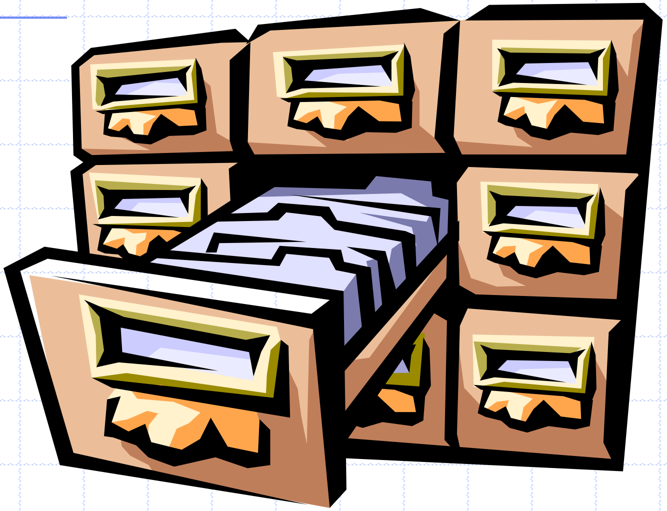
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Maps



Maps and Hashtables

- Define the Map ADT
- Implement an unsorted map
- Explain Hash functions
- Implement a hash table
- Use sorted maps
- Explain Set ADT

Priority Queues - Review

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v) - inserts an entry with key k and value v
  - removeMin() - removes and returns the entry with smallest key
- Additional methods
  - min() - returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
- Sequence priority queues are implemented as unsorted or sorted lists
- The performance for insert and removeMin and min is O(1) and O(n) for unsorted and O(n) and O(1) for sorted

Priority Queues - Review

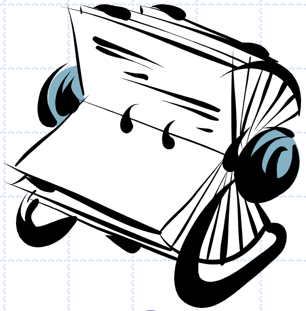
- A **heap is a binary tree storing keys at its nodes** and satisfying Heap-order, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$, and Complete binary tree properties.
  - A heap storing $\boldsymbol{n}$ keys has height $\boldsymbol{O}(\log \boldsymbol{n})$
  - Can be used to implement priority queues
- Algorithm **upheap** restores the heap-order property by **swapping *k* along an upward path from the insertion node -** runs in $\boldsymbol{O}(\log \boldsymbol{n})$ time
- Algorithm **downheap** restores the heap-order property by **swapping key *k* along a downward path from the root -** runs in $\boldsymbol{O}(\log \boldsymbol{n})$ time
- We can construct a heap storing $\boldsymbol{n}$ given keys in using a **bottom-up construction** with log $\boldsymbol{n}$ phases - runs in $\boldsymbol{O}(\boldsymbol{n})$ time.

Priority Queues - Review

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$

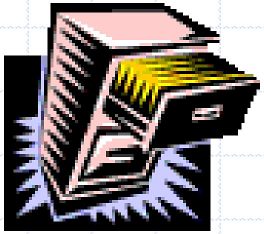
- We can **use a priority queue to sort a list of comparable elements**
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time - The algorithm is called **heap-sort**

Maps



- A map models a searchable collection of **key-value** entries
- The main operations of a map are for **searching**, **inserting**, and **deleting** items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database (based on student id)

