Assignment 3

The assignment 3 is based on to different ways of implementing hashing, namely Double Hashing and Separate Chaining. The provided interfaces namely Student_ and MyHashTable_ are stored in separate files Student_.java and MyHashTable_.java respectively. The class Assignment3 contains the main method. The following classes are implemented:

1.Pair<K1,K2>

- The **generic pair class** is an auxiliary class which takes in two values of **type K1 and K2** and creates a **Pair** object of that type. It also contains a **boolean** variable **recentlyEmpty** which is used in the implementation of Double Hashing. The **toString()** method returns the **concatenation of the toString() representation of each of the K1 and K2 type objects** which are stored in the object. Other than these there are **getter and setter methods** for each of the attributes of the class. As all the attributes are declared as private. The compareTo(Pair<K1,K2> k) returns an integer which is the result of the comparison of the toString() representation of the object in arguments and the present object.

2. **Student.java**

- The **class Student** is used for creating new **Student_** type objects. The arguments which are passed to the constructor are **first name(first), last name(last), hostel(hostel), department(dept) and cgpa(cgpa) which are all strings**. All of these variables are stored in the class and along with this there are two additional **Pair<String,String> objects namely keyDH and keySCBST**. These are formed within the constructor of the Student and the arguments passed are **firstname,lastname for keyDH** and **first+“ ”, last** for **keySCBST**. There are **getter and setter for each of the attributes** of the class and the **toString()** representation of the Student object would be the **concatenation of first“ ”+last+“ ”+department“ ”+cgpa**.

3.DoubleHashing<K,T>.java

- The **class DoubleHashing<K,T>** implements the interface **MyHashTable_<K,T>.** It stores an **integer** variable **size** which is passed in the constructor and an array **arr** of **Pair<K,T>** is created which stores the **key-value pair** at a particular index.

Param Khakhar
2018CS10362

Assignment 3

- The method **insert(K key, T s)** inserts an external object of type **T** to the index which is calculated by the hash function. The arguments to the hash function is the **size** of the array and the string **full name concatenated with the last name** which is obtained by invoking the toString() method of the key and hence keeping the code generic. A while loop runs which inserts the key into the index calculated by the hash function if the array slot is either **null** or **recently empty**. The integer variable **calc** stores the number of times the hash function is calculated and is incremented with each unsuccessful iteration(unsuccessful insertion) of the loop.The loop terminates if the **boolean variable inserted becomes true** or when the number of calculations performed exceeds the size of the array, i.e. when the array becomes full.
- The following details were obtained while simulating insertion for different sizes of inputs and different hash table size.
- Theoretically, the average time complexity for uniform hashing and load factor < 1 can be calculated by analysing in terms of random variable X which denotes the number of probes made in an unsuccessful search. Let Ai denote the event that the ith probe occurs to an unoccupied slot. Then the event X >= i is intersection of A1, A2, A3, ….Ai-1. The probability of X >= i would be

  Where load factor(LF) = n/m < 1
  P(X>=i) = (n/m)*(n-1/m-1)*(n-2/m-2)*....(n-i+2/m-i+2) <= (n/m)^(i-1)

The expectation of the above random variable turns out to be 1/1-LF. The average number of probes while inserting would be the average of unsuccessful probes + 1 and hence the average number of probes for
Insertion would be big-theta(2-LF/1-LF).

Simulation of the DH implemented on various table sizes and number of insertions yielded the following results

Param Khakhar
2018CS10362

Assignment 3

HashTable size	Number of insertions	Total hash calculations	Max hash calculation	Min hash calculation	Load factor	Avg Hash calculations
101	35	48	4	1	0.35	1.37
101	45	73	7	1	0.45	1.622
101	55	86	7	1	0.55	1.56
101	65	110	7	1	0.65	1.69
101	75	158	10	1	0.75	2.1
101	85	200	13	1	0.85	2.36
401	85	100	3	1	0.21	1.33
701	85	90	2	1	0.12	1.2
1003	85	92	2	1	0.085	1.23

The average number of probes or hash function calculations is approximately constant(around 1) and is consistent with the theoretical expectations.

