GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 5

Time Complexity Report

Muhammed Bedir ULUCAY

1901042697

Preface:

Normally each hash table when it reach the element it time complexity is constant time because our defined hash code is make a nearly uniqe index for key,value.

We are controlling Load Factor in hash map to reach constant time but to easy compare we can say that:

Our defined table element has three different table data structure these are ; linked list, tree set and reaching index to make more easy compare we assume the lenght of linked list or tree is number of element defined as **“m”** so that we can **compare more easy** each other.

n = hash table number of element

m = hash table element’s number of element

Hash Table Linked List Chain Class:

```java
/** Method get for class HashtableChain.
@param key The key being sought
@return The value associated with this key if found;
otherwise, null
*/
@Override
public V get(Object key) {

    int index = key.hashCode() % table.length;

    if (index < 0)
        index += table.length;
    if (table[index] == null)
        return null; // key is not in the table.

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        if (nextItem.getKey().equals(key))
            return nextItem.getValue();
    }
    // assert: key is not in the table.
    return null;
}
```

Reaching index of key is constant reaching an element in linked list is lineear time.

So that : **T(n,m) = Θ(m)**

```java
/** Method put for class HashtableChain.
@Override
public V put(K key, V value) {

    int index = key.hashCode() % table.length;

    if (index < 0)
        index += table.length;
    if (table[index] == null)
        table[index] = new LinkedList<>();

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        // If the search is successful, replace the old value.
        if (nextItem.getKey().equals(key)) {
            // Replace value for this key.
            V oldVal = nextItem.getValue();
            nextItem.setValue(value);
            return oldVal;
        }
    }
    // assert: key is not in the table, add new item.
    table[index].addLast(new Entry<>(key, value));
    numKeys++;

    if (numKeys > (LOAD_THRESHOLD * table.length))
        rehash();

    return null;
}
```

Reaching index of key is constant adding new element to linked list is lineear.

So that : **T(n,m) = Θ(m)**

```java
/**Rehash the hash table if the L is reach the defined limit*/
private void rehash() {

    @SuppressWarnings("unchecked")
    LinkedList<Entry<K, V>>[] newTable = new LinkedList[table.length * 2];

    for(LinkedList<HashtableLinkedListChain.Entry<K, V>> ll : table) {
        if(ll != null) {
            for(Entry<K, V> e : ll) {
                int index = e.hashCode() % newTable.length;

                if(index < 0)
                    index += table.length;

                if(newTable[index] == null )
                    newTable[index] = new LinkedList<>();

                newTable[index].addLast(e);
            }
        }
    }
    table = newTable;
}
```

We are replace each element in the hash map max number of element is : m * n

So that : **T(n,m) = Θ(n * m)**

```java
/**Removing taking key in the hash map
 * @param Object key
 * @return Value if we find the key in hash map
 * other wise return null*/
@Override
public V remove(Object key) {

    int index = key.hashCode() % table.length;

    if(index < 0)
        index += table.length;

    if(table[index] == null)    return null;

    for(Entry<K, V> e:table[index]) {
        if(e.getKey().equals(key)) {
            table[index].remove(e);
            numKeys--;
            return e.getValue();
        }
    }
    return null;
}
```

Reaching index of key is constant removing an element to linked list is lineear.

So that : **T(n,m) = Θ(m)**

Hash Table Tree Set Chain Class:

```java
@SuppressWarnings("unchecked")
@Override
public V get(Object key) {

    K nkey = (K) key;

    int index = nkey.hashCode() % table.length;     //arrange index
    if(index < 0)
        index += table.length;

    if(table[index] == null)    return null;

    //Comparing tree element one by one
    for(Entry<K, V> e : table[index]) {
        if(e.equals(new Entry<K,V>(nkey, null)))
            return e.getValue();
    }
    return null;
}
```

Reaching index of key is constant reaching an element in RBT is logaritmic.

So that : **T(n,m) = Θ(log m)**

```java
/** Method put for class HashtableTreeChain.
 * The map if already in map we change the value
 * if it is not in the map adding new element the map
 * according to the hash location*/
@Override
public V put(K key, V value) {

    K nkey = (K) key;
    int index = nkey.hashCode() % table.length;

    if(index < 0)
        index += table.length;

    if(table[index] == null)            //If the location is free assign it new Tree Set
        table[index] = new TreeSet<>();

    for(Entry<K, V> e: table[index]) {
        if(e.getKey().equals(nkey)) {       //Compare If already exist in the tree
            V oldVal = e.getValue();
            e.setValue(value);
            return oldVal;
        }
    }

    table[index].add(new Entry<>(key, value));      //Add as new entry
    numKeys++;

    if (numKeys > (LOAD_THRESHOLD * table.length))  //Checking the load Factor
        rehash();

    return value;
}
```

Reaching index of key is constant adding an element in RBT is logaritmic.