The Map ADT



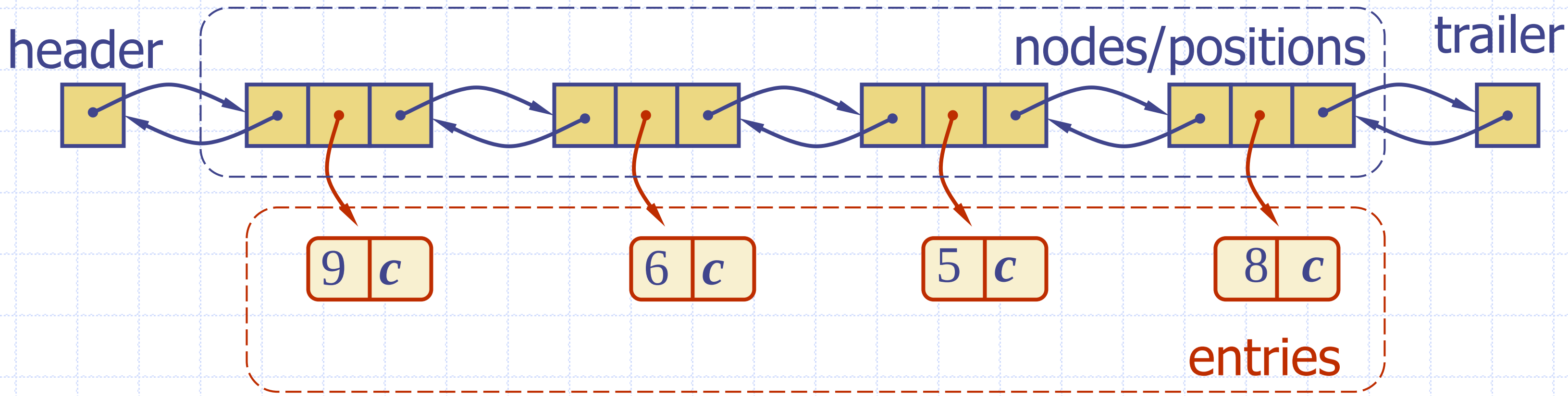
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the **entries** in M
- keySet(): return an iterable collection of the **keys** in M
- values(): return an iterator of the **values** in M

Example

Operation	*Output*	*Map*
isEmpty()	**true**	Ø
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*)
isEmpty()	**false**	(7,*B*),(8,*D*)

