Data Structures

Augustin Cosse.

Spring 2021

April 27, 2021

Maps

- A map is an abstract data type designed to efficiently store and retrieve values based on a uniquely identifying search key
- Specifically, a map stores key value pairs (k, v) which we call entries where k is a key and v is the corresponding value
- Keys are required to be unique so that the mapping is defined by the association of keys to values.
- Map are also known as associative arrays because the entry's key serves somewhat like an index into the map
- Unlike in arrays, however, keys do not need to be numeric and do not need to directly designate a position within the structure

Maps

- Common applications of maps include:
    - A university's information system relies on some form of student ID as a key that can be mapped to that student's associated record
    - The domain name system (DNS) maps a host name (e.g. `www.wiley.com`) to an internet protocole (IP) address (e.g. `208.215.179.146`)
    - A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information

Maps

- Since a map stores a collection of object, it should be viewed as a collection of key-value pairs
- As an ADT, a map M supports the following methods (Part I)

size()	returns the number of entries in M
isEmpty()	Returns a boolean indicating whether M is empty
get(k)	Returns the value v associated to to key k
put(k, v)	If M does not have an entry with key equal to k then add entry (k, v) to M and return null else replace with v the existing value of the entry with key k and return the old value
remove(k)	removes from M the entry with key equal to k
keySet()	Returns an iterable collection containing all keys stored in M

Maps

- As an ADT, a map M supports the following methods (Part II)

values()	Returns an iterable collection containing all the values of entries stored in M (with repetition if multiple keys map to the same value)
entrySet()	Returns an iterable collection containing all the key-value entries in M

- Notice that each of the operations get(k), put(k, v) and remove(k) returns the existing value associated to key k if the map has such an entry and returns null otherwise. This can introduce ambiguity in maps that allow null as a natural entry

Maps

- To resolve that ambiguity, some of the implementation of the `java.util.Map` interface explicitly forbid use of a null value (and null keys).
- When a null entry is allowed, it is also often possible to resolve the ambiguity by relying on the method containsKey(k) that is defined in the interface.

Maps

Method	***Return Value***	***Map***
isEmpty()	true	$\{\}$
put$(5,A)$	null	$\{(5,A)\}$
put$(7,B)$	null	$\{(5,A),(7,B)\}$
put$(2,C)$	null	$\{(5,A),(7,B),(2,C)\}$
put$(8,D)$	null	$\{(5,A),(7,B),(2,C),(8,D)\}$
put$(2,E)$	C	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(7)	B	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(4)	null	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(2)	E	$\{(5,A),(7,B),(2,E),(8,D)\}$
size()	4	$\{(5,A),(7,B),(2,E),(8,D)\}$
remove(5)	A	$\{(7,B),(2,E),(8,D)\}$
remove(2)	E	$\{(7,B),(8,D)\}$
get(2)	null	$\{(7,B),(8,D)\}$
remove(2)	null	$\{(7,B),(8,D)\}$
isEmpty()	false	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

The java Map interface

- The formal definition of the java Map interface can be found below. Note that it relies on generics to store the key-value pair $< K, V >$.

```java
public interface Map<K, V>{
  int size();
  boolean isEmpty();
  V get(K key);
  V put(K key, V value);
  V remove(K key);
  Iterable<K> keySet();
  Iterable<V> values();
  Iterable<Entry<K, V>> entrySet();
}
```

Application: counting word frequencies

- As an application of maps, we consider the problem of counting the number of occurences of words in a document
- This is a standard task when performing a statistical analysis of a document, for example when categorizing an email or news article

Application: counting word frequencies

- We begin with an empty map and start mapping the words to their frequencies. We scan through the the input considering adjacent characters to be words

```
public class WordCount { // Part I
  public static void main(String[ ] args) {
    Map<String,Integer> freq = new ChainHashMap<>( );
    // scan for words, using nonletters as delimiters
    Scanner doc =
     new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
    while (doc.hasNext( )) {
    // convert next word to lowercase
    String word = doc.next( ).toLowerCase( );
    Integer count = freq.get(word);
    if (count == null)
      count = 0;
    freq.put(word, 1 + count); }
```