- The method **update(K key,T s)** updates the **key** to a new value **T s** if the key is already present and returns the number of calculation of hash functions or in other words number of probes made until the required key is found. It is implemented using a while loop which terminates when the number of probes are greater than the size of the table in case the table is full and the value isn't present or when the key encounters a key,value pair which is recently empty implying the value which needs to be updated no longer exists. The **boolean variable found** is used in the conditional block of the while loop to control its execution whereas another **boolean variable exist** is used to confirm the presence of the key-value pair for which the value needs to be updated. If in a particular iteration the desired key isn't found, a new index is calculated for probing and the **integer variable calc** which stores the number of hash calculations is incremented by one. If the desired key is found and isn't recently empty, the new value replaces the older value.

Param Khakhar
2018CS10362

Since the **update()** method may try to **update the key which is not already present** it's **time complexity is bounded from above by the time required for an unsuccessful search** and according to the analysis of the time complexity in the insert method, that would be **big-theta(1/1-LF).**

- The method **contains(K key)** uses the same algorithm as update but here we are only looking for the key in the hash table. If present the **boolean variable exist** is assigned true and returned accordingly, and if the key doesn't exist, or if the key-value pair is recently empty, the boolean variable **exist** is assigned false and returned accordingly outside the loop. Again the number of executions of while loop are bounded by the size of the array (i.e. maximum possible probes made and the fact that the key is not present).
  Since the **contains()** method may try to look for a key which is not already present it's **time complexity** is **bounded from above** by the **time required for an unsuccessful search** and according to the analysis of the time complexity in the insert method, that would be **big-theta(1/1-LF).**

- The method **address(K key)** uses the same algorithm as the **contains(K key)** method but an additional **integer variable index** is used which stores the **most recently calculated index** and gets updated with the progress of the loop. The method throws a **NotFoundException** if the desired key isn't present in the hash table which is determined by the **boolean variable exist** and if it's present, the most recently calculated index is returned.
  Since the **address()** method may try to look for a key which is not already present it's **time complexity** is **bounded from above** by the **time required for an unsuccessful search** and according to the analysis of the time complexity in the insert method, that would be **big-theta(1/1-LF).**

- The method **get(K key)** uses the same algorithm as update but here we are only looking for the key in the hash table. If present the **boolean variable exist** is assigned true and returned accordingly, and if the key doesn't exist, or if the key-value pair is recently empty, the boolean variable **exist** is assigned false and returned accordingly outside the loop. Again the number of executions of while

Param Khakhar
2018CS10362

loop are bounded by the size of the array (i.e. maximum possible probes made and the fact that the key is not present). If the element is present, it is stored in the **variable s which is of type T(for generic implementation)** and it is **initialized as null.** If s is not null it is returned otherwise, a **new NotFoundException** is thrown. Since the **get()** method may try to look for a key which is not already present it's **time complexity** is **bounded from above** by the **time required for an unsuccessful search** and according to the analysis of the time complexity in the insert method, that would be **big-theta(1/1-LF).**

- The method **delete(K key)** determines the presence of the required **key-value** pair using the same algorithm of **contains(K key).** If the **key-value pair is found,** the **recently empty attribute of the pair is set to be true and the attribute k2 of the pair is set to be null by the setter method setSecond().**
  Since the **delete()** method may try to look for a key which is not already present it's **time complexity** is **bounded from above** by the **time required for an unsuccessful search** and according to the analysis of the time complexity in the insert method, that would be **big-theta(1/1-LF).**

4. BNode.java

- The class **BNode<K,T>** implements a generic node and stores the variables **private BNode<K,T> left, private BNode<K,T> right, private BNode<K,T> parent, private T value, and private K key.**
- Since all the variables are private , **getter and setter methods** are implemented for each of them.The key which is used here would be **Pair<first+" ",last>,** the compareTo() method stores the string representation of the key in the node and key of the node in the argument which is done by using the toString() method on each of the keys. Then, the **split()** method on the string representation of the keys returns an **array** which separately contains the **first name and the last name** of the Student(in this case). Then the first names of both the keys are compared and returned as the answer.
- The method **toString()** returns the **toString()** of the stored key.

