**ASSIGNMENT 3**

**Level 3 Computer Science**

**CSC3021 Concurrent Programming 2016-2017**

**Question 1: Lock-based synchronization**

*/\*\*  
 \* I have included the Hash class to answer question one.   
 \* To enable a coarse-grained solution I made sure that the Hash class was thread-safe.   
 \* This provides a more restrictive access to shared resources and is known to be a sequential bottleneck.  
 \* I have opted to use the synchronized key word to make the object thread-safe.  
 \* This means calling threads will have to acquire the objects intrinsic lock (instance lock) before any progression.  
 \* Once a method call is finished, the thread who has acquired the lock will release it for another to use.  
 \* The subsequent thread will then be able to view the updated state or true state of the object.  
 \** ***@param*** <*K*> *This is a Key. It will be used as an identifier for a value in the chain.  
 \** ***@param*** <*V*> *This is the value. It will be associated with a key in the chain.  
 \*/*public class Hash<K, V> {  
 private int num\_buckets;  
 private Chain<K, V>[] buckets;  
  
 public Hash(int num\_buckets\_) {  
 num\_buckets = num\_buckets\_;  
 buckets = (Chain<K, V>[]) new Chain[num\_buckets];  
 for (int i = 0; i < num\_buckets; ++i)  
 buckets[i] = new Chain<K, V>();  
 }  
 public synchronized int getArraySize() {  
 return num\_buckets;  
 }  
 public synchronized void resize(int new\_num\_buckets) {  
 Chain<K, V>[] old\_buckets = buckets;  
 int old\_num\_buckets = num\_buckets;  
  
 num\_buckets = new\_num\_buckets;  
 buckets = (Chain<K, V>[]) new Chain[num\_buckets];  
 for (int i = 0; i < num\_buckets; ++i)  
 buckets[i] = new Chain<K, V>();  
  
 // Iterate and rehash  
 for (int i = 0; i < old\_num\_buckets; ++i) {  
 Chain<K, V>.Iterator iter = old\_buckets[i].iterator();  
 while (iter.hasNext()) {  
 Chain<K, V>.KeyValue kv = iter.next();  
 add(kv.key, kv.value);  
 }  
 }  
 }  
 private int bHash(int hash) {  
 return Math.*abs*(hash % num\_buckets);  
 }  
 public synchronized boolean add(K key, V value) {  
 int bhash = bHash(key.hashCode());  
 return buckets[bhash].add(key, value);  
 }  
 public synchronized V get(K key) {  
 int bhash = bHash(key.hashCode());  
 return buckets[bhash].get(key);  
 }  
 public synchronized boolean remove(K key) {  
 int bhash = bHash(key.hashCode());  
 return buckets[bhash].remove(key);  
 }  
 public synchronized int size() {  
 int size = 0;  
 for (int i = 0; i < num\_buckets; ++i)  
 size += buckets[i].size();  
 return size;  
 }  
}

**Question 2: Lock-based synchronization, again**

I have included the Chain class to demonstrate coarse-grained locking of single hash chains. The Hash class is unchanged.

import java.util.Iterator;  
import java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
 \* I have included only the Chain class to answer question 2.  
 \* I have enabled locking of each distinct chain within the hashtable. This will allow the data structure to have  
 \* greater parallelism and less lock contention.  
 \*/*class Chain<K, V> {  
 private Node head;  
 private ReentrantLock lock = new ReentrantLock();  
  
 public class KeyValue {  
 public K key;  
 public V value;  
 }  
  
 // This iterator is only required for Q6 and  
 // may otherwise be ignored.  
 public class Iterator {  
 private Chain<K, V>.Node cur;  
 private Chain<K, V> list;  
  
 public Iterator(Chain<K, V> list\_) {  
 list = list\_;  
 cur = list.head;  
 }  
  
 public boolean hasNext() {  
 return cur.next != null  
 && cur.next.hash != java.lang.Integer.*MAX\_VALUE*;  
 }  
  
 public KeyValue next() {  
 KeyValue kv = new KeyValue();  
 cur = cur.next;  
 kv.key = cur.key;  
 kv.value = cur.value;  
 return kv;  
 }  
  
 void remove() {  
 throw new UnsupportedOperationException();  
 }  
 }  
  
 public Iterator iterator() {  
 return new Iterator(this);  
 }  
  
 private class Node {  
 int hash;  
 K key;  
 V value;  
 Node next;  
  
 public Node(int hash) {  
 this.hash = hash;  
 this.key = null;  
 this.value = null;  
 this.next = null;  
 }  
  
 public Node(int hash, K key, V value) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 }  
 }  
 public Chain() {  
 Node tail = new Node(java.lang.Integer.*MAX\_VALUE*);  
 head = new Node(java.lang.Integer.*MIN\_VALUE*);  
 head.next = tail;  
 }  
  
