README ASSIGNMNET 3

<u>To complete the assignment I made the following .java files</u>:-

1. MyHashTable_<K,T>:
   a. Contains the class definition of the NotFoundException
   b. The MyHashTable_<K,T> interface as defined in the assignment webpage
2. MyHashTableDH<K,T>: It is an implementation of the MyHashTable_<K,T> interface. It uses double hashing to deal with collisions in the hash-table.
   a. Fields: -
      i. Object[] hashtable : An array that contains objects of type KeyValueStatus<K,T>. This is the actual hash-table. Elements of this array are store the Key, Value and Status(full empty or temporarily empty) at an index of the hash-table. It is initialized with a KeyVlaluStatus at each index having key and value set to null and status set to empty when a new hash-table is created using the MyHashTable<K,T> implementation
      ii. int tablesize: Size of the table a prime integer greater than the size of the dataset
   b. Methods:-
      (Note: Assume that n is the size of the hash-table)
      i. MyHashTableDH(int size): Time Complexity : O(n)
         Constructor, takes the hash-table size as argument and initializes the tablesize field and also the hash-table field with KeyValueStatus (custom class) objects having key and value at every index set to null and status at every index set to empty as each index is initially empty
      ii. Two hash functions:
         - djb2(String str, int hashtableSize) : h1
         - sdbm(String str, int hashtableSize) : h2
      iii. insert(K key, T obj): Time Complexity : Best O(1) | Average O(1) | Worst O(n)
         Assumes that no slot in the hash table exists with having a KeyValueStatus with key i.e. the key passed as argument to the function. It finds a place to insert the new key, object pair in the hash-table using double hashing as specified in the assignment. It inserts the new key, object pair in a position which is not already full and updates the status at that position to full. Also returns the number of hashes done to insert the new object
      iv. update(K key, T obj): Time Complexity : Best O(1) | Average O(1) | Worst O(n)
         Assumes that a slot in the hash-table exists with key passes in the argument and updates it object to the object passed in the function. It uses double hashing to find the appropriate slot at which update needs to take place and updates the object at that index to the new object. Also returns the number of hashes done to update the key
      v. delete(K key) : Time Complexity : Best O(1) | Average O(1) | Worst O(n)
         Assumes that a slot in the hash-table exists with key passes in the argument and updates it object to the object passed in the function. It uses double hashing to find the appropriate slot at which deletion needs to take place and sets the key and value at that index of the table to null and updates the

status of that index to Temp_empty i.e. temporarily empty. Also returns the number of hashes done to perform the delete operation

vi. contains(K key): Time Complexity : Best O(1) | Average O(1) | Worst O(n)
Searches the hash-table using double hashing and stops searching until It finds an empty slot i.e. with status empty or a slot that actually contains the KeyValueStatus object corresponding to the key provided as argument or it has searched the entire table. Returns true is the while searching it finds a position in the hash-table that has the KeyValueStatus object corresponding to the key, false otherwise

vii. get(K key) : Time Complexity : Best O(1) | Average O(1) | Worst O(n)
Uses a search algorithm similar to the contains method and returns the object i.e. of type T is the search results in a position containing the KeyValueStatus object corresponding to the key throws the NotFoundException otherwise

viii. address(K key) : Time Complexity : Best O(1) | Average O(1) | Worst O(n)
Uses a search algorithm similar to the contains and get method and in case of a positive result of search returns the index at which the key is found else throws NotFoundException

3. MyHashTableSCBST<K extends Comparable<K>,T> : This is another implementation of the MyHashTable_<K,T> interface. It uses separate chaining using Binary Search Tree as the auxiliary data structure to deal with collisions in the hash-table.
   a. Fields :-
      i. Object[] hashtable: This is the array which is the hash-table that uses separate chaining to deal with collisions. Each index of this array contains the an object of type BST<KeyVal<K,T>> i.e. a BST that has Nodes having values of type KeyVal<K,T> i.e. a key value pair
      ii. Int tablesize: This is the size of the hash-table
   b. Methods :-
      i. MyHashTableSCBST(int size) : Time Complexity : O(n) n: size of the hash-table
         Constructor : Take the size of the hash-table as argument and initializes the fields appropriately
      ii. djb2(String str, int hashtableSize) : h1
      iii. insert(K key, T obj) : Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
         Assumes that the hash-table does not contain a KeyVal object with the key equal to the key to be inserted. Finds the index at using the hash function h1 where the new KeyVal is to be inserted and calls the insert method of the BST class on the BST at the required index and passes the new KeyVal with Key and Value equal to the passed key and value in the argument of the function. Also returns the number of nodes of the BST touched while inserting the new object
      iv. update(K key, T obj): : Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
         Assumes that the hash-table does contains a KeyVal object with the key equal to the key to be updated. It gets the reference to the KeyVal object

using the search() method defined in the BST class and updates the value to the new object. Also returns the number of nodes touched while updating the key

