GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework #6 Report

Berkan AKIN
171044073

1. SYSTEM REQUIREMENTS

Q1)

a)Creating a chained hashmap by implementing the KWHashMap interface. The hashmap to be created will be used with binary tree search. Each row of the table will be a tree. If the same index appears, it will be added to the nodes of the tree. The methods of the KWHashMap interface in the book will be implemented. Likewise, the binary search tree will be modified to hold key and value.

b)A chain hashmap will be made using the chaining technique for hashing and hashing technique that is a combination of the double hashing methods. Research about coalesced hashing and Research about double hashing than expalin. Written methods should be tested.

2. PROBLEM SOLUTION APPROACH

Regarding my system's requirements and problems, i created a container class to keep and modift the data easily. Then I was able to set up a hierarchy and find a solution, by correctly determining the class relationships and the ease provided by my container class.

PROBLEM SOLUTION APPROACH My Problem solution steps are;

1. – Specify the problem requirements
2. – Analyze the problem
3. – Design an algorithm and Program
4. – Implement the algorithm
5. – Test and verify the program
6. – Maintain and update the program

I create and test each class separately. Then I combine these classes according to the problem and test them.

3. TEST CASES

1.1 BSTHashMap Class

if table is empty than return true.

```
/** Returns true if empty */
public boolean isEmpty() {
  return numKeys == 0;
}
```

Put Element in table

```
  public V put(Comparable key, Comparable comparable) {
    int index = key.hashCode() % table.length;
    if (index < 0)
      index += table.length;
    if (table[index] == null) {
      // Create a new linked list at table[index].
      table[index] = new BinarySearchTree < K, V > ();
    }

    table[index].insert(key,comparable);

    numKeys++;
    if (numKeys > (LOAD_THRESHOLD * table.length))
      rehash();
    return null;
  }
```

When the size of the table reaches a certain ratio, it increases its size.

```
public void rehash() {
  // Save a reference to oldTable
    BinarySearchTree  [] oldTable = table;
  // Double capacity of this table
  table = new BinarySearchTree[2 * oldTable.length + 1];

  // Reinsert all items in oldTable into expanded table.
  numKeys = 0;
  for (int i = 0; i < oldTable.length; i++) {
    if (oldTable[i] != null) {
      while(oldTable[i].root != null) {
        // Insert entry in expanded table
        put(oldTable[i].root.key,oldTable[i].root.value);
        oldTable[i].deleteKey(oldTable[i].root.key);
      }
    }
  }
}
```

Deletes data from binary search tree

```
/**** BEGIN EXERCISE ****/
public V remove(K key) {
  int index = key.hashCode() % table.length;
  if (index < 0)
    index += table.length;
  if (table[index] == null)
    return null; // Key not in table
  table[index].deleteKey((Comparable) key);


  return null; // Key not in table
}
```

Return element number of table

```java
/** Returns the number of entries in the map */
public int size() {
  return numKeys;
}
```

1.2 DoubleHash Class

Calculate hash1 value.

```java
private int hash1(K key) {
    //System.out.println("key: "+key +"  hashCode:" + key.hashCode());
    return key.hashCode()%table.length;
}
```

Calculate hash2 value.

```java
private int hash2(K key) {
    int primeNum = primeNumber((int) Math.round(table.length*0.8));
    //System.out.println("PrimeNumber: " +primeNum);
    return primeNum - (key.hashCode()%primeNum);
}
```

Calculate Double hash value.

```java
private int hashFunction(K key,int i) {
    int hash1 = hash1(key);
    int hash2 = hash2(key);
    return (hash1+(i*hash2))%table.length;
}
```

Put Key and value in table.

```java
@Override
public V put(K key, V value) {
    int index = hashFunction(key,0);
    int i=0,fullIndex;

    double loadFactor =
            (double) (numKeys + numDeletes) / table.length;
        if (loadFactor > LOAD_THRESHOLD)
          rehash();

    if(table[index] == null || table[index] == DELETED ) {
        table[index] = new Entry(key,value);
        numKeys++;
        return value;
    }
    else {
        while(i<table.length) {
            i++;
            fullIndex = index;
            index = hashFunction(key,i);
            if(table[index] == null || table[index] == DELETED) {
                table[index] = new Entry(key,value);
                table[fullIndex].index = index;
                table[fullIndex].next = table[index];
                numKeys++;
                return value;
            }
        }

    }
```