 // Insert value for key.  
 public boolean add(K key, V value) {  
 Node pred, curr;  
 // Require key != null and value != null  
 // Get hash code  
 int hash = key.hashCode();  
 lock.lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 if (hash == curr.hash) {  
 return false;  
 } else {  
 Node node = new Node(hash, key, value);  
 node.next = curr;  
 pred.next = node;  
 return true;  
 }  
 } finally {  
 lock.unlock();  
 }  
 }  
 // Lookup value for key  
 public V get(K key) {  
 Node pred, curr;  
 int hash = key.hashCode();  
 lock.lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 return (hash == curr.hash) ? curr.value : null;  
 } finally {  
 lock.unlock();  
 }  
 }  
 // Remove key/value pair  
 public boolean remove(K key) {  
 // Require key != null  
 Node pred, curr;  
 int hash = key.hashCode();  
 lock.lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 if (hash == curr.hash) {  
 pred.next = curr.next;  
 return true;  
 } else {  
 return false;  
 }  
 } finally {  
 lock.unlock();  
 }  
 }  
 public int size() {  
 int size = 0;  
 Node n = head.next;  
 while (n.next != null) {  
 size++;  
 n = n.next;  
 }  
 return size;  
 }  
}

**Question 3: Readers/writers lock**

I have provided two solutions to this problem (Could have used a similar technique with Q2). Both work the same way only they are implemented in different classes. The first solution uses an array of Reader/Writers locks in the Hash class leaving the Chain class unchanged. The second solution modifies only the chain class with the use of Readers/Writers lock. The Hash class left unchanged. Note: *the second solution is the version used to record results in the experiments.*

*First:*

import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
*/\*\*  
 \* I have included the Hash class only to answer question 3. In keeping with the fine-grained approach in question 2,  
 \* we want to enhance this with the addition of Reentrantlocks.  
 \* The readLock may be held simultaneously by multiple reader threads. Only if there are no writers at this point.  
 \* The writerLock is exclusive.  
 \* We have an array of Reentrantlocks because we want to have a writers and readers lock for each chain.  
 \** ***@param*** <*K*> *This is a Key. It will be used as an identifier for a value in the chain.  
 \** ***@param*** <*V*> *This is the value. It will be associated with a key in the chain.  
 \*/*public class Hash<K,V> {  
 private int num\_buckets;  
 private Chain<K,V>[] buckets;  
 private final ReentrantReadWriteLock[] rwlock;  
  
 public Hash(int num\_buckets\_) {  
 num\_buckets = num\_buckets\_;  
 buckets = (Chain<K,V>[]) new Chain[num\_buckets];  
 rwlock = new ReentrantReadWriteLock[num\_buckets];  
 for( int i=0; i < num\_buckets; ++i ) {  
 buckets[i] = new Chain<K, V>();  
 rwlock[i] = new ReentrantReadWriteLock();  
 }  
 }  
  
 public int getArraySize() {  
 return num\_buckets;  
 }  
 public void resize( int new\_num\_buckets ) {  
 Chain<K,V>[] old\_buckets = buckets;  
 int old\_num\_buckets = num\_buckets;  
  
 num\_buckets = new\_num\_buckets;  
 buckets = (Chain<K,V>[]) new Chain[num\_buckets];  
 for( int i=0; i < num\_buckets; ++i )  
 buckets[i] = new Chain<K,V>();  
 // Iterate and rehash  
 for( int i=0; i < old\_num\_buckets; ++i ) {  
 Chain<K,V>.Iterator iter = old\_buckets[i].iterator();  
 while( iter.hasNext() ) {  
 Chain<K,V>.KeyValue kv = iter.next();  
 add( kv.key, kv.value );  
 }  
 }  
 }  
 private int bHash( int hash ) {  
 return Math.*abs*(hash % num\_buckets);  
 }  
 */\*\*  
 \* Allows me to obtain the index of the Reentrantlock to be acquired from the corresponding hashcode.  
 \* We have an array of Reentrantlocks because we want to have a writers and readers lock for each chain.  
 \* If the hashtable is re-sized to double its size. The lock array will stay the same size but still  
 \* cover every entry in the chain array. (Not concerned with re-sizing in this question.)  
 \** ***@param*** *hash - hash code of key  
 \** ***@return*** *- index of lock  
 \*/* private int lHash( int hash ) {  
 return Math.*abs*(hash % rwlock.length);  
 }  
  
 public boolean add( K key, V value ) {  
 int keyHash = lHash(key.hashCode());  
 rwlock[keyHash].writeLock().lock();  
 try {  
 int bhash = bHash(key.hashCode());  
 return buckets[bhash].add( key, value );  
 }finally {  
 rwlock[keyHash].writeLock().unlock();  
 }  
 }

public V get( K key ) {  
 int keyHash = lHash(key.hashCode());  
 rwlock[keyHash].readLock().lock();  
 try {  
 int bhash = bHash( key.hashCode() );  
 return buckets[bhash].get( key );  
 }finally {  
 rwlock[keyHash].readLock().unlock();  
 }  
 }  
 public boolean remove( K key ) {  
 int keyHash = lHash(key.hashCode());  
 rwlock[keyHash].writeLock().lock();  
 try {  
 int bhash = bHash( key.hashCode() );  
 return buckets[bhash].remove( key );  
 }finally {  
 rwlock[keyHash].writeLock().unlock();  
 }  
 }  
 public int size() {  
 int size = 0;  
 for( int i=0; i < num\_buckets; ++i )  
 size += buckets[i].size();  
 return size;  
 }  
}