5.BinarySearchTree.java

Param Khakhar
2018CS10362

Assignment 3

- The variable **BNode<K,T> root** is maintained and is initialized to null when the tree is constructed. Since the variable is private, there are **getter and setter** methods to change this.
- The method **search(K k)** searches the binary search tree for the **presence of key k which is of type K**. The search begins at the root node. Since the expected type of key to be compared here is a **Pair<first“”,last>**, the **toString() method is called for the key k** in the arguments and then using the **split()** method on the strings **returns an array of length 2** which contains the **first name and the last name**. Two arrays **arr1(first name,last name)** of the required **key k** and **arr2(first name,last name) of the key of the current node** is stored. A **while loop** is used to implement the process and which would get **terminated when either the current node current node looks for the key in the children of the leaves or when the required key is found i.e. both the first name as well as the last name of the keys which are stored as strings in arr1 and arr2 match. In order to compare the strings, the compareTo() method** is used. Within the loop if the value of the required key is less than the key of the current node, then the current node is assigned to its left child and again the iteration continues. If the current key is smaller than the required key, then the current node is assigned to the right child of the current key. If both the keys happen to be the same (the case when there are entries with same first names) the current node is assigned then assigned to its right child because the cases when there are same first names are conventionally inserted to the right of the current node and this **goes on until both the first as well as the last name match**. The array arr2 containing the first name and the last name of the node is updated accordingly at the end of each iteration.
  Since at each node(current node) we are making a choice between the left and right Children of a particular node, and this procedure can take place from the root to all the way down to the leaves, the search turns out to be
  Big-Oh(height of the tree).

- The method **insert(BNode<K,T> b)** inserts a binary node at a suitable position in the BST. It first checks whether the node is present or not by calling **search(). If the key is already present then it returns the node and if not it returns null. If null is returned only then the node is inserted.** The nodes’ **compareTo()** is used

Param Khakhar
2018CS10362

to make comparisons among the current node and the **BNode<K,T> b**. The insert method uses the same algorithm as the **search(K k)** method and the tree is traversed until we find a suitable position where the **BNode<K,T>b** should be placed. The search process starts all the way from the **root** and the BNode<K,T> is inserted as a child of a suitable leaf. Assigning involves changing the left/right pointers of the leaf and changing the parent of the **BNode<K,T> b. The method also considers the case when insertion takes place at the root.**
Since we're searching for a suitable position for insertion, the time complexity of this operation would be the same as that for the search which is **Big-Oh(height of the tree). An integer variable count is used to maintain the number of nodes which are touched. In case of insertion, the variable count is incremented with each iteration of the loop.**

- The method **searchAddress(K k)** uses the same algorithm for the **search() method** but it maintains a string which stores the direction of traversal. In the **while loop of the search method, if the current node gets assigned to its left child, the letter 'L' is added to the string whereas the letter 'R' is added when the current node gets assigned to its right child.**
  The complexity of **searchAddress() would be the same as that of search() which is Big-Oh(height of the tree).**

- The method **Transplant(BNode<K,T> n1,BNode<K,T> n2)** interchanges the nodes **n1 and n2** by **reassigning the parent attribute of the n2 to the parent of n1 and reassigning the left or right attribute of the parent of node n1 to n2**. This is an auxiliary method which is used within the **delete() method.**

- The **delete(K k) method** first searches the BST for the presence of the key k. If the key is not present in the tree, -1 is returned. There are 4 cases which are separately dealt within the delete method. An integer variable count is maintained which counts the number of nodes touched during the deletion process.
  While doing a search the count is incremented with each iteration of the while loop.

Param Khakhar
2018CS10362

Assignment 3

1. If the node to be deleted has no child, then simply the required node's parent's right or left reference is set null accordingly.
2. If the node has either a left child or a right child (not both), then the child is promoted to the position of the parent and this is achieved by transplanting the node to be deleted and its only child. Since we're changing the references of the child node, the count gets incremented by 1 within this block.

3. If the node has both a left child as well as a right child, then the successor of the root is found out by going right once from the node to be deleted and finding the left most node in the subtree rooted at the node which is to the right of the node to be deleted. The successor is obtained by iterating through a while loop until the left attribute of the current node becomes null. The variable count is incremented for each of the node traversed while finding the successor. When the successor is obtained, two cases arise

    I. If the successor is the right child of the node to be deleted. In that case, the parents of the node to be deleted have their attributes changed to point to the successor keeping the rest of the things unchanged. The successor's left attribute is set to the left of the node to be deleted and this left subtree's parents are updated to refer to the successor.