v. delete(K key) Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
Generates the KeyVal to be deleted using only the key as the Keyals are compared using the Keys only the value set to null makes no difference. Using the KeyVal generated it calls the delete method of the BST class on the BST at the appropriate location in the hash-table and returns the number of nodes touched while deleting the required node

vi. contains(K key) : Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
Generate a KeyVal with the given Key and value set to null. Uses the search method defined in the BST class that returns null if the a node with given KeyVal is not found in the BST and returns true if the node is found and false otherwise

vii. get(K key) : Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
Uses the search method defined in the BST class to get a reference to the node having the KeyVal with Key key and returns its object if the key is found and throws NotFoundException if the key is not found i.e. when the search function returns null

viii. address(K key): Time Complexity : O(h) where h:height, in a completely random case O(α) α:load factor | Worst O(n) where n : number of elements in the hash-table
Uses the search function to return the address of the key in the hash-table if the key exists in the hash-table and throws NotFoundException otherwise

4. BST<K extends Comparable<K>> : A generic BST whose nodes contain objects of type K whose objects are comparable each other. In this particular implementation of BST Right sub-tree of any node contains all the nodes in the BST that have a values that are greater than or equal to the value of the current node and the left sub-tree contains all the elements that are strictly less than the current node's value
   a. Fields :-
      i. Node<K> root : The root of the BST
   b. Methods :-
      (Note : Assume that h is the height of the BST and n is the number of elements in the BST)
      i. BST(Node<K> node): Constructor Takes a node and initializes the root of the BST with that node
      ii. insert(K insert_node_value): Time Complexity : Best O(logn)| Average O(h) | Worst O(n)
      Takes the value that is to be inserted to the BST and finds the appropriate place in the BST where the value can be inserted, the search stops when the cursor hits a null i.e. when a reference to be followed is null in that case a new node with the value equal to the insert_node_value is made and the

reference which was to be followed is set to point the newly created node. It also returns the number of nodes that are encountered while inserting the node with the new value in the BST

iii. delete(K delete_node_value) : Time Complexity : Best O(logn)| Average O(h) | Worst O(n)
Assumes that there exists a node with the value equal to the delete_node_value in the BST which is to be deleted. Now, for deletion it first finds the node in the BST that has the value equal to the delete_node_value. After finding the node to be deleted the actual deletion is performed in different ways depending on the following cases and also returns the number of nodes encountered while performing the deletion:-
    - Case 1 : if the node to be deleted has no child :-
        - Case 1.1 : If the node is the root of the BST : The root of the BST is simply set to null
        - Case 1.2 : The node to be deleted is not the root of the BST but is the left child of its parent then the left child of the parent was set to null
        - Case 1.3 : The node to be deleted is not the root of the BST but is the right child of its parent then the right child of the parent was set to null
    - Case 2 : The node to be deleted has no left child but has a right child
        - Case 2.1 : The node is the root of the BST :
            - Root of the BST is set to the right child of the delete node
            - Delete node's right child's parent is set to null
        - Case 2.2 The node to be deleted is not the root of the BST but is the left child of its parent then
            - The left child of the parent was set to the right child of the delete node
            - Delete node's right child's parent is set to the delete node's parent
        - Case 2.1 The node to be deleted is not the root of the BST but is the right child of its parent then
            - The right child of the parent was set to the right child of the delete node
            - Delete node's right child's parent is set to the delete node's parent
    - Case 3 : The node to be deleted has no right child but has a left child. Handled similar to case 2
    - Case 4 : The node to be deleted has both right and left children then we find the successor of the node which is the smallest element of the right sub-tree of the node and then replace the value of the node to be deleted by the value of the successor and the corresponding child i.e. left/right depending on where we find the successor of the successor's parent is set to the successor's right child (which may possibly be null) also if the successor has a non-null right child then its parent is set to the successor's parent

        iv. Object[] search(K search_node_value) : Time Complexity : Best O(1)| Average O(h) | Worst O(n)
        This method searches the node with value search_node_value in the BST and returns an array of three objects of the type Object if the node if found and null if it is not found. The elements of the returned array are of fixed type so that type can be type casted in future for specific uses. The Object objects of the returned array are:
            - A reference to the node having the value equal to the search_node_value so that it can be manipulated if the need arises (Type : Node<K>)
            - The number of nodes encountered in the deletion process (Type : int)
            - The address of the found node in the format given in the assignment 3 webpage (Type : String)