*Second:*

import java.util.Iterator;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
*/\*\*  
 \* I have included only the Chain class to answer question 2.  
 \* I have enabled locking of each distinct chain within the hashtable. This will allow the data structure to have  
 \* greater parallelism and less lock contention.  
 \*/*class Chain<K, V> {  
 private Node head;  
 private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();  
  
 public class KeyValue {  
 public K key;  
 public V value;  
 }  
  
 // This iterator is only required for Q6 and  
 // may otherwise be ignored.  
 public class Iterator {  
 private Chain<K, V>.Node cur;  
 private Chain<K, V> list;  
  
 public Iterator(Chain<K, V> list\_) {  
 list = list\_;  
 cur = list.head;  
 }  
  
 public boolean hasNext() {  
 return cur.next != null  
 && cur.next.hash != java.lang.Integer.*MAX\_VALUE*;  
 }  
  
 public KeyValue next() {  
 KeyValue kv = new KeyValue();  
 cur = cur.next;  
 kv.key = cur.key;  
 kv.value = cur.value;  
 return kv;  
 }  
  
 void remove() {  
 throw new UnsupportedOperationException();  
 }  
 }  
  
 public Iterator iterator() {  
 return new Iterator(this);  
 }  
  
 private class Node {  
 int hash;  
 K key;  
 V value;  
 Node next;  
  
 public Node(int hash) {  
 this.hash = hash;  
 this.key = null;  
 this.value = null;  
 this.next = null;  
 }  
  
 public Node(int hash, K key, V value) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 }  
 }  
 public Chain() {  
 Node tail = new Node(java.lang.Integer.*MAX\_VALUE*);  
 head = new Node(java.lang.Integer.*MIN\_VALUE*);  
 head.next = tail;  
 }  
  
 // Insert value for key.  
 public boolean add(K key, V value) {  
 Node pred, curr;  
 // Require key != null and value != null  
 // Get hash code  
 int hash = key.hashCode();  
 lock.writeLock().lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 if (hash == curr.hash) {  
 return false;  
 } else {  
 Node node = new Node(hash, key, value);  
 node.next = curr;  
 pred.next = node;  
 return true;  
 }  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
 // Lookup value for key  
 public V get(K key) {  
 Node pred, curr;  
 int hash = key.hashCode();  
 lock.readLock().lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 return (hash == curr.hash) ? curr.value : null;  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
 // Remove key/value pair  
 public boolean remove(K key) {  
 // Require key != null  
 Node pred, curr;  
 int hash = key.hashCode();  
 lock.writeLock().lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 if (hash == curr.hash) {  
 pred.next = curr.next;  
 return true;  
 } else {  
 return false;  
 }  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
 public int size() {  
 int size = 0;  
 Node n = head.next;  
 while (n.next != null) {  
 size++;  
 n = n.next;  
 }  
 return size;  
 }  
}

**Question 4: Hand-over-hand locking**

I have included the Chain class only to answer this question.

import java.util.Iterator;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
class Chain<K, V> {  
 private Node head;  
  
 public class KeyValue {  
 public K key;  
 public V value;  
 }  
  
 // This iterator is only required for Q6 and  
 // may otherwise be ignored.  
 public class Iterator {  
 private Node cur;  
 private Chain<K, V> list;  
  
 public Iterator(Chain<K, V> list\_) {  
 list = list\_;  
 cur = list.head;  
 }  
  
 public boolean hasNext() {  
 return cur.next != null  
 && cur.next.hash != Integer.*MAX\_VALUE*;  
 }  
  
 public KeyValue next() {  
 KeyValue kv = new KeyValue();  
 cur = cur.next;  
 kv.key = cur.key;  
 kv.value = cur.value;  
 return kv;  
 }  
  
 void remove() {  
 throw new UnsupportedOperationException();  
 }  
 }  
  
 public Iterator iterator() {  
 return new Iterator(this);  
 }  
  
 private class Node {  
 int hash;  
 K key;  
 V value;  
 Node next;  
 */\*\*  
 \* Lock provided for each individual node.  
 \*/* Lock lock;  
  
 public Node(int hash) {  
 this.hash = hash;  
 this.key = null;  
 this.value = null;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
  
 public Node(int hash, K key, V value) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
  
 */\*\*  
 \* Accessor methods for locking and unlocking nodes associated lock.  
 \*/* public void lock() {  
 lock.lock();  
 }  
 public void unlock() {  
 lock.unlock();  
 }  
 }  
  
 public Chain() {  
 /\*  
 Setting head to MAX and tail to MIN Integers because this will let us know if we are  
 at the end of the chain. We may need to add something or remove something.  
 \*/  
 Node tail = new Node(Integer.*MAX\_VALUE*);  
 head = new Node(Integer.*MIN\_VALUE*);  
 head.next = tail;  
 }  
  