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map **in a list S** (based on a doublylinked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()   { the next position in B }
        if p.element().getKey() = k        then
            return p.element().getValue()
    return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.set(p,(k,v))
        return t        {return the old value}
S.addLast((k,v))
n = n + 1      {increment variable storing number of entries}
return null    { there was no entry with key equal to k }
```

The remove(k) Algorithm

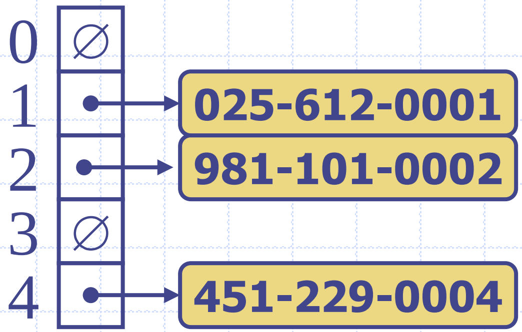
```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n – 1      {decrement number of entries}
        return t       {return the removed value}
return null            {there is no entry with key equal to k}
```

Performance of a List-Based Map

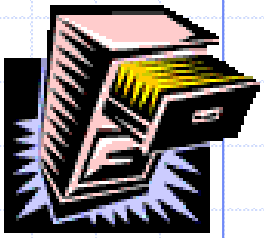
- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)
- **Hash tables** are the most efficient data structure for implementing a Map

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Hash Tables

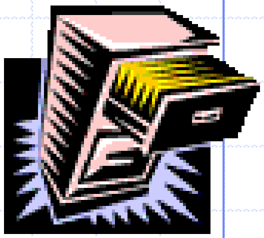


Recall the Map ADT



- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Intuitive Notion of a Map

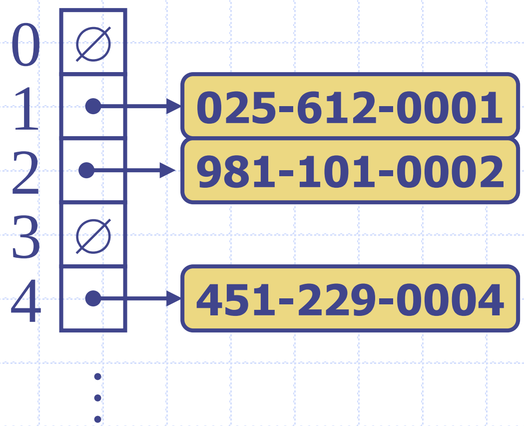


- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to $N - 1$, for some $N \geq n$.

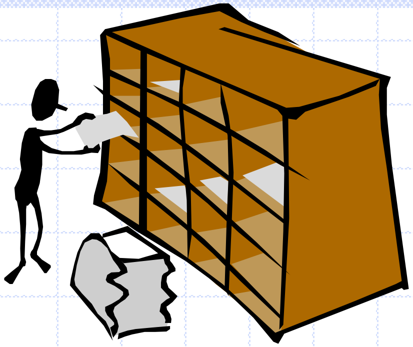
0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



Hash Functions and Hash Tables



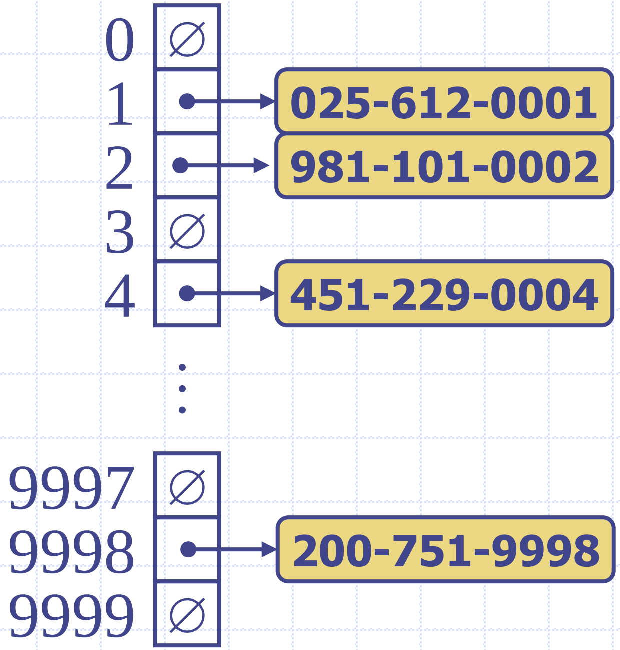
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $$h(x) = x \bmod N$$

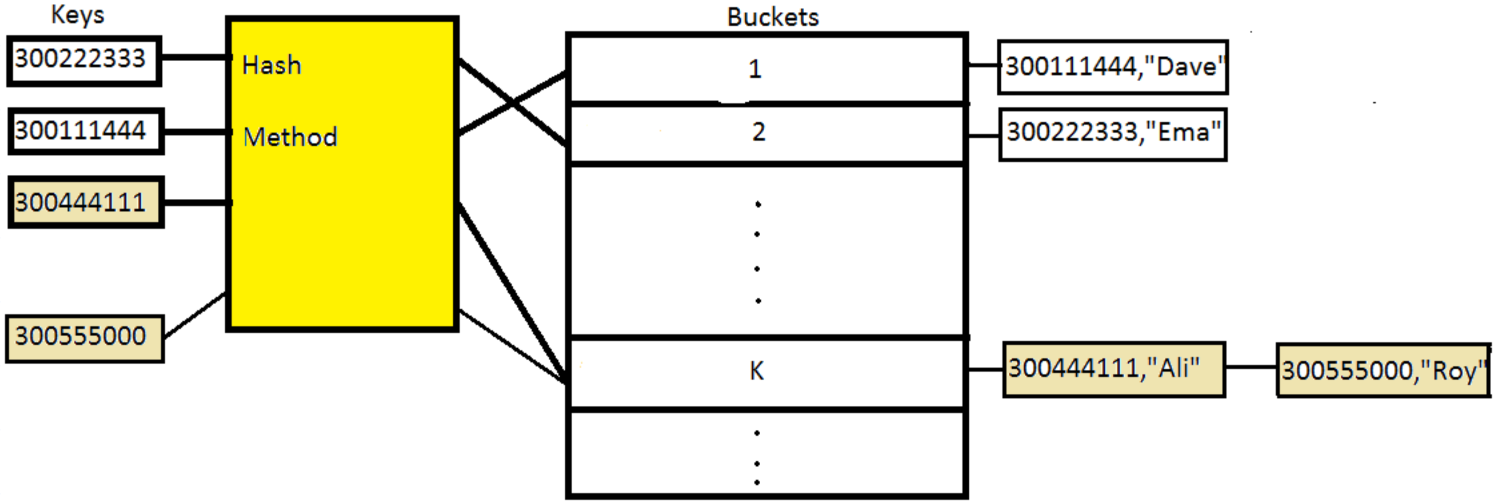
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $\boldsymbol{h}(\boldsymbol{x}) = \text{last four digits of } \boldsymbol{x}$



Hash Functions and Hash Tables



- For example, if the hash code equals one, the Hashtable places the value into Bucket 1, and so on.
- Multiple keys may map to the same bucket, something known as a *collision.*
- A linked list is used to maintain key/values in the same bucket

Hash Functions

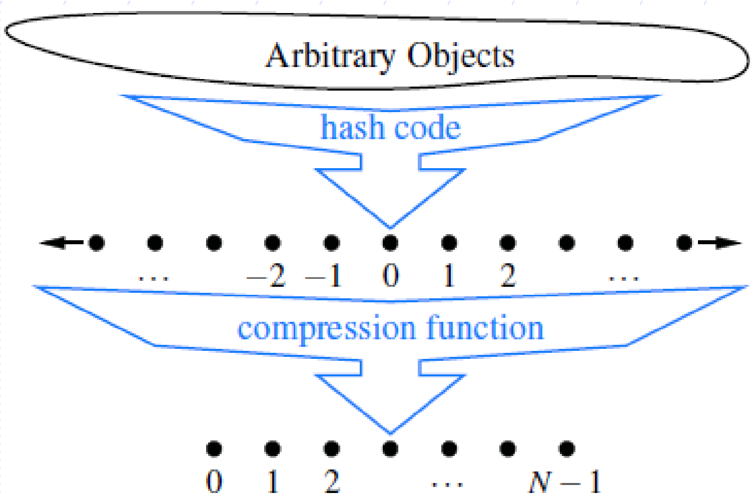
- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

  h_2: integers $\rightarrow [0, N - 1]$

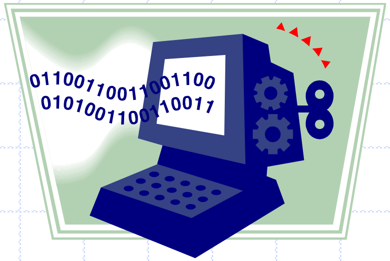


- The **hash code** is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

- The goal of the **hash function** is to "disperse" the keys in an apparently random way
- hash code portion of the computation is independent of a specific hash table size

Hash Codes implementation



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0\, a_1\, \ldots\, a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots \\ \ldots + a_{n-1} z^{n-1}$$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

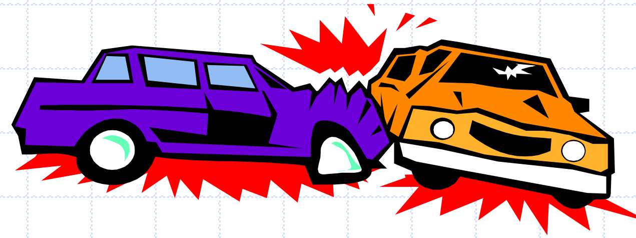
Abstract Hash Map in Java