So that : **T(n,m) = Θ(log m)**

```java
/**Rehash the hash table if the L is reach the defined limit*/
private void rehash() {
    @SuppressWarnings("unchecked")
    TreeSet<Entry<K, V>>[] newTreeSet = new TreeSet[table.length * 2];

    for(TreeSet<Entry<K, V>> trs : table) {
        if(trs != null) {
            for(Entry<K, V> e : trs) {
                int index = e.getKey().hashCode() % newTreeSet.length;

                if(index < 0)
                    index += table.length;

                if(newTreeSet[index] == null)
                    newTreeSet[index] = new TreeSet<>();

                newTreeSet[index].add(e);
            }
        }

    }
    table = newTreeSet;
}
```

We are replace each element in the hash map max number of element is : m * n

So that : **T(n,m) = Θ(n * m)**

```java
/**Removing taking key in the hash map
 * @param Object key
 * @return Value if we find the key in hash map
 * other wise return null*/
@Override
public V remove(Object key) {

    int index = key.hashCode() % table.length;

    if(index < 0)
        index += table.length;

    if(table[index] == null)    return null;

    for(Entry<K, V> e :table[index]) {      //Compare the tree element to find removing element
        if(e.getKey().equals(key)) {
            table[index].remove(e);
            numKeys--;
            return e.getValue();        //If we find it returning removed key value
        }
    }

    return null;
}
```

Reaching index of key is constant removing an element in RBT is logaritmic.

So that : **T(n,m) = Θ(log m)**

Hash Table Coalesed Hash Map:

Coalesed Hash map is different than others a bit.To reach an element it is always use an saved index.So that it make faster reaching but this is cost as more memory space.

So that we can use O notation for calculating time complexity.

Mostly we will assume these methods as **amortised constant** time because we keep load factor in under control to reach constant time.

```java
/**Adding first for next2 value*/
private void addFirst(K key, V value,int index) {

    //Finding new free position
    int fromBottom = getNextEmptyFromBottom();
    //Setting next 2 of the normally add location
    table[index].setNext2(fromBottom);

    table[table[index].getNext2()] = new Node<K,V>(key,value);
}
```

Each operation is constant time.

So that : **T(n) = amortised O(1)**

```
/**Adding end of next2 */
private void addLast(K key, V value,int index) {

    int newloc = table[index].getNext2();

    for(int i=1,power = 1; ;++i,power = i*i) {

        //calculate the next index
        int loc = (newloc + power) % table.length;

        // if the key not found and reach the last we insert the end of hash queue
        if(table[loc] == null) {
            table[loc] = new Node<K,V>(key,value);
            findBeforeLocation(key,loc,power,i);
            return;
        }//If the key already in map we chance the value of key
        else if(table[loc].getKey().toString().equals(key.toString())) {
            table[loc].setValue(value);
            return;
        }
    }
}
```

We need to find the next place to put.But we use always as constant operation.In finding series.

So that : **T(n) = amortised O(1)**

```
/**Finding before index adding same hash value in the hash table*/
private void findBeforeLocation(K key,int loc,int power,int i) {

    for(int j = 1; ;j++) {

        //calculate the reverse of location
        int beforeLoc = loc - power + ((i-j) * (i-j)) ;

        while(beforeLoc < 0) {
            beforeLoc += table.length ;
            if(beforeLoc > 0) break;
        }

        //We are checking hash code and setting next and prev
        if(table[beforeLoc].getKey().hashCode() == new Node<K, V>(key,null).hashCode()) {
            table[loc].setPrev(beforeLoc);
            table[beforeLoc].setNext1((loc));
            break;
        }
    }
}
```

These is also track back to one before same hash code object.Even in two cycles constant time.

So that : **T(n) = amortised O(1)**

```java
/**Returning value of getting key in hash map
 * Wrapper method*/
@Override
@SuppressWarnings("unchecked")
public V get(Object key) {

    K nkey = (K) key;
    int index = nkey.hashCode();        //location of nkey in hash map

    if(table[index] == null)
        return null;

    //If location is taken by another key we are looking for next2 value
    if(table[index].getKey().hashCode() != nkey.hashCode() && table[index].getNext2() != null)
        index = table[index].getNext2();

    return get(nkey, index);
}
```

Operations are constant but we calling a recursive function in return.It's time complexity is define.

So that : **T(n) = amortised O(1)**

```java
/**Return value of key in hash map*/
private V get(K key, int index) {

    if(table[index].getKey().toString().equals(key.toString()))     //if we find the key return the value
        return table[index].getValue();

    if(table[index].getNext1() == null)             //if we dont find the key return null
        return null;

    index = table[index].getNext1();                //setting next

    return get(key,index);
}
```

As we sad in before function we are assume this method as amortised constant time because we keep load factor in under control to reach constant time.