 // Insert value for key.  
 public boolean add(K key, V value) {  
 // Require key != null and value != null  
 int hash = key.hashCode(); // hashCode of entry to be added  
 head.lock(); // lock the first node in the chain  
 Node pred = head; // set the first node to pred  
 try {  
 Node curr = pred.next; // set the next node to curr (this could be tail if nothing has been added yet)  
 curr.lock(); // lock the next node in the list (Again, as above!)  
 try {  
 while (curr.hash < hash) { //Are we at the position in the chain where we want to add the node? False = yes we are, True = no, continue  
 pred.unlock(); // we know we want to add there entry after curr at this point so we can unlock pred for another thread and move to acquiring the next lock for the node in the chain.  
 pred = curr;  
 curr = curr.next;  
 curr.lock();  
 }  
 if (hash == curr.hash) {  
 curr.value = value; //the key is present, update value.  
 return false;  
 }  
 Node node = new Node(hash, key, value);  
 node.next = curr;  
 pred.next = node;  
 return true;  
 } finally {  
 curr.unlock();  
 }  
 } finally {  
 pred.unlock();  
 }  
 }  
  
 // Lookup value for key  
 public V get(K key) {  
 // Require key != null  
 Node pred = null, curr = null;  
 // Get hash code  
 int hash = key.hashCode();  
 head.lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 curr.lock();  
 try {  
 while (curr.hash < hash) {  
 pred.unlock();  
 pred = curr;  
 curr = curr.next;  
 curr.lock();  
 }  
 if (hash == curr.hash) {  
 return curr.value;  
 }  
 return null;  
 } finally {  
 curr.unlock();  
 }  
 } finally {  
 pred.unlock();  
 }  
 }  
  
 // Remove key/value pair  
 public boolean remove(K key) {  
 Node pred = null;  
 Node curr = null;  
 // Require key != null  
 // Get hash code  
 int hash = key.hashCode();  
 head.lock();  
 try {  
 pred = head;  
 curr = pred.next;  
 curr.lock();  
 try {  
 while(curr.hash < hash) {  
 pred.unlock();  
 pred = curr;  
 curr = curr.next;  
 curr.lock();  
 }  
 if (hash == curr.hash) { // key present, update value  
 pred.next = curr.next; // remove reference  
 return true;  
 }  
 return false; //key not found  
 } finally {  
 curr.unlock();  
 }  
 } finally {  
 pred.unlock();  
 }  
 }  
  
 public int size() {  
 int size = 0;  
  
 Node n = head.next;  
 while (n.next != null) {  
 size++;  
 n = n.next;  
 }  
 return size;  
 }  
}

**Question 5: Resizable hash table (Bonus)**

To answer this question, I have included the *Chain.java* class (from question 4) and the modified *Hash.java* class. This corresponds to **‘*solution 1*’**.

In addition to this I have added another algorithm that has better performance but still has the same properties as required in the question. I used an optimistic synchronization technique in the chain class. This will reduce synchronization costs as it will search without unnecessarily acquiring locks. It will only lock on nodes that it needs to lock on. This corresponds to **‘*solution 2*’**.

Lastly, I included a third solution that shows even better performance than the second. This uses ‘Lazy synchronization’. It gives us more to talk about in experiment 2. This is ***‘solution 3’.***

*Q5 – Solution 1*

import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
*/\*\*  
 \* Question 5 Hash class  
 \*/*public class Hash<K,V> {  
 private volatile Chain<K,V>[] buckets;  
 private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();  
  
 public Hash(int num\_buckets\_) {  
 buckets = (Chain<K,V>[]) new Chain[num\_buckets\_];  
 for( int i=0; i < num\_buckets\_; ++i )  
 buckets[i] = new Chain<K,V>();  
 }  
  
 public int getArraySize() {  
 return getBucketArray().length;  
 }  
  
 */\*\*  
 \* This resize method takes advantage of the RCU (technique commonly used in the Linux kernel).  
 \* We ensure that no other write operations can be performed at the time of a resize, i.e. add(x), remove(x).  
 \* We create a synchronization edge between the reading operation and the resize operation. This is achieved by  
 \* declaring the 'buckets' variable as volatile. In doing this I enforce a happens-before relationship  
 \* to prevent the reordering. Undesirable reordering may cause a reading thread to suffer from reading the new  
 \* 'buckets' array (updated size) prematurely; before the array has actually been re-sized.  
 \** ***@param*** *new\_num\_buckets - the new size of the array (determined by driver)  
 \*/* public void resize( int new\_num\_buckets ) {  
 final ReentrantReadWriteLock lock = this.lock;  
 lock.writeLock().lock();  
 try {  
 Chain<K,V>[] bucketArrayToResize = getBucketArray();  
 int bucketArraySize = bucketArrayToResize.length;  
  