```
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;                    // number of entries in the dictionary
  protected int capacity;                 // length of the table
  private int prime;                      // prime factor
  private long scale, shift;              // the shift and scaling factors
  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();
  }
  public AbstractHashMap(int cap) { this(cap, 109345121); }    // default prime
  public AbstractHashMap() { this(17); }                       // default capacity
  // public methods
  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)                 // keep load factor <= 0.5
      resize(2 * capacity - 1);           // (or find a nearby prime)
    return answer;
  }
```

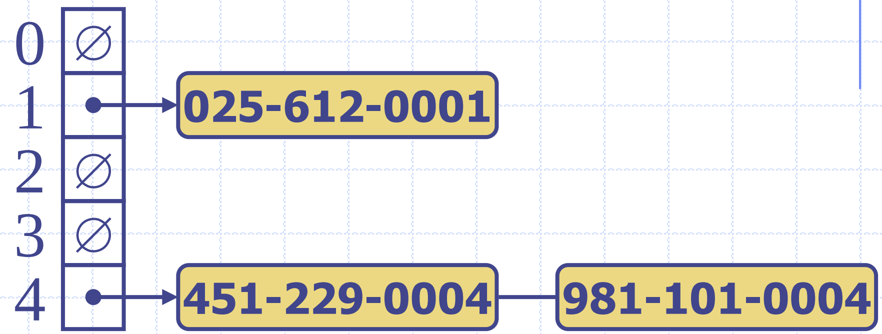
Abstract Hash Map in Java, 2

