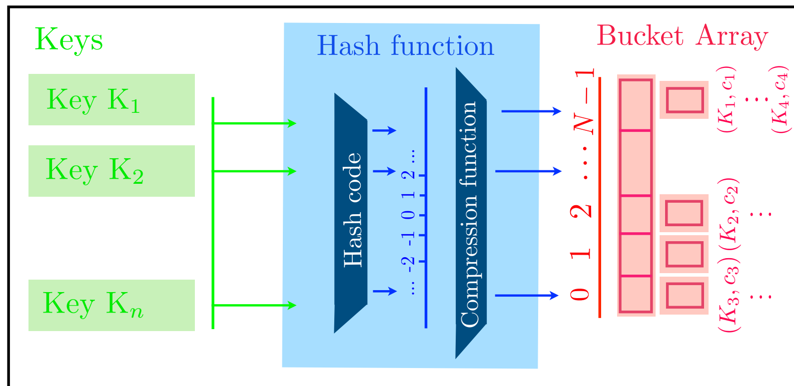
Data Structures

Augustin Cosse.

Spring 2021

April 29, 2021

Reminders: The hash table



Hash Table

Reminders

- In the hash table scheme described so far, it is important that the load factor $\lambda = n/N$ is kept below 1
- With separate chaining (considering two levels of containers or storing each bucket as a new map), when λ gets close to 1 the probability of a collision increases
- This adds to the overhead of our operations as we must now use additional maps to store the buckets that have collisions
- Similarly with open addressing (storing the bucket array as a simple array), as the load factor grows and start approaching 1, clusters of entries start to grow as well. These clusters cause the probing strategies to "bounce around" the bucket array for a considerable amount of time before they find an empty spot

Reminders

- All in all, and for the two approaches, experiments suggest that we should maintain $\lambda < 0.9$ for separate chaining and $\lambda < 0.5$ for open addressing
- If an insertion causes the load factor to go above the specified threshold, then it is common to resize the table adn to reinsert all objects into the new table
- Although we need not define a new hash code for each object, we do need to reapply a new compression function that takes into consideration the size of the new table
- When rehashing to a new table, it is a good requirement for the new array's size to be a prime number approximately doubling the previous size

An anecdote on Hashing and Computer Security

- In a 2003 paper, researchers discuss the possibilty to exploit a hash table worst case performance to cause a denial of service attack of internet technologies
- Since many published algorithms compute hash codes with a deterministic function, an attacker could precompute a very large number of moderate length strings that all hash to the identical 32 bits hash code.
- When two keys map to the same hash code, those keys will be inseparable in the collision resolution
- In 2011, another team of researchers demonstrated an implementation of just such an attack

An anecdote on Hashing and Computer Security

- Web servers allow a series of key-value parameters to be embedded in a URL using a syntax such as

  `?key1=val1&key2=val2&key3=val3`

- Those key value pairs are strings and a typical Web server immediately stores them in a hash-map.
- Servers already place a limit on the length and number of such parameters to avoid overload, but they presume that the total insertion time in the map will be linear in the number of entries
- However, if all the keys were to collide, the insertion into the map will require quadratic time, causing the server to perform an inordinate amount of work.

An anecdote on Hashing and Computer Security

- In 2012, the open JDK team announced the following resolution: they distributed a security patch that includes an alternative hash function that introduces randomization into the computation of hash codes maling it less tractable to reverse engineer a set of colliding strings.

Java implementation of the Hash table

- In this course, we will study two implementations of the Hash table: one using separate chaining, and the other using open addressing with linear probing
- While the two approaches are quite different, there are many higher level commonalities to the two hashing algorithms. For that reason, we will extend the `AsbtractMap` class to define a new `AbstractHashMap` class which provides much of the functionality common to our two has table implementations
- The `AbtractHashMap` class will not provide any concrete representation of the bucket array.

Java implementation of the Hash table

- In our first implementation, our `AbstractHashMap` class will assume each of the following to be abstract methods that will have to be implemented by the subclasses

createTable()	creates an initially empty table having size equal to the a designated `capacity` instance variable
bucketGet(h,k)	This method should mimic the semantics of the public `get` method but for a key k that is known to hash to bucket h
bucketPut(h, k, v)	This method should mimic the semantics of the public `put` method but for a key k that is known to hash to bucket h

Java implementation of the Hash table

- In our first implementation, our `AbstractHashMap` class will assume each of the following to be abstract methods that will have to be implemented by the subclasses

bucketRemove(h, k)
: This method should mimic the semantics of the public `remove` method, but for a key k known to hash to bucket h

entrySet()
: This standard map method iterates through all entries of the map. We do not delegate this on a per-bukcket basis because "buckets" in open addressing are not inherently disjoint

Java implementation of the Hash table

- We will design our `AsbtractHashMap` class so that it implements randomized Multiply-Add-and-Divide (MAD)

```
public abstract class AbstractHashMap<K,V>
                  extends AbstractMap<K,V> {
  protected int n = 0; // n of entries in dictionary
  protected int capacity; // length of table
  private int prime; // prime factor
  private long scale, shift;
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random( );
    scale = rand.nextInt(prime-1) + 1;
    shift = rand.nextInt(prime);
    createTable( );}
```