 Chain<K,V>[] newBucketArray = (Chain<K,V>[]) new Chain[new\_num\_buckets];  
 for( int i=0; i < new\_num\_buckets; ++i ) {  
 newBucketArray[i] = new Chain<K, V>();  
 }  
 // Iterate and rehash  
 for( int i=0; i < bucketArraySize; i++ ) {  
 Chain<K,V>.Iterator iter = bucketArrayToResize[i].iterator();  
 while( iter.hasNext() ) {  
 Chain<K,V>.KeyValue kv = iter.next();  
 int bhash = Math.*abs*(kv.key.hashCode() % newBucketArray.length);  
 newBucketArray[bhash].add(kv.key, kv.value);  
 }  
 }  
 /\*  
 Here I am setting the new buckets array to the global shared variable.  
 Any reads of the variable after this point will be sure to see the updated values/size.  
 Memory reclamation is handled by the GC so the old reference to 'buckets' will be disposed of  
 accordingly and we do not need to worry about future references to it.  
 \*/

setBucketArray(newBucketArray);  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
  
 private int bHash( int hash ) {  
 return Math.*abs*(hash % getArraySize());  
 }  
  
 public boolean add(K key, V value) {  
 final ReentrantReadWriteLock lock = this.lock;  
 lock.readLock().lock();  
 try {  
 int bhash = bHash(key.hashCode());  
 return buckets[bhash].add(key, value);  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
  
 */\*\*  
 \* Notice a reading thread will not be required to wait, and can continue its operations regardless of  
 \* concurrent threads updates.  
 \* Obtain a reference to the latest 'buckets' array. Achieved through use of volatile keyword.  
 \*/* public V get( K key ) {  
 Chain<K, V>[] current\_Array = getBucketArray();  
 int bhash = Math.*abs*(key.hashCode() % current\_Array.length);  
 return current\_Array[bhash].get( key );  
  
 }  
  
 public boolean remove( K key ) {  
 final ReentrantReadWriteLock lock = this.lock;  
 lock.readLock().lock();  
 try {  
 int bhash = bHash( key.hashCode() );  
 return buckets[bhash].remove( key );  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
  
 public int size() {  
 int size = 0;  
 for( int i=0; i < getArraySize(); ++i )  
 size += buckets[i].size();  
 return size;  
 }  
  
 public Chain<K,V>[] getBucketArray() {  
 return buckets;  
 }  
 public void setBucketArray(Chain<K,V>[] b) {  
 buckets = b;  
 }  
}

*Q5 – Solution 2 – Optimistic synchronization alternative (uses same hash class as Q5 Solution1)*

import java.util.Iterator;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
 \* Using optimistic synchronization, uses the same Hash class from solution 1 in question 5  
 \*/*class Chain<K, V> {  
 private Node head;  
  
 public class KeyValue {  
 public K key;  
 public V value;  
 }  
  
 // This iterator is only required for Q6 and  
 // may otherwise be ignored.  
 public class Iterator {  
 private Node cur;  
 private Chain<K, V> list;  
  
 public Iterator(Chain<K, V> list\_) {  
 list = list\_;  
 cur = list.head;  
 }  
  
 public boolean hasNext() {  
 return cur.next != null  
 && cur.next.hash != Integer.*MAX\_VALUE*;  
 }  
  
 public KeyValue next() {  
 KeyValue kv = new KeyValue();  
 cur = cur.next;  
 kv.key = cur.key;  
 kv.value = cur.value;  
 return kv;  
 }  
  
 void remove() {  
 throw new UnsupportedOperationException();  
 }  
 }  
  
 public Iterator iterator() {  
 return new Iterator(this);  
 }  
  
 private class Node {  
 int hash;  
 boolean marked;  
 K key;  
 V value;  
 Node next;  
 */\*\*  
 \* Lock provided for each individual node.  
 \*/* Lock lock;  
  
 public Node(int hash) {  
 this.hash = hash;  
 this.marked = false;  
 this.key = null;  
 this.value = null;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
  
 public Node(int hash, K key, V value) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
  
 */\*\*  
 \* Accessor methods for locking and unlocking nodes associated lock.  
 \*/* public void lock() {  
 lock.lock();  
 }  
 public void unlock() {  
 lock.unlock();  
 }  
 }  
  
 public Chain() {  
 /\*  
 Setting head to MAX and tail to MIN Integers because this will let us know if we are  
 at the end of the chain. We may need to add something or remove a node.  
 \*/  
 Node tail = new Node(Integer.*MAX\_VALUE*);  
 head = new Node(Integer.*MIN\_VALUE*);  
 head.next = tail;  
 }  
  
 // Insert value for key.  
 public boolean add(K key, V value) {  
 int hash = key.hashCode();  
 while (true) {  
 Node pred = head;  
 Node curr = head.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 pred.lock();  
 curr.lock();  
 try {  
 if (validate(pred, curr)) {  
 if (curr.hash == hash) {  
 return false;  
 } else {  
 Node node = new Node(hash, key, value);  
 node.next = curr;  
 pred.next = node;  
 return true;  
 }  
 }  
 } finally {  
 pred.unlock(); curr.unlock();  
 }  
 }  
 }  
  
 // Lookup value for key  
 public V get(K key) {  
 int hash = key.hashCode();  
 while(true) {  
 Node pred = this.head;  
 Node curr = pred.next;  
 while (curr.hash < hash) {  
 pred = curr; curr = curr.next;  
 }  
 pred.lock(); curr.lock();  
 try {  
 if (validate(pred, curr)) {  
 return curr.hash == hash ? curr.value : null;  
 }  
 } finally {  
 pred.unlock(); curr.unlock();  
 }  
 }  
 }  
  
