Data Structures and Algorithm Analysis

Lab 12, Hash Table

Contents

- Implementing hash tables.

Hash table

In this lesson we will try to implement hash tables.

Writing a simple hash table is not so hard as writing a red-black tree.

Hash table

We start by using an array to store the elements.

Let’s define the HashTable class.

Define HashTable class

This is a simple definition of HashTable. We have a type node to denote the key-value pair.

```
public class HashTable <Key, Value> {

  class Node<Key, Value> {
      Key key;
      Value val;

      Node(Key k, Value v ) {
          key = k;
          val = v;
      }
  }

  private Node<Key, Value>[] allData;

  public HashTable( int capacity ) {
      allData = new Node[capacity];
  }
```

Define HashTable class

Note that dealing with generic arrays could be difficult, this is due to the JAVA language feature.

```
public class HashTable <Key, Value> {

  class Node<Key, Value> {
      Key key;
      Value val;

      Node(Key k, Value v ) {
          key = k;
          val = v;
      }
  }

  private Node<Key, Value>[] allData;

  public HashTable( int capacity ) {
      allData = new Node[capacity];
  }
```

Write put method in HashTable

First let's implement our "put" method as usual.

```
/**
  * @return whether the operation is successful
  */
public boolean put( Key key, Value value ) {
    int hashcode = key.hashCode();
    allData[hashcode] = new Node<Key, Value>(key, value);
    return true;
}
```

Write put method in HashTable

In order to test our put method, let's define a Particle class to insert. Note how we calculate the hashcode.

```
public class Particle {
  public double X;
  public double Y;

  public Particle( double x, double y ) {
        X = x;
        Y = y;
  }

  public int hashCode() {
        return (int)(X+Y);
  }

  public boolean equals( Object o ) {
        return (o instanceof Particle) && ((Particle)o).X == X
           && ((Particle)o).Y == Y;
  }
}
```

Write put method in HashTable

Now run the following code, we will see the follow error:

```
public static void main( String[] args ) {
  HashTable<Particle, String> table = new HashTable<>(10);
  table.put(new Particle(-1, -1), "Particle 1");
}
```

```
Exception in thread "main" java.lang.
  ArrayIndexOutOfBoundsException: Index -2 out of bounds for
   length 10
at HashTable.put(HashTable.java:26)
at HashTable.main(HashTable.java:32)
```

This is because hashCode does not guarantee to generate positive value. Even if it does, it does not always produce value within 10.

Write put method in HashTable

Now run the following code, we will see the follow error:

```
public static void main( String[] args ) {
  HashTable<Particle, String> table = new HashTable<>(10);
  table.put(new Particle(-1, -1), "Particle 1");
}
```

```
Exception in thread "main" java.lang.
  ArrayIndexOutOfBoundsException: Index -2 out of bounds for
   length 10
at HashTable.put(HashTable.java:26)
at HashTable.main(HashTable.java:32)
```

This is because hashCode does not guarantee to generate positive value. Even if it does, it does not always produce value within 10.

Write put method in HashTable

We need a hash function here. Let's start with converting a int value to positive here (we will soon see why we shouldn't do that).

```
private int hash( Key key ) {
    return (key.hashCode() & 0x7fffffff) % allData.length;
}

public boolean put( Key key, Value value ) {
    int hashcode = hash(key);
    allData[hashcode] = new Node<Key, Value>(key, value);
    return true;
}
```

Write put method in HashTable

But this put method has a drawback: it will put multiple elements in the same slot. This is not what we want for a hashtable. Let's implement separate chaining.

We change from:

```
private Node<Key, Value>[] allData;
```

to:

```
private LinkedList<Node<Key, Value>>[] allData;
```

Write put method in HashTable

We can also use ArrayList instead of an array. Let's use an ArrayList.

We change from:

```
private LinkedList<Node<Key, Value>>[] allData;
```

to:

```
private ArrayList<LinkedList<Node>> allData;
int capacity;

public HashTable( int capacity ) {
    this.capacity = capacity;
    allData = new ArrayList<>(capacity);
    for( int i = 0; i < capacity; ++ i )
        allData.add(new LinkedList<>());
}
```

Write put method in HashTable

How we can modify our put method to put a lot of objects inside our hashtable.

```java
public void put( Key key, Value value ) {
  int hashcode = hash(key);
  allData.get(hashcode).add(new Node(key, value));
}
```

Next we should write a small test.

Write put method in HashTable

Test the hash table with the following code:

```
private void printAll() {
  for( LinkedList<Node> list : allData )
      for( Node node : list )
          System.out.println(node.key+" "+node.val);
}

public static void main( String[] args ) {
  HashTable<Particle, String> table = new HashTable<>(10);
  table.put(new Particle(-1, -1), "Particle 1");
  table.put(new Particle(-1, -2), "Particle 2");
  table.printAll();
}
```

Write put method in HashTable

The test is successful. However, if we try again with capacity 10000 and insert randomly generated data from [0, 1] many times, we will see that all of the data fall in slot 0 and 1. This is bad for performance reasons.

The solution is to redesign our hash functions. Both in Particle and HashTable.

Write put method in HashTable

This time we try the hash function given by the algs4 library.

```java
private int hash(Key key) {
  int h = key.hashCode();
  h ^= (h >>> 20) ^ (h >>> 12) ^ (h >>> 7) ^ (h >>> 4);
  return h % capacity;
}
```

We see that the data distributes more evenly on the array.

Write get method in HashTable

Now we can easily write get and delete for the hash table:

```
public Value get( Key key ) {
  int hashcode = hash(key);
  for( Node node : allData.get(hashcode) )
      if( node.key.equals(key) )
          return node.val;
  return null;
}
```

We see that the data distributes more evenly on the array.

Write delete method in HashTable

Delete an element in the hash table:

```
public Value delete( Key key ) {
  int hashcode = hash(key);
  Iterator<Node> ite = allData.get(hashcode).listIterator();
  while( ite.hasNext() ) {
      Node n = ite.next();
      if( n.key.equals(key) ) {
          ite.remove();
          return n.val;
      }
  }
  return null;
}
```

We see that the data distributes more evenly on the array.