5. Node<K> :
    a. Fields: -
        i. Node<K> right_child : The right child of the current node
        ii. Node<K> left_child : The left child of the current node
        iii. Node<K> parent : The parent of the current node
        iv. K value : The value of the current node
    b. Methods:-
        i. Node(K val, Node<K> parent_node, Node<K> left_node, Node<K> right_node) : Constructor 4 arguments and initializes all the 4 fields appropriately
6. KeyVal<K extends Comparable<K>, T> implements Comparable<KeyVal<K,T>> : Key value pairs that will be stored as the values of the Nodes in the BST
    a. Fields :-
        i. K key : The key of the key value pair
        ii. T value : The value of the key value pair
    b. Methods :
        i. KeyVal(K new_key , T new_value) : Constructor – Takes 2 arguments and initialises the two fields
        ii. compareTo(KeyVak<K,T> another_KeyVal) : Compares the two KeyVal objects based on their keys
        iii. equals(Object obj) : Checks the equality of the two KeyVal objects using only the equality of their keys
7. KeyValueStatus<K,T> : The objects of this class are used to store the key, value and the status of a slot in the hash-table in the Double Hashing implementation.
    It defines the enum Status to take 3 values
    i. Empty: If the slot was never filled
    ii. Temp_Empty : If the slot was filled once but the value stored in it was removed
    iii. Full : If the slot is currently full
    a. Fields :-
        i. K key : Key stored previously in the given slot
        ii. T value : Value stored previously in the given slot
        iii. Status status : The status of the current slot
    b. Methods :-

        i. KeyValueStatus() : Constructor – Takes no argument and initialises the status as empty, the key and value will be externally initialized when the slot is filled

8. Pair<X, Y> implements Comparable<Pair<X, Y>> : The class Pair<X,Y> is a generic form of the a pair of values of two different type X and Y
    a. Fields:-
        i. Private X x : value of the X part of the pair
        ii. Private Y y : value of the Y part of the pair
    b. Methods :-
        i. toString() : Returns x.toString()+y.toString()
        ii. compareTo(Pair<X,Y> another_pair) : Compares the given pair with another_pair only based on the value of the compareString defined for each pair
        iii. compareString() : returns this.x.toString i.e. depends only on x
        iv. equals(Oject obj) : Checks equality based on the equality of the toStrings of the two pairs
9. Student_ : Interface used as in the assignment 3 webpage
10. Student : Class that implements the student interface
11. Assignment3 : The driver class that has the main method to process the inputs from the file.
    a. Methods :-
        i. Public static void main(String args[]) : Takes three String arguments and creates a hash-table using either of the two implementations depending on the argument passed for the type of implementation and the size of the hash-table and then passes this array with the file name to another method process_inputs
        ii. Private static void process_inputa(MyHashTable_<Pair<String,String>,Student> hashtable,String input_file_path) : Takes a hash-table and a file name as input and line by line processes the queries in the input file on the hash-table and gives the required output

Time Complexity Analysis:

1. Separate chaining : The time complexity of the separate chaining implementation is same as that of the BST assuming that the hash function h1 hashes keys in O(1) time. Therefore the time take for insertion, deletion and search as same as that for the operations in a BST which is O(height of the BST) given a BST and O(log n) where n is the number of elements in the BST given that the distribution (order of insertion in BST) is completely random.

2. Double Hashing:
We express our analysis in terms of the load factor $\alpha = n/m$.

Claim: given an open address hash table with load factor $\alpha = n/m < 1$, the expected no. of probes in n unsuccessful search is at most $1/(1-\alpha)$ assuming uniform hashing.

Proof: Expected no. of probes in case of search

$$E(x) = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i}$$

$$= \frac{1}{1-\alpha}$$

Getting the average insertion time using actual test cases:-

Test Case : Inserting 799 elements.

1. Checking Double Hashing Implementation:--
   a. Table size: 1229
      Load Factor : 0.65
      Average Steps : 1.58
   b. Table size: 1259
      Load Factor : 0.63
      Average Steps : 1.60
   c. Table size: 1289
      Load Factor : 0.61
      Average Steps : 1.60
2. Checking Separate Chaining implementation:-
   a. Table Size:7
      Load Factor : 114.14
      Expected Steps : 6.83 (= log (Load Factor) for a completely random case)
      Actual Average Steps: 7.67
   b. Table Size: 10
      Load Factor : 15.98
      Expected Steps : 3.99
      Actual Average Steps: 4.13
   c. Table Size: 100
      Load Factor : 7.99
      Expected Steps : 2.99
      Actual Average Steps: 3.15

Conclusion : Therefore the average number of steps for insertion for the implementations are the same as the theoretically predicted values within the experimental limits.

Double Hashing vs Chaining

1.Separate Chaining is better in terms of space as it allows a load factor greater than 1 and collisions are taken care of by maintaining a BST at each slot. However it takes an average of O(log α) time for its operations which is worse than the Double Hashing implantation.
In the worst case if the hash function is known then input might be given that makes the implementation behave as a linked list and the operations become O(n) where n is the number of elements in the hash-table.

2. Double Hashing provides an efficient implantation as its operations have a complexity of O(1) but it also requires a lot of space as it does not allow values of load factor, $\alpha<1$ therefore the size need to be greater than that of the dataset.