  
 // Remove key/value pair  
 public boolean remove(K key) {  
 int hash = key.hashCode();  
 while (true) {  
 Node pred = head;  
 Node curr = head.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 pred.lock(); curr.lock();  
 try {  
 if (validate(pred, curr)) {  
 if (curr.hash == hash) {  
 pred.next = curr.next;  
 return true;  
 } else {  
 return false;  
 }  
 }  
 } finally {  
 pred.unlock(); curr.unlock();  
 }  
 }  
 }  
  
 private boolean validate(Node pred, Node curr) {  
 Node node = head;  
 while (node.hash <= pred.hash) {  
 if (node == pred) {  
 return pred.next == curr;  
 }  
 node = node.next;  
 }  
 return false;  
 }  
  
 public int size() {  
 int size = 0;  
  
 Node n = head.next;  
 while (n.next != null) {  
 size++;  
 n = n.next;  
 }  
 return size;  
 }  
}

*Q5 – Solution 3 – Lazy synchronization alternative (uses same hash class as Q5 Solution1, therefore has required properties as requested from Q5.)*

import java.util.Iterator;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
\* Using lazy synchronization so that contains() (get in this class) calls are wait-free, and add() and remove()  
\* methods, while still blocking, transverse the list/chain only once (in the absence of contention).  
\*/*class Chain<K, V> {  
 private Node head;  
  
 public class KeyValue {  
 public K key;  
 public V value;  
 }  
 // This iterator is only required for Q6 and  
 // may otherwise be ignored.  
 public class Iterator {  
 private Node cur;  
 private Chain<K, V> list;  
 public Iterator(Chain<K, V> list\_) {  
 list = list\_;  
 cur = list.head;  
 }  
 public boolean hasNext() {  
 return cur.next != null  
 && cur.next.hash != Integer.*MAX\_VALUE*;  
 }  
 public KeyValue next() {  
 KeyValue kv = new KeyValue();  
 cur = cur.next;  
 kv.key = cur.key;  
 kv.value = cur.value;  
 return kv;  
 }  
 void remove() {  
 throw new UnsupportedOperationException();  
 }  
 }  
 public Iterator iterator() {  
 return new Iterator(this);  
 }  
 private class Node {  
 int hash;  
 boolean marked;  
 K key;  
 V value;  
 Node next;  
 */\*\*  
 \* Lock provided for each individual node.  
 \*/* Lock lock;  
 public Node(int hash) {  
 this.hash = hash;  
 this.marked = false;  
 this.key = null;  
 this.value = null;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
 public Node(int hash, K key, V value) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 this.lock = new ReentrantLock();  
 }  
 */\*\*  
 \* Accessor methods for locking and unlocking nodes associated lock.  
 \*/* public void lock() {  
 lock.lock();  
 }  
 public void unlock() {  
 lock.unlock();  
 }  
 }  
 public Chain() {  
 /\*  
 Setting head to MAX and tail to MIN Integers because this will let us know if we are  
 at the end of the chain. We may need to add something or remove a node.  
 \*/  
 Node tail = new Node(Integer.*MAX\_VALUE*);  
 head = new Node(Integer.*MIN\_VALUE*);  
 head.next = tail;  
 }  
 // Insert value for key.  
 public boolean add(K key, V value) {  
 int hash = key.hashCode();  
 while (true) {  
 Node pred = head;  
 Node curr = head.next;  
 while (curr.hash < hash) {  
 pred = curr; curr = curr.next;  
 }  
 pred.lock();  
 try{  
 curr.lock();  
 try {  
 if (validate(pred, curr)) {  
 if (curr.hash == hash) {  
 return false;  
 } else {  
 Node node = new Node(hash, key, value);  
 node.next = curr;  
 pred.next = node;  
 return true;  
 }  
 }  
 } finally {  
 curr.unlock();  
 }  
 } finally {  
 pred.unlock();  
 }  
 }  
 }  
 // Lookup value for key  
 public V get(K key) {  
 int hash = key.hashCode();  
 Node curr = head;  
 while (curr.hash < hash) {  
 curr = curr.next;  
 }  
 return curr.hash == hash && !curr.marked ? curr.value : null;  
 }  
 // Remove key/value pair  
 public boolean remove(K key) {  
 int hash = key.hashCode();  
 while (true) {  
 Node pred = head;  
 Node curr = head.next;  
 while (curr.hash < hash) {  
 pred = curr;  
 curr = curr.next;  
 }  
 pred.lock();  
 try {  
 curr.lock();  
 try {  
 if (validate(pred, curr)) {  
 if (curr.hash != hash) {  
 return false;  
 } else {  
 curr.marked = true;  
 pred.next = curr.next;  
 return true;  
 }  
 }  
 } finally {  
 curr.unlock();  
 }  
 } finally {  
 pred.unlock();  
 }  
 }  
 }  
 private boolean validate(Node pred, Node curr) {  
 return !pred.marked && !curr.marked && pred.next == curr;  
 }  
 public int size() {  
 int size = 0;  
  