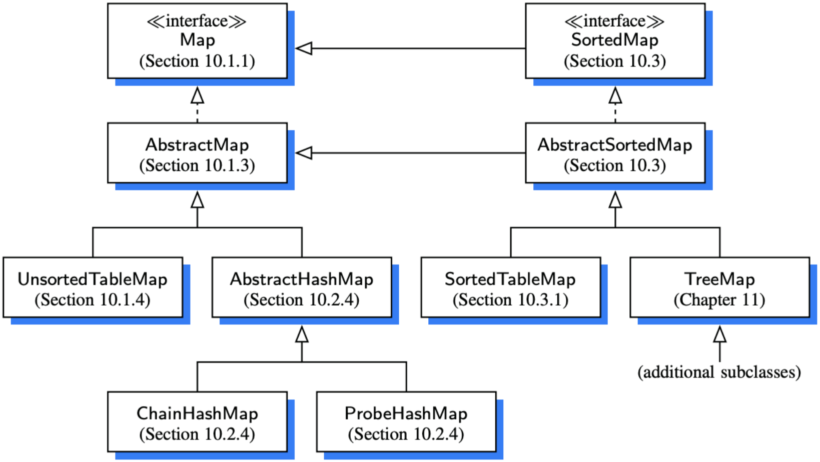
Application: counting word frequencies

- After processing the entire input, we loop through the `entrySet()` of the map to determine which word has the most occurences

```
public class WordCount { // Part II
  public static void main(String[ ] args) {
    // ..
    int maxCount = 0;
    String maxWord = "no word";
    for (Entry<String,Integer> ent : freq.entrySet( ))
      if (ent.getValue( ) > maxCount) {
        maxWord = ent.getKey( );
        maxCount = ent.getValue( );
      }
}}
```

Abstract Map base class

- We will consider multiple possible implementations of the map ADT using a variety of data structures. As before, we will rely on a combination of abstract and concrete classes



Abstract Map base class

- We begin by designing an `AbstractMap` base class from which we will derive all the other map implementations

```
public abstract class AbstractMap<K,V> implements Map<K,V> {
  public boolean isEmpty( ) { return size( ) == 0; }
  //Part I: ------- nested class -----------
  protected static class MapEntry<K,V> implements Entry<K,V> {
    private K k; // key
    private V v; // value
    public MapEntry(K key, V value) {
      k = key;
      v = value;}
    // public methods of the Entry interface
    public K getKey( ) { return k; }
    public V getValue( ) { return v; }
    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected V setValue(V value) {
    V old = v;
    v = value;
    return old;}}
```

Abstract Map base class

- We create an iterable instance as the output returned by the method

```
public abstract class AbstractMap<K,V> implements Map<K,V> {

  //Part II: Support for public keySet method
  private class KeyIterator implements Iterator<K> {
    private Iterator<Entry<K,V>> entries = entrySet( ).iterator( );
    public boolean hasNext( ) { return entries.hasNext( ); }
    public K next( ) { return entries.next( ).getKey( ); }
    public void remove( ) {
    throw new UnsupportedOperationException( ); }
   }
  private class KeyIterable implements Iterable<K> {
    public Iterator<K> iterator( ) { return new KeyIterator( ); }
   }
public Iterable<K> keySet( ) { return new KeyIterable( ); }
```

Abstract Map base class

- We create an iterable instance as the output returned by the method

```
public abstract class AbstractMap<K,V> implements Map<K,V> {

  //Part III: Support for value iterator
  private class ValueIterator implements Iterator<V> {
    private Iterator<Entry<K,V>> entries = entrySet( ).iterator( );
    public boolean hasNext( ) { return entries.hasNext( ); }
    public V next( ) { return entries.next( ).getValue( ); }
    public void remove( )
     { throw new UnsupportedOperationException( ); }
  }
  private class ValueIterable implements Iterable<V> {
    public Iterator<V> iterator( ) { return new ValueIterator( ); }
  }
  public Iterable<V> values( ) { return new ValueIterable( ); }
}
```

A first simple Unsorted Map

- We will start with a simple concrete implementation of the map ADT that relies on storing key-value pairs in arbitrary order within a Java ArrayList
- Within this first implementation, each of the `get(k)`, `put(k, v)` and `remove(k)` methods will require an initial scan of the array to determine whether an entry with key equal to k exists.

