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

Muhammed Sinan Pehlivanoğlu
1901042664

BINARY SEARCH HASH TABLE

GET METHOD

```
@Override
public V get(K key) throws NullPointerException{
    int index = key.hashCode() % table.length;
    if(index <0) {
        index += table.length;
    }
    if(table[index] == null){
        return null;
    }/*
    for (Entry<K,V> entry: table[index]) {
        if(entry.getKey().equals(key)){
            return entry.getValue();
        }
    }*/
    Entry<K,V> entriy = table[index].find(new Entry<>(key,null));
    if(entriy != null){
        return entriy.getValue();
    }
    return null;
}
```

Find method of Binary Search Tree
Average Case is O(logn)

So
T(M) = O(logM).

IS EMPTY METHOD

```
@Override
public boolean isEmpty() { return numKeys == 0; }
```

T(n) = θ **(1).**

PUT METHOD

```
public V put(K key, V value) throws NullPointerException{
    int index = key.hashCode() % table.length;
    if(index < 0) {index += table.length;}

    if(table[index] == null) {table[index] = new BinarySearchTree<>(); };
    /*
    for (Entry<K,V> entry : table[index]) {
        if(entry.getKey().equals(key)){
            V oldValue = entry.getValue();
            entry.setValue(value);
            return oldValue;
        }
    }*/

    Entry<K,V> entry = table[index].find(new Entry<>(key,null));
    if(entry != null){
        V oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    }
    table[index].insert(new Entry<>(key,value));
    ++numKeys;
    if(numKeys > (Load_Threshold * table.length)){
        rehash();
    }
    return null;
}
```

Average case of find and insert Methods are O(logn).
So put methoid takes O(logM).

REMOVE

```
public V remove(K key) throws NullPointerException{
    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].delete(e);
            numKeys--;
            return e.getValue();        //If we find it returning removed key value
        }
    }*/
    Entry<K,V> entry = table[index].find(new Entry<K,V>(key,null));
    if(entry != null){
        table[index].delete(entry);
        numKeys--;
        return entry.getValue();
    }
    return null;
}
```

Find method of binary search takes O(logn) in average.
Delete method of binary search takes O(logn) in average.
So remove method takes O(logM) in average.

REHASH

```
 */
private void rehash(){

    BinarySearchTree<Entry<K,V>>[] newBst = new BinarySearchTree[table.length*2+1];
    int index;

    for(BinarySearchTree<Entry<K, V>> bst : table) {
        if(bst != null) {
            for(Entry<K, V> e : bst) {
                index  = e.getKey().hashCode() % newBst.length;
                if(index < 0) {
                    index += newBst.length;

                }
                if(newBst[index] == null) {
                    newBst[index] = new BinarySearchTree<Entry<K, V>>();
                }
                newBst[index].insert(e);
            }
        }

    }
    table = newBst;
}
```

Insert method of Binary Search Tree takes O(logN) in average.
Size of the Binary Search Tree is N.
So Rehash method takes O(NlogN).

SIZE METHOD

```
 */
@Override
public int size() { return numKeys; }
```

It takes θ(1).

HYBRİD HASH TABLE

GET METHOD

```
@Override
public V get(K key) throws NullPointerException {
    Entry<K,V> entry = find(key);
    if(entry != null) return  entry.getValue();
    return null;
}
```

Find Method takes amortised O(1).

IS EMPTY METHOD

```
@Override
public boolean isEmpty() { return numKeys ==0; }
```

Θ(1).

PUT METHOD

```
public V put(K key, V value) throws NullPointerException {
    int hash1 = hash1(key);
    int hash2 = hash2(key);
    int i =1;
    int index = hashFunction(hash1,hash2,i);

        while (table[index] != null){
            if(table[index].getKey().equals(key) && !table[index].deleteStatus){
                V oldVal = table[index].getValue();
                table[index].setValue(value);
                return oldVal;
            }
            index = hashFunction(hash1,hash2,i);
            ++i;
            if(index < 0 ) index = index+table.length;
        }
    table[index] = new Entry<>(key,value);
    table[index].hashCode = index;
    ++numKeys;
    Entry<K,V> iter = table[hashFunction(hash1,hash2, i: 1)];

    if( hashFunction(hash1,hash2, i: 1)!= index){
        if(iter != null){
            while (iter.next!=null){
                iter = iter.next;
            }
        }
        iter.next = table[index];
    }

double loadFactor = (double) (numKeys + numDeleted) / table.length;
if(loadFactor > Load_Threshold) rehash();
return null;
```

T(N) = amortised O(1).