 Node n = head.next;  
 while (n.next != null) {  
 size++;  
 n = n.next;  
 }  
 return size;  
 }  
}

**Question 6: Evaluation and Discussion**

Measure the throughput of each of your algorithms for Q1-4 (and optionally Q5 if you have answered Q5) using the supplied driver.

|  |  |  |
| --- | --- | --- |
| Parameter | Experiment (a) | Experiment (b) |
| *CPU model, version,*  *number of cores* | MacBook Pro (13-inch, Late 2011)  2.4GHz dual-core Intel Core i5 processor  16GB RAM | |
| *Operating system*  *(name and version)* | OSX El Capitan - Version: 10.11.6 | |
| *JVM*  *(name and version)* | Java version “1.8.0\_102”  Java™ SE Runtime Environment (build 1.8.0\_102-b14)  Java HotSpot™ 64-Bit Server VM (build 25.102-b14, mixed mode) | |
| *<threads>* | 1-32 | 6 |
| *<size>* | 300,000 | 300,000 |
| *<contains>* | 18 | 33%-98% (1..98) |
| *<dly>* | 1000 (1 second) | 1000 (1 second) |
| *<rounds>* | 10 | 10 |

**Experiments:**

a)

*Vary the number of threads from 1 to about 32 to measure the throughput of each of your implementations. You may set <rounds> to at least 5, select 90% lookup operations (corresponding to <contains> = 18) and set <dly> to 1 second.*

***Explanation of results:***

We can see that most lines on the graph have an increased throughput when there are more threads introduced, until a point. There is an exception where a variable number of threads do not affect the throughput under any circumstance. These patterns and others are described in greater detail below.

***Question 1*** – Introduced a coarse-grained hash table where only one thread can access the table at any time. A more restrictive access to shared resources is applied. If we only allow one thread to access the data structure at a time, then we are prohibiting concurrency. We have no concurrency between threads in this case. Looking at the graph shows us that the average number of operations remains constant as the number of threads increase. An advantage of this approach is it simplicity. It is a trivial task to see that the object is thread-safe. This experiment was performed on my dual core laptop. Having multiple CPU’s means that tasks can literally run at the same time, providing parallelism to a program. We would like to see an increase in *‘Average Operations/Second’* when allowing 2 threads to use our data structure. This is not the case as coarse-grained solution implemented in this way disallows multiple thread access at once, creating a sequential bottleneck with high thread contention.

For this reason, the performance will remain constant. Regardless of the number of threads contending for the shared resource. Using the synchronized key word in java on a method means the invoking thread needs to acquire the objects intrinsic lock before making any progression. There is only one of these locks per object. Making entry to the object critical section mutually exclusive.

***Question 2*** – Again, this implementation has adopted a coarse-grained approach. The only difference is that instead of locking the whole data structure, we are locking individual hash chains that a thread has accessed. Without looking at the graph. We can guess what should happen here. We will have increased concurrency since more than one thread can safely access the data structure at once. It is not until the second thread is added that we see the noticeable improvement in comparison to Question 1 at the same stage. There is less contention as each individual thread can access their own bucket within the hash table without synchronization (Provided the other thread is accessing a different bucket which, according to statistics, is more likely than not). When the second thread is added, we can see that the performance approximately doubles. This can also be explained by the fact that threads are now executing in parallel; from the exploitation of our second CPU core.

Another interesting pattern that can been seen from the graph is that 3 out of the 6 different implementations start at approximately the same number of *‘Average Operations/Second’.* A single core is responsible for running this single thread. Initially, the effects of each algorithm cannot be noticeably observed. This is due to the lack of contention between threads. This low contention and lack of synchronization allows threads to run as fast as possible without waiting for other threads to release locks.

We continue to see an increase in the performance of Q2 until the introduction of a fourth thread, where performance levels off. This is due to more threads contending for the buckets so more synchronization is required, thus levelling off performance. Each bucket has one lock that needs to be acquired upon entry and released for other threads upon exit.

***Question 3*** – The third approach had the best performance in my experiment (in terms of in *‘Average Operations/Second’*). As the number of threads increased from 1 to 4 I observed a significant increase in the number of in *‘Average Operations/Second’.* Again, this is a by-product of another CPU core being introduced but it also tells us something else. The increase is more than the increase in Question 2. This can be explained through the usage of Javas ReadWrite lock, *‘java.util.concurrent.locks.ReentrantReadWriteLock’*.

The driver invokes the following functions on the hash table in this order:

* *add(x)*
* *contains(x) OR get(x) (variable number of these operations, 90% in this experiment)*
* *remove(x)*

Previously in question 2, a single lock was used per chain. This means that any thread accessing a chain, regardless of the function being invoked had to wait for another to leave.

In contrast, question 3 will allow many readers (threads invoking *contains(x)*) to access the data structure at once but only when there are no write operations being performed. This is where q3 gets the significant performance increase. There is less contention between reading invocations on the hash table. Therefore, we observe more throughput.