Get value with given key

```java
@Override
public V get(K key) {
    int index = hashFunction(key,0)%table.length;
    if(table[index].key.equals(key)) {
        return table[index].value;
    }
    else {
        int tmpIndex;
        while(table[index].index != -1) {
            index = table[index].index;
            if(table[index].key.equals(key)) {
                return table[index].value;
            }

        }
        return null;
    }
}
```

When the size of the table reaches a certain ratio, it increases its size.

```java
private void rehash() {
    // Save a reference to oldTable.
    Entry < K, V > [] oldTable = table;
    // Double capacity of this table.
    table = new Entry[2 * oldTable.length + 1];

    // Reinsert all items in oldTable into expanded table.
    numKeys = 0;
    numDeletes = 0;
    for (int i = 0; i < oldTable.length; i++) {
      if ( (oldTable[i] != null) && (oldTable[i] != DELETED)) {
        // Insert entry in expanded table
        put(oldTable[i].key, oldTable[i].value);
      }
    }
}
```

Deletes data from the table

```
    */
  public V remove(K key) {
      int index = hashFunction(key,0)%table.length;
        if(table[index].key.equals(key)) {
            V oldValue = table[index].value;
            table[index] = DELETED;
            numKeys--;
            return oldValue;
        }
        else {
            int fullIndex;
            while(table[index].index != -1) {
                fullIndex = index;
                index = table[index].index;
                if(table[index].key.equals(key)) {
                    V oldValue = table[index].value;
                    table[index] = DELETED;
                    numKeys--;
                    table[fullIndex].index =-1;
                    table[fullIndex].next = null;
                    return oldValue;
                }

            }
        }
        return null;
    }
```

Returns the largest prime number less than the parameter

```
public int primeNumber(int n)
{

    // All prime numbers are odd except two
    if (n % 2 != 0)
        n -= 2;
    else
        n--;

    int i, j;
    for (i = n; i >= 2; i -= 2) {
        if (i % 2 == 0)
            continue;
        for (j = 3; j <= Math.sqrt(i); j += 2) {
            if (i % j == 0)
                break;
        }
        if (j > Math.sqrt(i))
            return i;
    }

    // It will only be executed when n is 3
    return 2;
}
```

Print all Table value;

```
public void printTable() {
    int i=0;

    for(i=0;i<table.length;i++) {
        if(table[i] != null) {
            System.out.println(i  + "   " + table[i].key + "/"+ table[i].value
        }
        else {
            System.out.println(i  + "   " + "null" +"   " + "null");
        }


    }
}
```

4. RUNNING AND RESULTS

Q1.1 Test And Result

Test Code	Function Name	Expected Result	Actual Result
fr_01	`put(Comparable key, Comparable comparable)`	Put key and value in bst tree	Pass
fr_02	`V remove(K key)`	Remove value with given key	Pass
fr_03	`void rehash()`	Increasing the size of the table	Pass
fr_04	`V get(K key)`	Get value with given key	Pass

fr_05	`printTable()`	Print all element in table	Pass

Q1.2 Test And Result

Test Code	Function Name	Expected Result	Actual Result
fr_01	`V put(K key, V value)`	Put key and value in table with double hashing and coalesced hashing techniques	Pass
fr_02	`V get(K key)`	Get value with given key	Pass
fr_03	`hash1(K key)`	Calculate hash1 value	Pass
fr_04	`int hash2(K key)`	Calculate hash2 value	Pass
fr_05	`int hashFunction(K key,int i)`	Calculate index	Pass
fr_06	`int primeNumber(int n)`	Returns the largest prime number less that the parameter	Pass
fr_07	`void rehash()`	increases its size of table	Pass
fr_08	`V remove(K key)`	Remove value and key with given key	Pass
fr_09	`printTable()`	Print all element in table	Pass

Q1.2.a

The coalesced hashing technique tries to minimize the itchy bump from the collision. It provides this with open addressing and sparete chaing.

Advanteges:

If the chain is short and simple, this technique implementation is good because we can access data quickly.

Disadvanteges:

costly to delete and resize

Q1.2.b

A technique used to avoid collisions for open addressing.

Advanteges:

advantageous in preventing collisions because more hash function can defeat the cluster.

Disadvantages:

The prime number finding algorithm is very inefficient. Time complexity is O(n^2)