```
// private utilities
private int hashValue(K key) {
  return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
}
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());
}
// protected abstract methods to be implemented by subclasses
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);
}
```

Collision Handling



- Collisions occur when different elements are mapped to the same cell



- Separate Chaining: let each cell in the table point to a linked list of entries that map there
- Separate chaining is simple, but requires additional memory outside the table

Map with Separate Chaining

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = **null then** {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ **null then** {k was found}
    n = n - 1
return t

Hash Table with Chaining

```
public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
  // a fixed capacity array of UnsortedTableMap that serve as buckets
  private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
  public ChainHashMap( ) { super( ); }
  public ChainHashMap(int cap) { super(cap); }
  public ChainHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable( ) {
    table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
  }
  /** Returns value associated with key k in bucket with hash value h, or else null. */
  protected V bucketGet(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    return bucket.get(k);
  }
  /** Associates key k with value v in bucket with hash value h; returns old value. */
  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);     // size may have increased
    return answer;
  }
```

Hash Table with Chaining, 2

```
/** Removes entry having key k from bucket with hash value h (if any). */
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());      // size may have decreased
  return answer;
}
/** Returns an iterable collection of all 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 (table[h] != null)
      for (Entry<K,V> entry : table[h].entrySet())
        buffer.add(entry);
  return buffer;
}
}
```

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by **placing the colliding item in the next** (circularly) **available table cell**
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

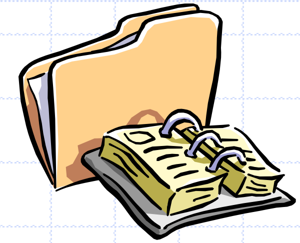
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return null
    else if c.getKey () = k
      return c.getValue()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until p = N
  return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***DEFUNCT***, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item ***DEFUNCT*** and we return element o
  - Else, we return ***null***
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***DEFUNCT***, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Probe Hash Map in Java

```java
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
  public ProbeHashMap() { super(); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable() {
    table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
  }
  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }
```

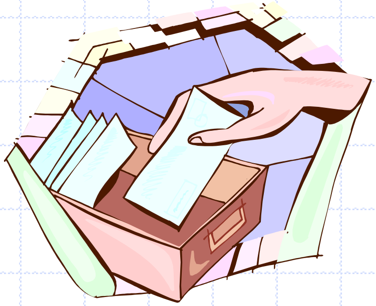
Probe Hash Map in Java, 2

```java
/** Returns index with key k, or −(a+1) such that k could be added at index a. */
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)) {                  // may be either empty or defunct
      if (avail == −1) avail = j;          // this is the first available slot!
      if (table[j] == null) break;         // if empty, search fails immediately
    } else if (table[j].getKey( ).equals(k))
      return j;                            // successful match
    j = (j+1) % capacity;                  // keep looking (cyclically)
  } while (j != h);                        // stop if we return to the start
  return −(avail + 1);                     // search has failed
}
/** Returns value associated with key k in bucket with hash value h, or else null. */
protected V bucketGet(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                  // no match found
  return table[j].getValue( );
}
```

Probe Hash Map in Java, 3

