recursively go down the tree until leftSize == K (since it's 0 indexed) and return the result. Therefore, the overall complexity for the findKth( ) method is O(log n) with n is the number of nodes on the trees.

3. Average case running times of all operations (suppose that n is the number of nodes on the trees and the trees are kept being balanced):

- insert: it keeps going down the tree until it finds an appropriate place to add a node in. The height of the tree is about log n, so the complexity is O(log n).
- find: it keeps going down the tree until it finds all of the desired duplicates. The height of the tree is about log n, so the complexity is O(log n).
- findKth: as explained above in 2., so the complexity is O(log n)
- findRange: finds all the city records whose values are within the range. It changes direction right away when there's some value that is out of the range. Worst case is O(n) when the whole tree is within the given range.
- delete: first, calls the find operation to obtain a list of all desired duplicates. Then, use the duplicates' addresses to remove all of them out of the two trees. The height of the tree is about log n, so its complexity is O(log n).
- sort: do an in-order traversal. The complexity is O(n)

- tree: also do an in-order traversal. The complexity is O(n)
- makenull: empty the tree. The complexity is O(1)