A first simple Unsorted Map

- We will therefore implement a public utility `findIndex(key)` that returns the index at which such an entry is found or -1 if no entry is found

```
public class UnsortedTableMap<K,V>
                  extends AbstractMap<K,V> {
  private ArrayList<MapEntry<K,V>> table =
                   new ArrayList<>( );
  public UnsortedTableMap( ) { }

  private int findIndex(K key) {
    int n = table.size( );
    for (int j=0; j < n; j++)
      if (table.get(j).getKey( ).equals(key))
        return j;
    return -1; // key was not found
}
```

A first simple Unsorted Map

- To remove an entry from the ArrayList, since the list is unsorted, instead of using the `remove` method that would result in shifting all the elements to shift the empty entry, we prefer to replace the removed entry with the last entry in the list.

```java
public V remove(K key) {
  int j = findIndex(key);
  int n = size( );
  if (j == -1) return null; // not found
  V answer = table.get(j).getValue( );
  if (j != n-1)
     // relocate last entry to empty space
    table.set(j, table.get(n-1));
  table.remove(n-1); // remove last entry of table
  return answer;
}
```

A first simple Unsorted Map

```
public int size( ) { return table.size( ); }
  public V get(K key) {
    int j = findIndex(key);
    if (j == -1) return null; // not found
    return table.get(j).getValue( );
}
/** Associates value with key*/
public V put(K key, V value) {
  int j = findIndex(key);
  if (j == -1) {
    table.add(new MapEntry<>(key, value));
    return null;
} else // key already exists
return table.get(j).setValue(value); }
```

A first simple Unsorted Map

- We finally provide support for the iterator over <key,value> pairs (Part I)

```
private class EntryIterator
                 implements Iterator<Entry<K,V>> {
  private int j=0;
  public boolean hasNext( ) { return j < table.size( ); }
  public Entry<K,V> next( ) {
    if (j == table.size( ))
            throw new NoSuchElementException( );
    return table.get(j++);
  }
  public void remove( )
  { throw new UnsupportedOperationException( ); }
}
```

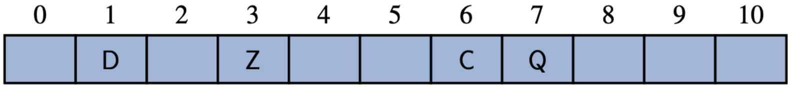
A first simple Unsorted Map

- We finally provide support for the iterator over <key,value> pairs (Part II)

```java
private class EntryIterable
                        implements Iterable<Entry<K,V>>{
  public Iterator<Entry<K,V>> iterator( )
                  { return new EntryIterator( ); }
}
/** Returns an iterable collection of key-value pairs */
public Iterable<Entry<K,V>> entrySet( )
                  { return new EntryIterable( ); }
}
```

A first simple Unsorted Map

- One of the most efficient data structures for storing a map one that is widely used in practice is the Hash table
- In order to understand the notion of hash table, let us consider a map with integer keys and represent it through the lookup table below



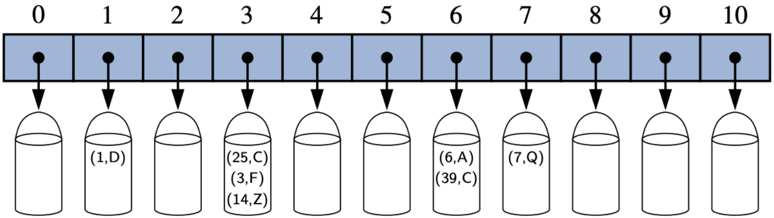
- In this representation, we store the value associated with key k at index k of the table. The basic map operations `get`, `put` and `remove` can then be implemented in $O(1)$ worst case time

A first simple Unsorted Map

- There are two challenges in extending this framework to the more general setting of a map.
  - First we will want the array to have a size equal to the size of the map
  - Second, keys in general maps are not required to be integers.
- The novel concept in a Hash table is the use of a Hash function to map general keys to corresponding indices in a table.
- Ideally keys should be mapped (by the hash function) to the whole interval $0, \dots, N-1$. However it might happen that one or more keys get mapped to the same integer

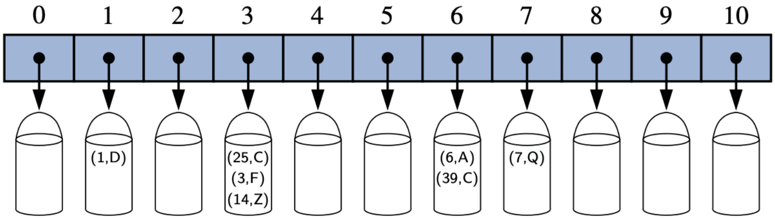
A first simple Unsorted Map

- We will conceptualize our table as a Bucket array where each bucket might manage a collection of entries that are sent to the corresponding index by the Hash function h.
- Given this idea, we can then use $h(k)$ as an index to navigate in our Bucket array. That is we store the map entry (k, v) in the bucket $BA[h[k]]$



