README file for assignment 3:

I have mentioned all additional classes with their methods and their implementations, if required.

A. **class Pair_ <F, S>**
   - **public String toString():** returns concatenation of the Strings of both elements in the pair.
   - **public int compareTo (Pair_ <F, S>):** compares the first element and then the second element of pair. If element to be compared is bigger than passed element, it returns -1.
     If it is lesser, it returns 1.
     If it is equal, 0 is returned.

B. **class Student**
   - **public String toString():** returns a string contain all information about the student, in the format: firstName secondName hostel department CGPA.

C. **class BSTNode <K, T>:** the nodes used in creating hash table with separate chaining with BST linked at each index of table.
   - Its constructor takes key and object as parameters and stores it, while keeping a reference to the left and right child node.
   - If no arguments are passed to the constructor, it simply creates a node with both key and object as null.

D. **class HashTableSCBST <K, T> :** I have maintained an array of BSTNode objects, where each object in the array is the root of the chained BST.

   - **public int insert(K key, T obj):** First the value of index is calculated using h1 function. If table[index] is null, the object is inserted as root of the chained BST.

     If it is not null, it is inserted in the already existing BST such that an object (Student) with lower or equal value of first element (firstName), as compared to the node, is inserted to the left of the node, else to the right. If first element (firstName) is same, second element of pair(lastName) is compare.
     The value of count of nodes touched keeps increasing by one, each times it goes left or right.

   - **public int update (K key, T obj):** If the value of passed key, compared with already present node's key is lesser, it proceeds to find it in left subtree and count of nodes touched is increased.
     If it is greater, it proceeds to find it in right subtree and count of nodes touched is increased.
     If it is equal, it replaces the object with new object and count of nodes touched is increased.
     If it reaches null, it means key is not present in BST and returns 0.

- **public int delete (K key):** If the value of passed key, compared with already present node's key is lesser, it proceeds to find it in left subtree and count of nodes touched is increased.
  If it is greater, it proceeds to find it in right subtree and count of nodes touched is increased.
  If it is equal, it means it is the node to be deleted and count of nodes touched is increased.
  - If left child of this node is null, it replaces the node to be deleted by its right child and count of nodes touched in increased.
  - If right child of this node is null, it replaces the node to be deleted by its left child and count of nodes touched in increased.
  - If none of the node (say N) is null, it means it has both left and child. In this case, we first find the node with the minimum value of key (say M) in its right subtree (greater than this node's key). On the way to find it, each time a we go left, count is increased by one.

    Then N's key and object is replaced by M's key and object. Then the right child of M is set equal to N's right child after deleting M from it, and its left child is set equal to N's original left tree.

- **public boolean contains (K key):** Proceeds search similar to update() method. If key at any node equals to the passed key, it returns true. If it reaches null while searching it returns false.

- **public T get (K key):** Proceeds search similar to update() method. If key at any node equals to the passed key, it returns the object at that node. If it reaches null while searching it returns false.

- **public String address (K key):**
  First, the value of index is calculated and added to the string to be returned, in the format: 'index-'.

  If the value of passed key, compared with already present node's key is lesser, it proceeds to find it in left subtree and adds L to the string to be returned.
  If it is greater, it proceeds to find it in right subtree and adds L to the string to be returned.
  If it is equal, it returns and string ends.

E. **class HashEntry <K, T>:** the nodes used in creating hash table with double hashing
   - Its constructor takes key and object as parameters and stores it.
   - If no arguments are passed to the constructor, it simply creates a node with both key and object as null.

F. **class HashTableDH <K, T>:** I have maintained an array of HashEntry objects, where each object in the array is a HashEntry.

- **public int insert(K key, T obj):**
  First value of index is calculated using hash1 method (using key) . If table[index] is null, object is inserted at table[index]. Else index is increased by value obtained by passing key in hash2 method. Count of steps is increased by one each time a new value of index in calculated.

- **public int update (K key, T obj):** First value of index is calculated using hash1 method (using key) . If table[index] is null (which may be caused due to deletion) or it's key is not equal to the passes key, we increase index by hash2 and count of steps by one. As soon as table[index]'s key becomes equal to key, it replaces the object at table[index] by the new object passes as parameter, and increases the number of steps by one.

- **public int delete (K key):** First value of index is calculated using hash1 method (using key) . If table[index] is null (which may be caused due to deletion) or it's key is not equal to the passes key, we increase index by hash2 and count of steps by one. As soon as table[index]'s key becomes equal to key, it replaces the object at table[index] by a 'null' and increases the number of steps by one.

- **public boolean contains (K key):** First value of index is calculated using hash1 method (using key) . If table[index] is null (which may be caused due to deletion) or it's key is not equal to the passes key, we increase index by hash2 and count of steps by one. As soon as table[index]'s key becomes equal to key, it returns true.

  If the value of index has been changed 'hashTableSize' number of times, it means it has traversed entire table (due to size being prime) and still key is not found. So it returns false.

- **public T get (K key):** First value of index is calculated using hash1 method (using key) . If table[index] is null (which may be caused due to deletion) or it's key is not equal to the passes key, we increase index by hash2 and count of steps by one. As soon as table[index]'s key becomes equal to key, it returns the object at that index.

- **public String address (K key):** First value of index is calculated using hash1 method (using key) . If table[index] is null (which may be caused due to deletion) or it's key is not equal to the passes key, we increase index by hash2 and count of steps by one. As soon as table[index]'s key becomes equal to key, it returns the string representation of the index.

Time Complexities:

1. **Separate Chaining using BST:**
   For inserting/deleting/updating an element in BST, we may have to traverse to the last level of nodes. Hence its time complexity is O(H).
   Now if the size of table is S, and number of elements to be inserted is N, the average height of the BST is log(N/P). Hence average time complexity of all operations is O(logN).

   In worst Case, size of table may be 1. And H=N in worst case. Hence worst case complexity is O(N).

2. **Double Hashing:**
   If table size is S, and number of elements N.
   Then the average time complexities of all operations is O (1/ (1-loaf factor)) which equals O(1).

   Worst TIme Complexity

   Suppose all the keys have same values of hash1 and hash2, then we have to traverse through each element, hence worst time complexity is O(N) for all operations.