```java
  /** Associates key k with value v in bucket with hash value h; returns old value. */
  protected V bucketPut(int h, K k, V v) {
    int j = findSlot(h, k);
    if (j >= 0)                                   // this key has an existing entry
      return table[j].setValue(v);
    table[-(j+1)] = new MapEntry<>(k, v);         // convert to proper index
    n++;
    return null;
  }
  /** Removes entry having key k from bucket with hash value h (if any). */
  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 this slot as deactivated
    n--;
    return answer;
  }
  /** Returns an iterable collection of all 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;
  }
}
```

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

$$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

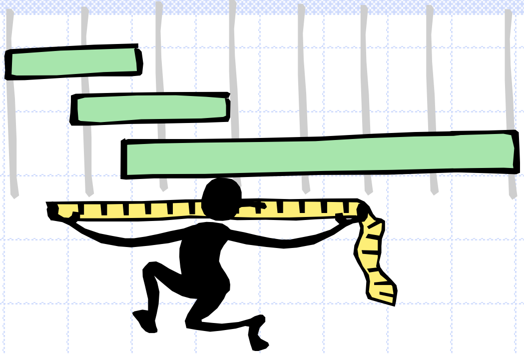
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing



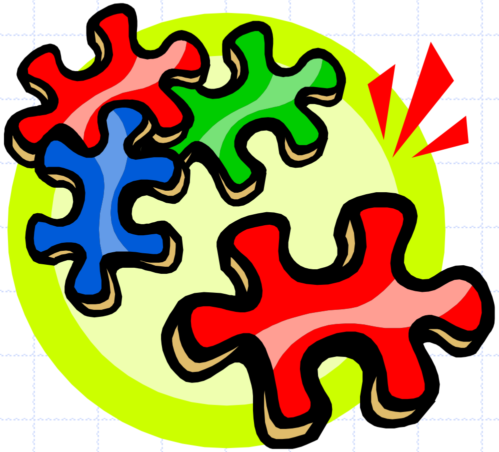
- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$1 / (1 - \alpha)$$

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Sets and Multimaps



Definitions

- A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
  - Elements of a set are like keys of a map, but without any auxiliary values.
- A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.
- A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.
  - For example, the index of a book maps a given term to one or more locations at which the term occurs.

Set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e\colon\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

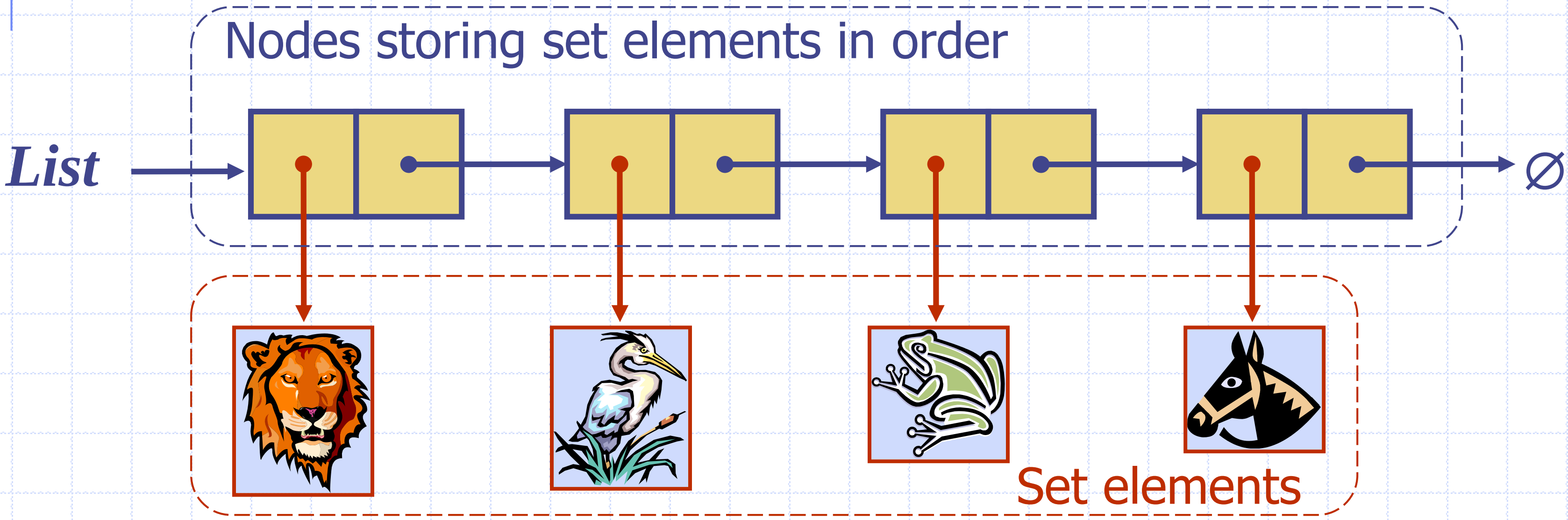
addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Storing a Set in a List

- We can implement a set with a list
- Elements are stored sorted according to some canonical ordering
- The space used is $O(n)$



Generic Merging

- Generalized merge of two sorted lists A and B
- Template method genericMerge
- Auxiliary methods
  - aIsLess
  - bIsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        a ← A.first().element(); b ← B.first().element()
        if a < b
            aIsLess(a, S); A.remove(A.first())
        else if b < a
            bIsLess(b, S); B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first()); B.remove(B.first())
    while ¬A.isEmpty()
        aIsLess(a, S); A.remove(A.first())
    while ¬B.isEmpty()
        bIsLess(b, S); B.remove(B.first())
    return S
```