Java implementation of the Hash table

- We will also provide support for automatic resizing of the underlying hash table when the load factor reaches a certain threshold

```java
public abstract class AbstractHashMap<K,V>
                  extends AbstractMap<K,V> {
  protected int n = 0; // n of entries in dictionary
  protected int capacity; // length of table
  private int prime; // prime factor
  private long scale, shift;
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random( );
    scale = rand.nextInt(prime-1) + 1;
    shift = rand.nextInt(prime);
    createTable( );}
```

Java implementation of the Hash table

- To manage the load factor, the `AsbtractHashMap` class declares a protected member n which should equal the current numbe rof entries in the map. It should however rely on the subclasses to update this field from within the methods`bucketPut` and `bucketRemove`

```
public abstract class AbstractHashMap<K,V>
                    extends AbstractMap<K,V> {
  // part II
  public AbstractHashMap(int cap)
  { this(cap, 109345121); } // default prime
  public AbstractHashMap( ) { this(17); }

  protected abstract void createTable( );
  protected abstract V bucketGet(int h, K k);
  protected abstract V bucketPut(int h, K k, V v);
  protected abstract V bucketRemove(int h, K k);}
```

Java implementation of the Hash table

- If the load factor of the table increases beyond 0.5, we request a bigger table (using the `createTable()`) method and reinsert all entries into the new table

```
public abstract class AbstractHashMap<K,V>
                        extends AbstractMap<K,V> {
  // part IV
   private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
    for (Entry<K,V> e : entrySet( ))
      buffer.add(e);
    capacity = newCap;
    createTable( ); // based on updated capacity
    n = 0;
     // will be recomputed while reinserting entries
    for (Entry<K,V> e : buffer)
      put(e.getKey( ), e.getValue( ));}
```

Java implementation of the Hash table

- We define the general `get`, `remove` and `put` based on the abstract `bucketGet`, `bucketPut` and `bucketRemove` methods.

```java
public abstract class AbstractHashMap<K,V>
                         extends AbstractMap<K,V> {
  // part III
  public int size( ) { return n; }
  public V get(K key) { return bucketGet(hashValue(key), key)
  public V remove(K key)
         { return bucketRemove(hashValue(key), key); }
  public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key, value);
    if (n > capacity / 2) // load factor <= 0.5
    resize(2 *capacity - 1); // (or find nearby prime)
    return answer;
}
```

Java implementation of the Hash table

- Recall that the Multiply-Add-and-Divide method for compression maps an integer from the hash function to the position $[(a * i + b \mod p)] \mod N$ in the bucket array. N and p are primes.

```
public abstract class AbstractHashMap<K,V>
                    extends AbstractMap<K,V> {
  // part IV
private int hashValue(K key) {
  return (int) ((Math.abs(
  key.hashCode( )*scale + shift) % prime) % capacity);
  }
```

I. Separate chaining

- To represent each bucket for separate chaining, we use an instance of the simpler `UnsortedTableMap` class.

- This idea of using a simple solution to a problem to create a more advanced one is called bootstrapping.

```
public class ChainHashMap<K,V>
                extends AbstractHashMap<K,V> {
// fixed capacity array that serves as bucket
  private UnsortedTableMap<K,V>[ ] table;
  public ChainHashMap( ) { super( ); }
  public ChainHashMap(int cap) { super(cap); }
  public ChainHashMap(int cap, int p) { super(cap, p); }
/** Creates empty table having length = capacity. */
  protected void createTable( ) {
    table = (UnsortedTableMap<K,V>[ ])
          new UnsortedTableMap[capacity];}
```

I. Separate chaining

- The entire hash table is then represented as a fixed-capacity array A of the secondary maps. Each cell, $A[h]$ is initially a null reference; we only create a secondary map when an entry is first hashed to a particular bucket

```
public class ChainHashMap<K,V>
                extends AbstractHashMap<K,V> {
// fixed capacity array that serves as bucket
  private UnsortedTableMap<K,V>[ ] table;
  public ChainHashMap( ) { super( ); }
  public ChainHashMap(int cap) { super(cap); }
  public ChainHashMap(int cap, int p) { super(cap, p); }
/** Creates empty table having length = capacity. */
  protected void createTable( ) {
    table = (UnsortedTableMap<K,V>[ ])
          new UnsortedTableMap[capacity];}
```

I. Separate chaining

- We implement the methods `bucketGet(h,k)`, `bucketPut(h,k,v)` and `bucketRemove(h,k)` by respectively calling `A[h].get(k)`, `A[h].put(k)` and `A[h].remove(k)`

```
protected V bucketGet(int h, K k) {
  UnsortedTableMap<K,V> bucket = table[h];
  if (bucket == null) return null;
  return bucket.get(k);
}
```