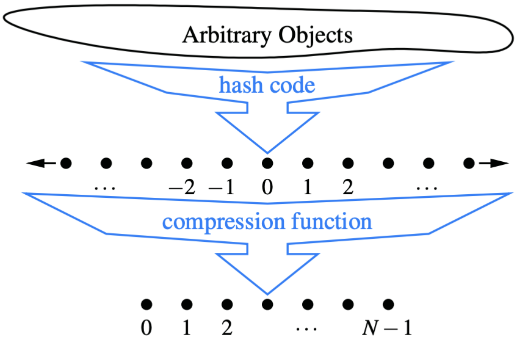
A first simple Unsorted Map

- if two or more keys share the same hash value, we say that a collision has occured (we will discuss how to handle collisions later)
- We say that a hash function is good if it maps the keys in our map so as to sufficiently minimize collisions
- In practice, we will also want the hash function to be fast and easy to compute



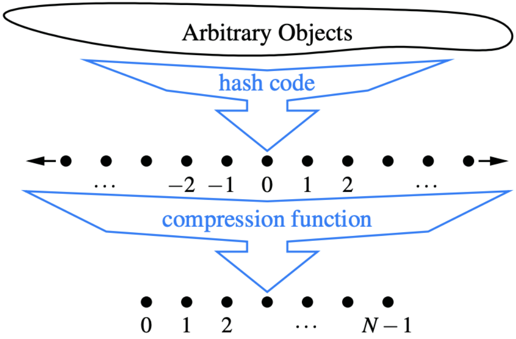
A first simple Unsorted Map

- It is common to view a hash function $h(k)$ as consisting of two parts:
  - A hash code that maps a key k to an integer
  - And a compression function that maps the output to the hash code to the range of indices [0,N-1] of the bucket array.


Arbitrary Objects
hash code
$\cdots$ -2 -1 0 1 2 $\cdots$
compression function
0 1 2 $\cdots$ $N-1$

A first simple Unsorted Map

- The advantage of this decomposition is that the Hash code is independent of the size of the hash table size
- This allows the development of a general hash code that can be for a hash table of any size



Hash codes (I)

- The first action that a hash function performs is to take an arbitrary key k in the map and compute an integer that is called the hash code for k
- We would like hash codes to avoid collisions as much as possible.
- Note that for any data type X that is represented using at most as many bits as our hashcode, we can simply take as our hash code, the integer associated to the bits
- In particular, java maintains 32 bits hash code and for any key of type **char**, **int**, **short** or **byte**, we can get a first hash code by casting the variables to int.

Hash codes (II)

- In a similar vein, for **float**, we can convert x to integer using `Float.floatToIntBits(x)` and use the resulting value as our hash code.
- For a type representation whose bit representation is larger than the hash code (such as **long** and **double** for example), such a scheme is not directly applicable. One solution can then be to use the high order (i.e. most significant or leftmost) 32 bits or conversely the low-order (rightmost) 32 bits
- Although simple in practice, such an approach might introduce a lot of collisions

Hash codes (III)

- A better approach would be to combine the low order and high order 32 bits.
- One approach to implement this idea could be to sum up the two 32 bits, ignoring overflow (whatever cannot be stored), or take the exclusive OR of the two components.
- That idea can be extended to any object that can be viewed as an n-tuple of 32 bits integers, such as $(x_1, x_2, \ldots, x_n)$. We can then obtain a 32 bits representation by combining the entries of the tuple as for example $\sum_{i=0}^{n-1} x_i$ or $x_0 \oplus x_1 \oplus \ldots x_{n-1}$

Polynomial Hash codes

- The summation and exclusive-or hash code are not good choices for character strings or other variables length objects that can be viewed of tuples $(x_1, x_2, \ldots, x_n)$ where the ordering of the x_i matters
- As an illustration of this, consider a 16 bits hash code for a character string s that sums the unicode values of the characters in s
- Such a hash code would produce a lot of unwanted collisions for common strings such as `stop`, `tops`, `pots`, ..
- A better hash code should somewhow take into consideration the positions of the x_i's

Polynomial Hash codes

- An alternative hash code that takes into account the ordering can be obtained by considering the polynomial

$$x_0a^{n-1} + x_1a^{n-2} + \ldots + x_{n-2}a + x_{n-1}$$

- Mathematically, this is simply a polynomial in a that takes as the components $(x_0, x_1, \ldots, x_{n-1})$ of x as its coefficients.
- Intuitively a polynomial hash code uses multiplication by different powers as a way to spread out the influence of each component across the code.

