class Student: implements Student_

Stores fname, lname, hostel, department and cpga of the student
fname() returns the first name of the student
lname() returns the last name of the student
hostel() returns the hostel of the student
department() returns the department of the student
cgpa() returns the cgpa of the student
toString() function overrides the toString() function and returns fname + lname + hostel + department + cgpa

class Pair:

Stores two values of String type (namely fname and lname)
toString() function overrides the toString() function and returns String fname+lname

class Node:

it contains two objects key and object and has two nodes in it both initialised to null

class DoubleHashing: implements MyHashTable_

Stores the size of the hashtable and maintains a HashTable storing the objects of type Node
insert() finds the index in which the object is to be inserted using the given key and deals with collisions
delete() finds the element with the given key and delete the object. It returns the total number of steps to find the final index of the element or returns 'E' if the element does not exist in the table
contains() finds element in the table with the given key and returns true or false accordingly
update() finds the element with the given key and updates the object. It returns the total number of steps to find the final index of the element or returns 'E' if the element does not exist in the table
get() finds the element with the given key and returns the value of the LNode. It throws NotFoundException if the element does not exist in the table
address() returns the final index of the element with the given key. It throws NotFoundException if the element does not exist in the table

Double Hashing:

n is the number of elements to be inserted.

insert():

Best case: O(1): When the index is free, we simply insert the element in the given index and hence O(1)
Worst case: O(n): When calculated initial index for all the elements is the same then we will have to loop n times to calculate new hash index in the worst case
Expected case: O(1): It is the observed complexity

update():

Best case: O(1): If we find the element at the initially calculated index then we just require a constant time and hence O(1)
Worst case: O(n): When calculated initial index for all the elements is the same then we will have to loop n times to calculate new hash index in the worst case
Expected case: O(1): Same as insert.

delete():

Best case: O(1): Same as update
Worst case: O(n): Same as update
Expected case: O(1): Same as update

contains():

Best case: O(1): Same as delete
Worst case: O(n): Same as delete
Expected case: O(1): Same as delete

get():

Best case: O(1): Same as contains
Worst case: O(n): Same as contains
Expected case: O(1): Same as contains

class Node:

- Stores Key, Value, left and right. Object of this class is stored in the hashtable of SeparateChaining

class BST:

Stores the root of the BST intially root is null.
insert() inserts the element in its proper position in the BST.It returns the depth of the node and prints E if already added
update() finds the element with the given key and updates the object. It returns the depth of the node and prints E if already added
delete() finds the element with the given key and delete the object. It returns the depth of the node+the no. of nodes visited before deleting the node and prints E if it node doesn't exist.
contains() finds element in the BST with the given key and returns true or false accordingly

get() finds the element and returns depth of node. It throws NotFoundException if the element does not exist in the table
address() returns the path of the node by using L or R accordingly. It throws NotFoundException if the element does not exist in the table

class BSTHash:
implements uses BST
makes a hashtable which uses hash values and double hashing to give index to elements in case of collisions.
insert()finds depth of node and then calls and returns the value from the insert function of the BST
update() finds depth of node and then calls and returns the value from the update function of the BST
delete() finds the element with the given key and delete the object. It returns the total number of steps to find the final index of the element or returns 'E' if the element does not exist in the table
contains() finds the node and returns true or false using function of BST
get() calculates the index of the BinaryTree in which the element is to be searched and then calls and returns the value from the get function of the BST
address() calculates the index of the BinaryTree in which the element is to be searched and then calls and returns the value from the address function of the BST

class assignment3:
DH() and SCBST() reads the file and answers the queries accordingly
returnStudentDetails() returns a string which contains all the details of the given student.

Separate Chaining using Binary Search Tree:
n is the number of nodes/elements in the tree.
h is the height of the tree (which is atleast log n (in complete tree) and at max n (all on same side of root))
insert():
Best case: O(log n): In a tree, how deep we need to go before inserting an element is of O(h) and in the best case it is O(log n)
Worst case: O(n): In a tree, in the worst case the tree created is a leaning tree and hence O(n)
Expected case: O(log n): It is observed

update():
Best case: O(log n): In a tree, how deep we need to go for finding and updating an element is of O(h) and in the best case it is O(log n)
Worst case: O(n): In a tree, in the worst case the tree created is a leaning tree and hence O(n)
Expected case: O(log n): It is observed

delete():
Best case: O(log n): In a complete tree, finding the element will take k (less than log n) steps and then finding the successor will take the remaining log n - k steps at max, hence a total of O(log n)
Worst case: O(n): In a tree, we need some k < h steps for finding and remaining h-k steps for finding the successor hence a total of h steps which is O(n) in the worst case
Expected case: O(log n): It is observed

contains():
Best case: O(log n): In a tree, how deep we need to go for finding an element is of O(h) and in the best case it is O(log n)
Worst case: O(n): In a tree, in the worst case the tree created is a leaning tree and hence O(n)
Expected case: O(log n): It is observed

get():
Best case: O(log n): In a tree, how deep we need to go for finding and returning an element is of O(h) and in the best case it is O(log n)
Worst case: O(n): In a tree, in the worst case the tree created is a leaning tree and hence O(n)
Expected case: O(log n): It is observed