***Question 4*** – Hand-over-hand locking enables more concurrency to be introduced. Does this mean that the algorithm will be efficient? Not necessarily. On the graph Hand-over-hand locking can be seen to have a noticeably lower *‘Average Operations/Second’* when ran with one thread. This difference is since much more locks are acquired and released during executions. The overheads of constant synchronization like this is quite large and does effect the number of operations performed per second. A thread acquires the lock for each successive entry before releasing the lock for its predecessor. This version provides more concurrency than coarse-grained locking, but threads may acquire many successive locks, which is undesirable because lock acquisition typically involves expensive atomic operations (namely, compare-and-swap). Moreover, concurrent threads moving through the list may contend for locks even if they are searching for unrelated list entries. This is the worst performing approach in experiment A. An increased number of threads leads to more contention coupled with expensive synchronization.

***Question 5*** – In question five we needed to be mindful of keeping the concurrent objects functions linearizable so that other threads are guaranteed to view the correct state of the object. The first solution (Q5 – RCU – Hand-over-hand) uses the exact same technique for accessing individual chains in the hash table as used in question 4. It does not come as a surprise then that the performances of the two are nearly identical. The only difference between them is the extra acquisition of a lock, to allow mutual exclusion when a resize operation is invoked. Resizing isn’t a requirement of experiment A.

In addition, I have provided a second solution that is an extension of hand-over-hand locking. It shows an improvement in performance. With optimised locking, there is much less lock acquisitions and releases while having the benefit of no contention on traversals of the list/chain. A further improvement to get better performance would be to adopt the lazy synchronisation algorithm (Q5 Solution 3). This is where the ‘*contains(x)’* function is wait-free. It does not have to acquire locks and will perform better as the number of elements increases. More threads accessing the shared resource will mean that there is more concurrency. Performance again, does not have a noticeable improvement because of the abundance of synchronization events.

b) *Vary the fraction of lookup operations between 33% and 98% while keeping the number of threads fixed at a value that maximizes throughput according to your measurement in (a). Again measure for 1 second.*

***Explanation of results:***

***Question 1:*** With course-grained locking per hash table. We can see that an increase in read operations does not drastically affect the Average number of operations performed per second. This can be explained simply. Every thread calling a function on the hash table will require the tables lock. This means that reads do not get priority of access into the data structure in order to increase throughput. There is one critical section, allowing only one thread in at a time. Performance remains constant regardless of the percentage of read operations.

***Question 2:*** A coarse grain solution per has chain allows for more concurrency when compared with question 1. We can obtain a higher throughput because of this. As the percentage of *contains(x)* operations increases we can notice a small increase in throughput. Is this small increase significant? Well, we know that our algorithm treats *contains(x)* calls the same as any other function call (it still needs to acquire the lock of the chain it wants access to). We can then say that the varying levels of contains calls should not affect throughput on the course-grained/hash chain approach.

***Question 3:*** How does question 3 differ from the first two questions? A readers-writers lock allows multiple readers to access the data structure concurrently, i.e. they the can even access the same node in a bucket at the same time. This is a good quality for a concurrent hash table as most are exposed to a high volume of contain operations (e.g. memcached). As the number of contains operations increases we would expect to see an increase in throughput due to the nature of low contention for reading threads. Reading threads are only starved when a writer accesses the data structure. This can explain the gradual rise in the graph as percentage of read operations increases.

***Question 4 & Question 5 – Hand-over-hand:*** We see no increase in the throughput as the percentage of contains operations increases. Again, we know that multiple threads that need to read the same resource must acquire locks. This means that thread contention remains constant, regardless of what operation is being invoked.

***Question 5 Solution 2 (Optimistic synchronization) & Question 5 Solution 3 (Lazy synchronization):***

With optimistic synchronization, we can see that the percentage of contains operations does not affect the throughput of the execution. The main reason for this is that any contains operations is treated with same as an add or remove. They still need to acquire locks when they are accessing a node in a chain. Let’s consider the following execution scenario:

* 6 threads are accessing the hash table at the same time. All of which are going to perform a read operation. Suppose they all want to access the 2nd node in the first bucket.
* There is high thread contention as each thread is looking to read the same value in the shared resource.
* Optimistic synchronization employs the policy where each reading thread will still have to wait for the target nodes lock to be released before it can read the value at that position. Only one thread at a time can read the value.
* Therefore, there is no benefit for throughput if all invocations are ‘*contains(x)*’.

Using the same scenario lets explain why we see an increased throughput for Lazy synchronization when the percentage of contains operations increase.

* 6 threads are accessing the hash table at the same time. All of which are going to perform a read operation. Suppose they all want to access the 2nd node in the first bucket.
* As opposed the optimistic synchronisation, our ‘*contains(x)’* function is wait-free in Lazy synchronization. Any ongoing changes to the list cannot delay even a single thread from completing a *contains(x)* function. This means all 6 threads can concurrently read the value in the hash table without waiting each other to finish.
* For this reason, we notice an increase in throughput when the percentage of ‘contains(x)’ operations increases. Lower thread contention, therefore more concurrency and thus we observe higher throughput.