1. None that I know of.
2. MyHash:

For my hash table, I had a pointer to an array of pointers (one per bucket) called m\_hashTable. I also held private member variables for the max load, the number of items, and the number of buckets. The constructor was simply creating an array of 100 pointers and setting all of them to nullptr, indicating no items in each bucket. The destructor looped through each bucket and if there is a linked list of pointers, then delete every item in the bucket. The reset function essentially was the destructor followed by the constructor. The find function determined the bucket of the key and then searches through the bucket’s linked list, if it has one, to see if any of the items has a matching key and returns its corresponding value. The associate function first calls the find function to see if the key is already contained, and if so, then it updates the value and returns. Otherwise, it checks if the load limit is passed, and if it is, then it dynamically allocates a new array of bucket pointers that’s double the size. The function copies over all the items in the previous array in, and then deletes it. Finally, the new key is inserted in. Otherwise, if the max load isn’t surpassed, then the key is simply inserted into the hash table.

Tokenizer:

Tokenizer contains a set of all the separators that is initialized at the time of construction. When tokenize is called, it creates a vector of strings and a string. For each character in the input string, if it’s contained within the set of separators, then push the string to the vector as long as it’s not empty and reset the string back to empty. Otherwise, it adds the character to the end of the string. There’s another check to see if the string contains characters at the end in case there’s no separator at the end of the string. Finally, the vector of words is returned.

WordList:

The data structures that make up WordList are two MyHash maps. The first one holds each of the strings for simplicity in the contains function. The second holds a mapping of letter patterns to a vector of all strings that match that letter pattern. I have a helper function that uses a MyHash map to get the letter pattern of a word. If a character is not found in the map, then associate it paired with a capital letter, incrementing the letter afterwards. After the mapping is created, a new string is created based on the mapped values of each character “key” and that’s the letter pattern outputted. The loadwordlist function first resets both of the MyHash maps. Then if it can successfully read the input, it takes the lower case version of each word in the file and adds it to the first vector. For the second vector, it associates the pattern with either a new vector containing that one string, or the vector with a new string appended to the end. The contains function simply calls the MyHash find function on the first hash table and if the resulting pointer isn’t nullptr, then it returns true.

vector<string> WordListImpl::findCandidates(string cipherWord, string currTranslation):

create a new vector of strings to hold matches

if the sizes of the two words are different, return empty vector

if cipherWord contains anything but letters or apostrophes, then return

if currTranslation contains anything but letters, apostrophes, or ‘?’, then return

get the lower case of both cipherWord and currTranslation

make a new vector of strings equal to the vector that corresponds to the letter pattern of lowercase cipherWord in the second hash table

for each word in that vector,

if every index of the word matches currTrans or the index of currTrans is ‘?’

push it into the vector of matches

otherwise

move to the next word

return matches vector

Translator:

The data structures incorporated in translator include a dynamically allocated array of characters that holds ‘A’ to ‘Z’, two vectors, one which holds a dynamically allocated array of characters that corresponds to the translations of plain text that is known. The other holds a dynamically allocated set that contains all the letters of plain text that have been translated. For example, if a mapping of “AZ” to “XD” is pushed, then the most recent index of the vector will hold ‘X’, ‘?’, ‘?’, ‘?’, …, ‘?’, ‘D’ (24 ‘?’ total). The vector holding the sets will hold ‘X’ and ‘D’. This means that the constructor pushes in a mapping of 26 question mark characters to the first index of the former vector and an empty set to the first index of the latter vector. The destructor destroys each of the arrays contained in both vectors as well as the array of characters from 'A’ to 'Z’. The pushMapping function works as follows:

If the number of characters of the two strings are different, return false

Go through each character

          If that index in the strings is not a letter of the alphabet then return false

          If the most recent set contained in vector 2 already contains the plain text character

                     Unless the mapping between the cipher letter and the plain letter already exists, return false

Dynamically allocate a new array of characters and a new set.

Copy over the array/set of the previous mapping into the new array/set.

For each of the letters in the translation string

          Insert the plain text character into the set

          Determine the index that should be changed in the plain text array that was just created

           If that index is currently inhabited by either the translated plain character already or a ‘?’ then insert it in at that index

           Else delete both the new array and the new set (indicating that there's a conflicting mapping already) and return false

Push back the new array and the new set and increment the counter

Return true

popMapping first checks if the counter is greater than 1, if it isn't then it returns false. Otherwise it decrements the counter, deletes array and set at that index and pops the counter index of the vector holding the plain translation and the counter index of the vector holding the sets. The getTranslation function creates a temporary string and looks at each character of the string to be translated. If it's not a letter, then that character is appended to the string. Otherwise, it looks at the most recent mapping in m\_plain and appends either a ‘?’ or the translated character based on whether or not there is a corresponding letter.

Decrypter:

My decrypter class contained a wordlist, a translator, and a tokenizer. The constructor only initialized the tokenizer with the string of characters that aren't letters. The load function simply returned and called loadWordList function.

The crack function works as follows:

Tokenize the ciphertext

Translate each string in the token vector and select the string with the least converted based on the number of ‘?’s

Translate the selected string

Create a vector that holds all the possible translations by calling findCandidates on the word list

If that vector is empty

Pop the mapping and return an empty vector

Create a new vector called ans that holds the possible solutions

For each string in the possible translations vector

Push the mapping created by that translation

If that was successful (i.e. no contradictions)

Translate the cipher text and tokenize it

Count the number of completed words

If any of the completed words are not found in the word list, pop the mapping

If all the words were fully translated

Push the fully translated string into the ans vector

Pop the map

Otherwise

Recursively call crack on the ciphertext with the new mapping

Insert the recursively obtained vector at the end of ans

Otherwise if not fully translated, continue

If not successful, also continue and move to next mapping

Pop the mapping of the translator

Sort the ans vector

Return the ans vector

3. I think all of the methods satisfy the big-O requirement.