Polynomial Hash codes

- Of course on a typical computer, evaluating the polynomial will be done using a finite representation and the value will periodically overflow.
- However, since we are interested in spreading of the object x with respect to the other keys, we can ignore such overflows
- We can also choose a so that it has non zero low order bits that will preserve some of the information content (including in the case of overflow)
- Experimental studies suggest that taking $a = 33, 37, 39$ and 41 is usually a good choice (produced few collisions) when used with character strings that are English words

Cyclic ShiftHash codes

- A variant of the polynomial Hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits

- For example, a 5 bits cycle shift of the 32 bits value

$$\underline{00111}101100101101010100010101000$$

  is achieved by taking the leftmost right bits and placing them on the rightmost side of the representation

$$101100101101010100010101000\underline{00111}$$

- While this operation has little meaning in terms of arithmetic, it accomplishes the goal of varying the bits of the calculation

Cyclic ShiftHash codes

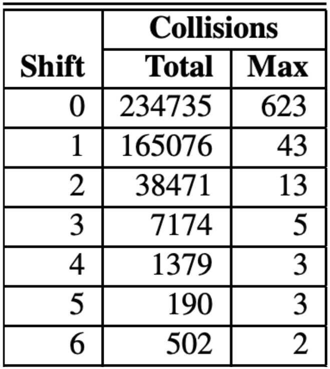
- An implementation of a cyclic-shift hash code computation for a character string in java can be obtained as below

```
static int hashCode(String s){
  int h=0;
  for(int i=0; i<s.length(); i++){
    h = (h<<5) | (h>>>27); // length is 32 bits
    // logical shift >>> adds 0's on the left
    // so we can use the logical or to concatenate
    h+= (int) s.charAt(i);
  }
}
```

Cyclic ShiftHash codes

- As with the traditional polynomial hash code, fine tuning is required when using the a cyclic shift and it is advised to carefully choose the amount of shift for each new character

- The table below is generated from a list of over 230 English words



	Collisions	
Shift	**Total**	**Max**
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2

Hash codes in Java

- The notion of Hash codes is an integral part of the java language. As an example, the Object class which serves as an ancestor of all object types includes a default `hashCode()` method that returns a 32 bits integer of type int
- We must be careful when relying on the default version of `HashCode`. For Hashing shemes to be reliable, it is imperative that two objects that are viewed as equal to each other have the same hash code.
- This is important because if an entry is inserted into a map and a later search is performed on a key that is considered equivalent to that entry's key, the map must recognize this as a match

Hash codes in Java

- If a class defines an equivalence through the `equals` method, then that class should also provide a consistent implementation of the `hashCode` method, such that if `x.equals(y)`, then `x.hashCode() == y.hashCode()`.
- As an example, Java's String class defines the `equals` method so that two instances are equivalent if they have precisely the same sequence of characters. The class also overrides the `hashCode` method to provide consistent behavior

Hash codes in Java

- As an example of how to implement a hashCode for a user defined class, we implement such a hashCode for the SinglyLinked List class below
- A robust hasCode can be computed on LinkedLists by taking the exclusive or of its elements hashCode and then applying a 5 bits cyclic shift

```
public int hashCode(){
  int h=0;
  for (Node walk = head; walk!=null; walk = walk.getNext)
    h^= walk.getElement().hashCode(); // bitwise or
    h = (h<<5) | (h>>>27);
}
```

Compression functions

- The hash code for a key k will typically not be suitable for immediate use with a bucket array because the value returned by the hash code may be negative or may simply exceed the capacity of the array
- Once we have determined the hash code, there is still the issue of mapping this integer in the range [0,N-1]
- This computation known as compression function, is the second action performed as part of the overall hash function
- Just as for the hash function, a good compression function is one that minimizes the number of collisions

Compression functions

- A simple compression function is the division method, which maps an integer i to $i \mod N$ where N is the size of the bucket array
- Additionally, if we take N to be a prime number, this will often reduce the number of collision (the result will tend to srepad out more)
- As an example if we consider keys with associated hash codes given by $\{200, 205, 210, 215, 220, ..., 600\}$ that we want to insert into a bucket array of size 100, each hash code will appear at least 3 times. If we use a bucket of size 101 instead, there will be no collisions
- Choosing N to be a prime number is often not enough to avoid collisions though.

