UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 13

Dictionary Data Structure (Part 3)

Yoan Pinzón

Table of Content Session 13

Dictionary Data Structure

Hash Table Representation

- It uses a **hash function** f to map keys into positions in a table called the **hash table**

- Let $\mathcal{K}$ be the domain of all keys and let $\mathcal{N}$ be the set of natural numbers. Then,
  $f : \mathcal{K} \rightarrow \mathcal{N}$. The element with key k is stored in position $f(k)$ of the hash table

Search for an element with key k:

- Compute $f(k)$ and see if there is an element at position $f(k)$ of the table

Delete an element with key k:

- Search for the element. If found, then make the position $f(k)$ of the table empty

Insert an element with key k:

- Search for the element. If not found, i.e., position $f(k)$ of the table is empty, then place the element at that position

- If the key range is small, then we can easily implement hashing

  E.g., let keys for a student record dictionary be 6 digit ID numbers, and that we have 1000 students with IDs ranging from 771000 to 772000. Then, $f(k) = k - 771000$ maps IDs to positions 0 through 1000 of the hash table

- The above situation is called *ideal hashing*

- However, what we do when the key range is large?

  E.g., keys are 12-character strings; each key has to be converted into a numeric value by say mapping a blank to 0, an 'A' to 1, ..., and a 'Z' to 26. This conversion maps the keys into integers in the range $[1, 27^{12} - 1]$

Hashing with Linear Open Addressing

- The size of the hash is smaller than the key range

- Find a hash function which maps several keys into the same position of the hash table

- A commonly used such function is

  $$f(k) = k\%D$$

  where D is the size of the hash table

- Each position of the hash table is called a *bucket*

- What happens if $f(k_1) = f(k_2)$, for $k_1 \neq k_2$, i.e., a so-called *collision* has occurred

- If the relevant bucket has space to store an additional element, then we are done. Otherwise, we have an *overflow* problem

- How do we overcome overflows?

- Search the table (sequentially) to find the next available bucket to store the new element

Search for element with key k

- Search starting from the *home bucket* $f(k)$ and by examining successive buckets (and considering the table as circular) until one of the following happens:

  1. A bucket containing the element with the key k is found

  2. An empty bucket is reached

  3. We returned to the home bucket

- In case 1, we have found the element we are looking for. In the other two cases, the table doesn't contain the requested element

Deleting the element with key k

- Deletion needs special care: if we simply make table position empty, then we may invalidate the correctness of the Search method
- This implies that deletion may require to move several elements in order to leave the table in a state appropriate for the Search method
- *Alternative solution:* introduce a field `neverUsed` in each bucket. Initially this is set to true

  When an element is placed into a bucket, its `neverUsed` field becomes false

  Case 2 of Search is replaced by: "a bucket with `neverUsed` field equal to true is reached"
- Deletion is accomplished by simply vacating the relevant bucket
- The alternative solution requires the re-organization of the hash table when the number of buckets with false `neverUsed` is large

Class Definition of HashTable

```
package unal.datastructures;

import java.util.*;

public class HashTable<K extends Comparable<? super K>, E>
   implements Dictionary<K, E>
{
   // fields
   protected int divisor;           // hash function divisor
   protected DataDict<K, E>[] table; // hash table array
   protected boolean[] neverUsed;   // parallel array
   protected int size;              // number of elements in table

   // constructor
   public HashTable( int theDivisor ) { /* ... */ }
```

```java
    // methods
    public boolean isEmpty( ) { /* ... */ }
    public int size( ) { /* ... */ }
    private int search( K theKey ) { /* ... */ }
    public E get( K theKey ) { /* ... */ }
    public E put( K theKey, E theElement ) { /* ... */ }
    public E remove( K theKey ) { /* ... */ }
    public String toString( ) { /* ... */ }
    public static void main( String[] args ) { /* ... */ }
}
```


DataDict class

```java
class DataDict<K extends Comparable<? super K>, E>
{
    // fields
    K key;      // its key
    E element; // element in node

    // constructor
    DataDict( )
    {
        key = null;
        element = null;
    }

    DataDict( K theKey, E theElement )
    {
        key = theKey;
        element = theElement;
    }
```

```java
    @Override
    public String toString( )
    {
        return "[" + Objects.toString( element ) +
            ", key=" + Objects.toString( key ) + "]";
    }
}
```


constructor

```java
    @SuppressWarnings( "unchecked" )
    public HashTable( int theDivisor )
    {
        divisor = theDivisor;
        table = new DataDict[ divisor ];
        neverUsed = new boolean[ divisor ];
        Arrays.fill( neverUsed, true );
        size = 0;
    }
```

isEmpty

```java
/** @return true iff the table is empty */
public boolean isEmpty( )
{
    return size == 0;
}
```


size

```java
/** @return current number of elements in the table */
public int size( )
{
    return size;
}
```