I. Separate chaining

- We implement the methods `bucketGet(h,k)`, `bucketPut(h,k,v)` and `bucketRemove(h,k)` by respectively calling `A[h].get(k)`, `A[h].put(k)` and `A[h].remove(k)`

```java
/** hashValue = h */
protected V bucketPut(int h, K k, V v) {
  UnsortedTableMap<K,V> bucket = table[h];
  if (bucket == null)
    bucket = table[h] = new UnsortedTableMap<>( );
  int oldSize = bucket.size( );
  V answer = bucket.put(k,v);
  n += (bucket.size( ) - oldSize);
  return answer;
}
```

I. Separate chaining

- We implement the methods `bucketGet(h,k)`, `bucketPut(h,k,v)` and `bucketRemove(h,k)` by respectively calling `A[h].get(k)`, `A[h].put(k)` and `A[h].remove(k)`

```
/** hashValue = h */
protected V bucketRemove(int h, K k) {
  UnsortedTableMap<K,V> bucket = table[h];
  if (bucket == null) return null;
  int oldSize = bucket.size( );
  V answer = bucket.remove(k);
  n -= (oldSize - bucket.size( ));
  return answer;
}
```

I. Separate chaining

- There is however need for care for several reasons:
  - First because we choose to leave table cells as `null` until a secondary map is needed. Each of the operations must thus begin by checking to see if $A[h]$ is null. In the case of `bucketGet` and `bucketRemove`, we can simply we can simply return `null` if the entry does not exist. In the `bucketPut`, a new entry must be inserted so we instantiate a new `UnsortedTableMap`
  - The second point to which we must pay attention is that in our implementation of the `AbstractHashMap` class, it is the child class which is responsible for maintaining the variable n when an entry is newly inserted or deleted. The methods `bucketPut` and `BucketRemove` therefore have to take this into account when making calls to the methods `put` and `remove`.

I. Separate chaining

- We conclude with an implementation of the `entrySet()` method which returns an iterable instance

```
public Iterable<Entry<K,V>> entrySet( ) {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
  for (int h=0; h < capacity; h++)
    if (table[h] != null)
      for (Entry<K,V> entry : table[h].entrySet( ))
        buffer.add(entry);
  return buffer;}
```

II. Linear Probing

- Just as we considered an implementation of a bucket table using separate chaining, we now consider an implementation of our bucket table relying on linear probing.
- To handle deletion, we place a special marker at the entry at which any entry has been deleted so that we can distinguish between this and empty entries
- This idea is implemented using the DEFUNCT entry as the sentinel

II. Linear Probing

```
public class ProbeHashMap<K,V>
                extends AbstractHashMap<K,V> {

                // Part I
  private MapEntry<K,V>[ ] table;
  private MapEntry<K,V> DEFUNCT =
                 new MapEntry<>(null, null);
  public ProbeHashMap( ) { super( ); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates empty table */
  protected void createTable( ) {
    table = (MapEntry<K,V>[ ]) new MapEntry[capacity]; }
```

II. Linear Probing

- On top of the `bucketGet`, `bucketPut` and `bucketRemove` methods, when considering linear probing, we will also need to add a method to either find a given key in the array or return an empty slot

- For that purpose, we implement the method `findSlot` which screens the whole bucket (the bottom of the bucket corresponding to the first null entry after h) to find an entry matching the specified key k or to return the first available/empty slot (DEFUNCT or null)

- This idea is implemented by keeping track of a variable '`avail`' which is updated only once (when the first empty slot is found)

II. Linear Probing

- If the key is not found, we return the first empty slot as a negative number to avoid mistaking it with the position of the key (which is returned as positive integer)
- When looking for the key, we must continue probing until we find the key, or until we reach the bottom of the bucket (i.e. the first null reference). In particular, we cannot stop the search upon reaching a DEFUNCT sentinel.

II. Linear Probing

```
    // Part II (see explanations on previous slide)
private boolean isAvailable(int j) {
   return (table[j] == null || table[j] == DEFUNCT);}
private int findSlot(int h, K k) {
   int avail = -1; // no slot available (thus far)
   int j = h; // index while scanning table
   do {
     if (isAvailable(j)) { // empty or defunct
       if (avail == -1) avail = j; //available slot!
       if (table[j] == null) break;
     } else if (table[j].getKey( ).equals(k))
       return j; // successful match
     j = (j+1) % capacity; // look cyclically
   } while (j != h);
   return -(avail + 1); // search has failed
   }
```

II. Linear Probing

```
      // Part III
  protected V bucketGet(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null; // no match found
    return table[j].getValue( );}
/** Associates (k,v) with bucket h. */
  protected V bucketPut(int h, K k, V v) {
    int j = findSlot(h, k);
    if (j >= 0) // this key has an existing entry
    // if a match is found replace the value
      return table[j].setValue(v);
 // if no match is found we add the key value pair
    table[-(j+1)] = new MapEntry<>(k, v);
    n++;
    return null;}
```

II. Linear Probing

```
    // Part IV
  protected V bucketRemove(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null; // nothing to remove
    V answer = table[j].getValue( );
    table[j] = DEFUNCT; // mark slot as deactivated
    n--;
    return answer;}
/**Returns collection of key-value entries of the map.*/
  public Iterable<Entry<K,V>> entrySet( ) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
    for (int h=0; h < capacity; h++)
      if (!isAvailable(h)) buffer.add(table[h]);
    return buffer;}}
```