So that : **T(n) = amortised O(1)**

```java
/**If we want to insert a new element but the location is taken
 * another value which has different hash code
 * so that we need to add different location and
 * we assign the index of inserted to next2 value*/
private void insertDifferentHashCode(K key,V value,int index) {

    if(table[index].getNext2() == null)
        addFirst(key, value, index);
    else
        addLast(key, value, index);
}
```

Add last and add first is amortised constant time.

T(n) = amortised O(1)

```
/**Inserting same hash in map and of the hash queue*/
private void insertSameHashCode(K key,V value,int index) {

    int loc = index;
    for(int i=1,power=1; ;++i,power = i*i) {

        //calculate the next index
        loc = (index + power) % table.length;
        if(loc < 0)
            loc += table.length;
        //If we reach the end of hash queue and
        //not find key in hash map insert the end of list
        if(table[loc] == null || table[loc].getKey() == null) {
            table[loc] = new Node<K,V>(key,value);
            findBeforeLocation(key,loc,power,i);
            return;
        }//If the key already in hash map change the value
        else if(table[loc].getKey().toString().equals(key.toString())) {
            table[loc].setValue(value);
            return;
        }
    }
}
```

Inserting element is calculte index and put index and put it valid place as same the others.

T(n) = amortised O(1)

```
/**Finding the key location in hash map
 * using next1 index*/
private V nextDeletion(K key,int index) {
    //If we find delete and shift map
    if(table[index].getKey().toString().equals(key.toString())) {
        V value = table[index].getValue();
        shiftTable(index);
        return value;
    }
    //if we dont find stop recursive
    if(table[index].getNext1() == null)
        return null;

    index = table[index].getNext1();
    return nextDeletion(key,index);
}
```

Finding key in the table and delete it and shifting array if it is need.These are also amortised constant time.

T(n) = amortised O(1)

```java
/**Adding new element to the hash map
 * According to the rules of hash code and open hash map*/
@Override
public V put(K key,V value) {
    //If the hash map is full return null
    if(numsKey == CAPACITY) {
        System.out.println("Table Full");
        return null;
    }

    //Getting index of key
    int index = key.hashCode() ;

    if(index < 0)
        index += table.length;

    //If simply location is free we add new node and return the value
    if(table[index] == null) {
        table[index] = new Node<K,V>(key,value);
        numsKey ++;
        return value;
    }

    //If the location taken by normally by key like k=13 is index=3  13.hash = 3
    if(table[index].getKey().hashCode() == new Node<K, V>(key,value).hashCode())
        insertSameHashCode(key,value, index);
    else    //If the location taken not same hash like 24 in 5   but 24.hash = 4 in that case we use next2
        insertDifferentHashCode(key,value, index);

    numsKey ++;
    return value;
}
```

Finding index of key and put it the location.Until find a valid space approching as quadratic.

T(n) = amortised O(1)

```java
/**Removing taking key in the hash map
 * @param Object key
 * @return Value if we find the key in hash map
 * other wise return null*/
@Override
@SuppressWarnings("unchecked")
public V remove(Object key){

    K number = (K) key;
    int index = number.hashCode();

    if(index < 0)
        index += table.length;

    if(table[index] == null)
        return null;

    if(table[index].getKey().hashCode() != number.hashCode()) {
        if(table[index].getNext2() == null)
            return null;
        index = table[index].getNext2();
    }

    V tmp = nextDeletion((K) key,index);

    if(tmp != null)
        numsKey--;

    return tmp;
}
```

Finding index is constant and track its way in nextDeletion function is amortised constant time

T(n) = amortised O(1)

```
/**Shifting table when we delete the according to index of
 * taking key in hash map*/
private void shiftTable(int index) {

    //Stop the recursive function
    if(table[index].getNext1() == null) {
        if(table[index].getPrev() != null )
            table[table[index].getPrev()].setNext1(null);
        table[index].clear();
        return;
    }

    //Arrange next index and current index
    int current = index;
    index = table[index].getNext1();

    //we look the next of next to check last element remove
    if(table[table[current].getNext1()] == null)
        table[current].setNext1(null);

    //swapping current and index of table
    table[current].swap(table[index]);

    //Setting next index when we removed
    if(table[index].getNext1() == null)
        table[current].setNext1(null);
    else
        table[current].setNext1(index);

    shiftTable(index);
}
```

Shift table is include so much constant time sconstant time operation like comparison or swapping.

In recursive call using next indexing until find the key.And we can do it all in amortised constant time.

T(n) = amortised O(1)