search

```java
/** search an open addressed hash table for an element with
 * key theKey
 * @return location of matching element if found, otherwise
 * return location where an element with key theKey may be
 * inserted provided the hash table is not full */
private int search( K theKey )
{
   int i = Math.abs( theKey.hashCode( ) ) % divisor;
   int j = i; // start at home bucket
   do
   {
      if( neverUsed[ j ] || ( table[ j ] != null && table[ j ].key.equals( theKey ) ) )
         return j;
      j = ( j + 1 ) % divisor; // next bucket
   } while( j != i ); // returned to home bucket?

   return j; // table full
}
```


get

```java
/** @return element with specified key
 * @return null if no matching element */
public E get( K theKey )
{
   // search the table
   int b = search( theKey );

   // see if a match was found at table[ b ]
   if( neverUsed[ b ] || !table[ b ].key.equals( theKey ) )
      return null; // no match

   return table[ b ].element; // matching element
}
```

put

```java
/** insert an element with the specified key
 * overwrite old element if there is already an
 * element with the given key
 * @throws IllegalArgumentException when the table is full
 * @return old element ( if any ) with key theKey */
public E put( K theKey, E theElement )
{
   // search the table for a matching element
   int b = search( theKey );

   // check if matching element found
   if( neverUsed[ b ] )
   {
      // no matching element and table not full
      table[ b ] = new DataDict<K, E>( theKey, theElement );
      neverUsed[ b ] = false;
      size++;
      return null;
   }
```



```java
   else
   {  // check if duplicate or table full
      if( table[ b ].key.equals( theKey ) )
      {  // duplicate, change table[ b ].element
         E elementToReturn = table[ b ].element;
         table[ b ].element = theElement;
         return elementToReturn;
      }
      else throw new IllegalArgumentException( "table is full" );
   }
}
```

remove

```java
/** remove from the hash table
 * @return removed element */
public E remove( K theKey )
{
   // search the table for a matching element
   int b = search( theKey );

   if( neverUsed[ b ] )
      return null; // no matching element and table not full
   if( table[ b ].key.equals( theKey ) )
   {
      E elementToReturn = table[ b ].element;
      table[ b ] = null;
      size--;
      return elementToReturn;
   }
   else
      return null;
}
```


toString

```java
/** convert to a string */
@Override
public String toString( )
{
   StringBuilder s = new StringBuilder( "\n[" );

   // put elements into the buffer
   for( int i = 0; i < divisor; i++)
      s.append( "{" + Objects.toString( table[ i ] ) +
         "," + ( neverUsed[ i ] ? "T" : "F" ) + "}, " );

   if( size > 0 )
       s.setLength( s.length( ) - 2 ); // remove last ", "

   s.append( "]\n" );

   // create equivalent String
   return new String( s );
}
```

main

```java
/** test method */
public static void main( String[] args )
{
   HashTable<Integer, Integer> h = new HashTable<>( 11 );

   h.put( 80, 180 ); h.put( 40, 140 ); h.put( 65, 165 );
   h.put( 58, 158 ); h.put( 24, 124 ); h.put( 2, 102 );
   h.put( 13, 113 ); h.put( 46, 146 ); h.put( 16, 116);
   h.put( 7, 107 ); h.put( 21, 121);
   System.out.println( h );

   try
   {
      h.put( 99, 99 );
   }
   catch( Exception e )
   {
      System.out.println( "No␣memory␣for␣99" );
   }
```



```java

   // update element
   h.put( 7, 29 );
   System.out.println( h );
   System.out.println( "Element␣" + h.get( 2 ) + "␣found" );
   System.out.println( "Element␣" + h.remove( 58 ) + "␣removed" );
   System.out.println( h );
   System.out.println( "Element␣" + h.get( 2 ) + "␣found" );
}
```

Compiling `HashTable.java`

```
C:\2016699\code> javac unal\datastructures\HashTable.java ↵
C:\2016699\code> java unal.datastructures.HashTable ↵
[{[107, key=7],F}, {[121, key=21],F}, {[124, key=24],F}, {[180, key=80],F}, {[15
8, key=58],F}, {[102, key=2],F}, {[113, key=13],F}, {[140, key=40],F}, {[146, ke
y=46],F}, {[116, key=16],F}, {[165, key=65],F}]

No memory for 99

[{[29, key=7],F}, {[121, key=21],F}, {[124, key=24],F}, {[180, key=80],F}, {[158
, key=58],F}, {[102, key=2],F}, {[113, key=13],F}, {[140, key=40],F}, {[146, key
=46],F}, {[116, key=16],F}, {[165, key=65],F}]

Element 102 found
Element 158 removed

[{[29, key=7],F}, {[121, key=21],F}, {[124, key=24],F}, {[180, key=80],F}, {null
,F}, {[102, key=2],F}, {[113, key=13],F}, {[140, key=40],F}, {[146, key=46],F},
{[116, key=16],F}, {[165, key=65],F}]

Element 102 found
```