    II. If the successor is a node other than the right node of the node to be deleted. In that case, the successor and its right node are interchanged and the successor's right is updated to the right of the node to be deleted. Also the parent reference of the node to the right of the node to be deleted are updated to point to the successor. Then all the steps of case I are performed for updating the left side of the node to be deleted.

Since while doing a delete we first find the element to be deleted, the time required to delete an element

Param Khakhar
2018CS10362

would be the same as that for making a search. Therefore the time complexity for deleting an element

Would be **Big-Oh(height of the tree).**

6.SeparateChaining<K,T>.java

- The class SeparateChaining<K,T>.java implements the already provided interface MyHashTable_<K,T>. An array of type **Pair<K,BinarySearchTree<K,T>>.The size of the array is passed while creating a SeparateChaining<K,T> object.**The array is created in the constructor.

- The method **insert(K key, T object)** first computes the index of the array where the insertion needs to be made. Since the key obtained here is of the type **Pair<first+“”,last>** , the **toString()** representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation** and **if the index of the array is null, a new BST is created and the node is inserted by calling the insert() method of the BST**. If the tree already exists, then only a call to the insert() method of the BST is made. The number of counts which are returned by the insert() method of the BST are returned by this method.

- The method **update(K key, T object) first computes the index at which an update needs to be made.** Since the key obtained here is of the type **Pair<first+“”,last>** , the **toString()** representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation.** Then a search is carried out to find the node with the required key by the same algorithm used by the **search() method in** BinarySearchTree<K,T>. **If the node is found then it’s value is updated by calling the setValue() method and if the key isn’t found, -1 is returned.**

- The method **delete(K key) first computes the index at which an update needs to be made.** Since the key obtained here is of the type **Pair<first+“”,last>** , the

toString() representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation. Then a call to the delete() method of the BST is made.**

- The method **contains(K key) first computes the index at which an update needs to be made.** Since the key obtained here is of the type **Pair<first+“”,last>** , the **toString()** representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation. Then a call to the method search(K key) of the BST is made. If the result of the search() is null, then false is returned and if not then true is returned.**

- The method **get(K key) first computes the index at which an update needs to be made.** Since the key obtained here is of the type **Pair<first+“”,last>** , the **toString()** representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation. Then a call to the search(K key) of the BST is made which returns BNode<K,T> if the key is present and null if the key isn’t present. If the key is present then the value of the key is returned and if the key isn’t present, a NotFoundException() is thrown.**

- The method **address(K key) first computes the index at which an update needs to be made.** Since the key obtained here is of the type **Pair<first+“”,last>** , the **toString()** representation of the Pair is used to get the string representation and then **split()** method is used to **isolate first name and last name and store them in an array.** The **index** is calculated by **calculating the hash of concatenation. Then a call to the BSTs searchAddress(key) is made which is a string. If the string isn’t null, a the concatenation of the computed index and string is returned otherwise a new NotFoundException() is thrown.**

7. Assignment3.java

Param Khakhar
2018CS10362

Assignment 3

- This is the class which contains the main method. Based on the command line inputs appropriate objects either DoubleHashing or SeparateChaining are created. The input queries are read from a file by using a BufferedReader. If the query is
  - **I. Insert a new Student object is created and is inserted accordingly to the DoubleHashing or SeparateChaining.**
  - **II. For update, respective calls to the update are made for DoubleHashing and SeparateChaining by creating a new Student object. If the student is not present an 'E' is printed.**
  - **III. For delete, respective calls to the delete() are made for DoubleHashing and SeparateChaining by creating a new Pair object(key) which is different for both the cases. If the student is not present an 'E' is printed.**
  - **IV. For get, respective calls to the get() are made for DoubleHashing and SeparateChaining by creating a new Pair object(key) which is different for both the cases. If the student is not present an 'E' is printed.**
  - **V. For contains(), respective calls to the contains() are made for DoubleHashing and SeparateChaining by creating a new Pair object(key) which is different for both the cases. If the student is present 'T' is printed, if not 'F' is printed.**
  - **VI. For address() respective calls to the address() are made for DoubleHashing and SeparateChaining by creating a new Pair object(key) which is different for both the cases. If the student is not present an 'E' is printed.**

Param Khakhar
2018CS10362