REHASH METHOD

```
private void rehash(){

    HybridHashTable<K,V> newHashTable = new HybridHashTable<>();
    newHashTable.table = new Entry[table.length*2+1];

    for (Entry<K,V> entry: table) {
        if(entry != null){
            newHashTable.put(entry.getKey(),entry.getValue());
        }
    }
    this.clone(newHashTable);

}
```

T(N) = $\theta(N)$.

REMOVE METHOD

```
 */
@Override
public V remove(K key) throws  NullPointerException{

    Entry<K,V> entry = find(key);
    if(entry != null){

        V oldValue = entry.getValue();
        if(entry.next != null){

            entry.setValue(entry.next.getValue());
            entry.setKey(entry.next.getKey());
            entry.next = entry.next.next;
        }
        else entry.deleteStatus = true;
        ++numDeleted;
        return oldValue;
    }

    return null;
}
```

T(N) = amortised O(1).

SIZE METHOD

```
@Override
public int size() { return numKeys - numDeleted; }
```

T(N) = $\theta(1)$

Q2

MERGE SORT

```java
public static <T extends Comparable<? super T>> void sort(T[] array) throws ClassCastException{
        if(array.length > 1){
            int halfSize = array.length/2;
            T[] leftSub= (T[]) new Comparable[halfSize];
            T[] rightSub = (T[]) new Comparable[array.length - halfSize];
            System.arraycopy(array, srcPos: 0,leftSub, destPos: 0,halfSize);
            System.arraycopy(array,halfSize,rightSub, destPos: 0, length: array.length- halfSize);
            sort(leftSub);
            sort(rightSub);
            merge(array,leftSub,rightSub);
        }
}
```

```java
private static <T extends Comparable<? super T>> void merge(T[]array,T[] leftSub, T[] rightSub){
    int i = 0;
    int j = 0;
    int k = 0;
    while (i < leftSub.length && j < rightSub.length){
        if(leftSub[i].compareTo(rightSub[j]) <0){
            array[k++] = leftSub[i++];
        }

        else{
            array[k++] = rightSub[j++];
        }
    }

    while (i < leftSub.length ){
        array[k++] = leftSub[i++];
    }


    while (j < rightSub.length ){
        array[k++] = rightSub[j++];
    }

}
```

Best Case Time Complexity: **O(n*log n)**

Worst Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

QUICK SORT

```
private static <T extends Comparable<? super T>> void quickSort(T[] array, int first, int last){
    if(first <last){
        int pivot = partitioning(array,first, last);
        quickSort(array,first, last: pivot-1);
        quickSort(array, first: pivot+1,last);
    }

}
```

```
 */
private static <T extends Comparable<? super T>> int  partitioning(T[] array, int first, int last) {
    T pivot = array[first];
    int right = first+1;
    int left = first;
    do{
        if(array[right].compareTo(pivot) <=0){
            ++left;
            T temp = array[right];
            array[right] = array[left];
            array[left] = temp;
        }
        ++right;
    }while (right <= last);

    T temp  = array[first];
    array[first] = array[left];
    array[left] = temp;
    return left;
}
```

Best Case Time Complexity: **O(n*log n)**

Worst Case Time Complexity: **O(n*n)**

Average Time Complexity: **O(n*log n)**

NEW SORT

```
private static <T extends Comparable<? super T>> T[] sort(T[] array, int head, int tail){
    if(head >tail)
        return array;

    max_min = minMaxFinder(array, head, tail);
    T temp = array[head];
    array[head] = array[max_min[0]];
    array[max_min[0]] = temp;

    temp = array[tail];
    array[tail] =array[max_min[1]];
    array[max_min[1]] = temp;
    return  sort(array, head: head+1, tail: tail-1);
}

private static <T extends Comparable<? super T>> int[] minMaxFinder(T[] array, int head, int tail){

   if(head< tail){

           int mid = (head+ tail) /2 ;

           int[] maxMin_left = minMaxFinder(array,head,mid);
           int[] maxMin_right = minMaxFinder(array, head: mid+1,tail);
           return maxMin(array, maxMin_left, maxMin_right);
   }

   return new int[]{head,head};
}
```

Time Complexity is O(Nlogn).