The MAD method

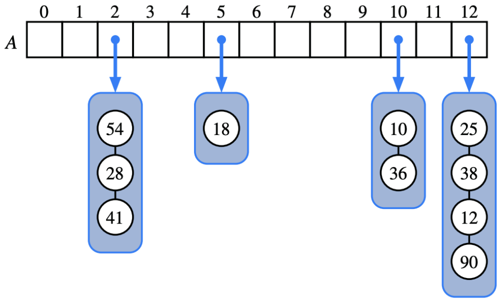
- A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys is the Multiply-Add-and-Divide (or MAD) method
- The method maps an integer i to $[(ai + b) \mod p] \mod N$ where N is the size of the bucket array, p is a prime number larger than N and a and b are integers chosen at random from the interval $[0, p-1]$ with $a > 0$.
- This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and and get us closer to having a "good" hash function (that is one such that the probability that any two keys collide is $1/N$).
- This good behavior would be the same as if the keys were thrown in the bucket uniformly at random

Collision handling schemes

- As we saw, the main idea of a hash table is to take a bucket array BA and a hash function h and use them to implement a map by storing each entry (k, v) in the bucket $BA[h(k)]$
- The idea is challenged, however, when we have multiple keys such that $h(k_1) = h(k_2)$.
- The existence of such collisions prevents us from being able to simply insert new entries directly in the bucket. Moreover, it also complicates the procedure for performing insertion, search and deletion operations.

Separate chaining

- A simple and efficient way for dealing with collisions is to have each bucket $BA[j]$ store its own secondary container, holding all entries (k, v) such that $h(k) = j$
- A natural choice for the container is a small map instance implemented using an unordered list, as shown below. This collision resolution rule is known as separate chaining and is illustrated below



Separate chaining

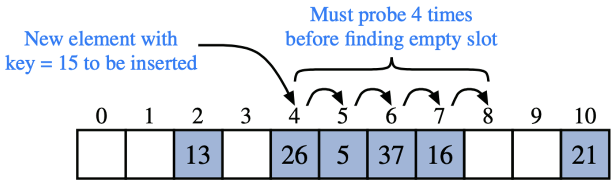
- In the worst case, operations on an individual bucket take time proportional to the size of the bucket
- Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N
- If given a good hash function, the core map operations should run in $O(\lfloor n/N \rfloor)$. The ratio $\lambda = n/N$ is called the load factor of the hash table
- if the load factor is bounded by a constant, we thus known that the core operations on the hash table will run in $O(1)$ expected time.

Open Addressing

- The separate chaining rule has many nice properties such as simple implementations of map operations. But it nevertheless has one disadvantage : it requires the use of an auxilliary data structure to hold entries with colliding keys
- An alternative known as open addressing. This alternative however requires a bit more complexity to properly handle collisions
- Open addressing requires that the load factor is always at most 1 and that entries are stored in the cells of the bucket array itself

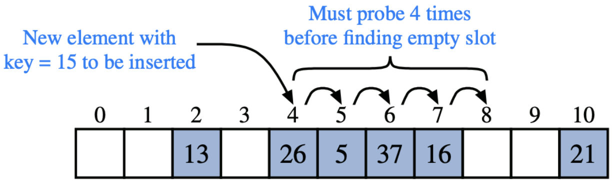
Open Addressing

- A simple method for collision handling with open addressing is linear probing
- With this approach, if we try to insert an entry (k, v) into a bucket $A[j]$ that is already occupied, where $j = h(k)$ then we next try $A[(j+1) \mod N]$. If $A[(j+1) \mod N]$ is also occupied, then we try $A[(j+2) \mod N]$ and so on, until we find an empty spot
- Of course, this collision resolution strategy requires that we change the implementation of the `get`, `put` or `remove` operations


New element with key = 15 to be inserted
Must probe 4 times before finding empty slot
0
1
2
3
4
5
6
7
8
9
10
13
26
5
37
16
21

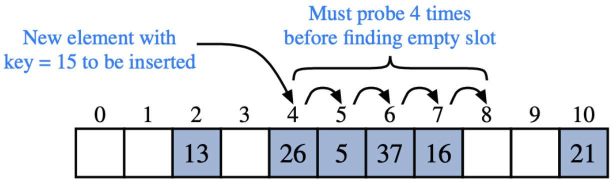
Open Addressing

- Of course, this collision resolution strategy requires that we change the implementation of the `get`, `put` or `remove` operations
- In particular, to attempt to locate an entry with key equal to k, we must examine consecutive slots, starting from $A[h(k)]$, until we either find an entry with an equal key, or an empty bucket.


New element with key = 15 to be inserted
Must probe 4 times before finding empty slot
0 1 2 3 4 5 6 7 8 9 10
13 26 5 37 16 21

Open Addressing

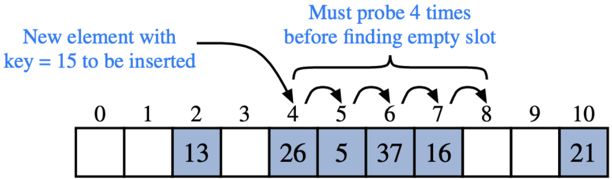
- To implement a deletion, we cannot simply remove a found entry from its slot in the array
- For example, after the insertion of key 15 in the array below, if the entry with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4 then index 5 then index 6 at which an empty cell is found



Open Addressing

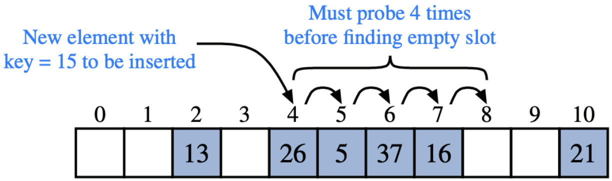
- A typical way to get around this difficulty is to replace a deleted entry with a “defunct” sentinel object. With this sepcial marker possibly occupying spaces in our hash table, we modify our search algorithm so that the search for a key k will skip over the cells containing the defunct sentinel and continue probing until reaching the desired entry or an empty bucket (ore returning back to where we started from)

- Additionaly, our algorithm for put should remember a defunct location encountered during the search for k, since this is a valid place to put a new entry (k, v), if not existing exntry is found beyond it.


New element with key = 15 to be inserted
Must probe 4 times before finding empty slot
0
1
2
3
4
5
6
7
8
9
10
13
26
5
37
16
21

Open Addressing

- Although use of open addressing can save space, linear probing suffers from an additional disadvantage: it tends to cluster the entries of a map into contiguous runs, which may even overlap
- Contiguous runs of occupied hash cells cause searches to slow down considerably



Open Addressing

- Another open addressing strategy known as quadratic probing iteratively tries the buckets $A[h(k) + f(i) \mod N]$ for $i = 0, 1, 2, \ldots$ where $f(i) = i^2$ until finding an empty bucket.

- As with linear probing, the quadratic probing strategy complicates the removal operation but it does avoid the kind of clustering patterns that occur with linear probing.

- Nevertheless, it creates it own kind of clustering known as secondary clustering in which the set of filled array cells still has a non uniform pattern, even if we assume that the original hash codes are distributed uniformly

- When N is a prime and the array is less than half full, the quadratic probing strategy is guaranteed to find an empty slot. However this guarantee does not hold anymore when either of this assumption falls.

Open Addressing

- A third open addressing strategy that does not cause the clustering of the kind produced by linear probing or the kind produced by quadratic probing is the double hashing strategy.
- In this strategy, we choose a secondary hash function h', and if h maps some key k to a bucket $A[h(k)]$ that is already occupied, then we iteratively try the buckets $A[h(k) + f(i) \mod N]$ for $i = 1, 2, 3, \ldots$ where $f(i) = i \cdot h'(k)$.
- The secondary hash function is not allowed to evaluate to 0. A common choice for this function is $h'(k) = q - (k \mod q)$ for some prime $q < N$. N should also be a prime
- Finally, a last approach to avoid clustering can be to try buckets $A[(h(k) + f(i)) \mod N]$ with $f(i)$ based on a pseudo random number generator