Using Generic Merge for Set Operations

- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time

Multimap

- A **multimap** is similar to a map, except that it can store multiple entries with the same key
- We can implement a multimap M by means of a map M′
  - For every key k in M, let E(k) be the list of entries of M with key k
  - The entries of M′ are the pairs (k, E(k))

Mulitmaps

get(k): Returns a collection of all values associated with key k in the multimap.

put(k, v): Adds a new entry to the multimap associating key k with value v, without overwriting any existing mappings for key k.

remove(k, v): Removes an entry mapping key k to value v from the multimap (if one exists).

removeAll(k): Removes all entries having key equal to k from the multimap.

size( ): Returns the number of entries of the multiset (including multiple associations).

entries( ): Returns a collection of all entries in the multimap.

keys( ): Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).

keySet( ): Returns a nonduplicative collection of keys in the multimap.

values( ): Returns a collection of values for all entries in the multimap.

Java Implementation

```
public class HashMultimap<K,V> {
  Map<K,List<V>> map = new HashMap<>();   // the primary map
  int total = 0;                          // total number of entries in the multimap
  /** Constructs an empty multimap. */
  public HashMultimap() { }
  /** Returns the total number of entries in the multimap. */
  public int size() { return total; }
  /** Returns whether the multimap is empty. */
  public boolean isEmpty() { return (total == 0); }
  /** Returns a (possibly empty) iteration of all values associated with the key. */
  Iterable<V> get(K key) {
    List<V> secondary = map.get(key);
    if (secondary != null)
      return secondary;
    return new ArrayList<>();             // return an empty list of values
  }
```

Java Implementation, 2

```
/** Adds a new entry associating key with value. */
void put(K key, V value) {
  List<V> secondary = map.get(key);
  if (secondary == null) {
    secondary = new ArrayList<>();
    map.put(key, secondary);     // begin using new list as secondary structure
  }
  secondary.add(value);
  total++;
}
/** Removes the (key,value) entry, if it exists. */
boolean remove(K key, V value) {
  boolean wasRemoved = false;
  List<V> secondary = map.get(key);
  if (secondary != null) {
    wasRemoved = secondary.remove(value);
    if (wasRemoved) {
      total--;
      if (secondary.isEmpty())
        map.remove(key);     // remove secondary structure from primary map
    }
  }
  return wasRemoved;
}
```

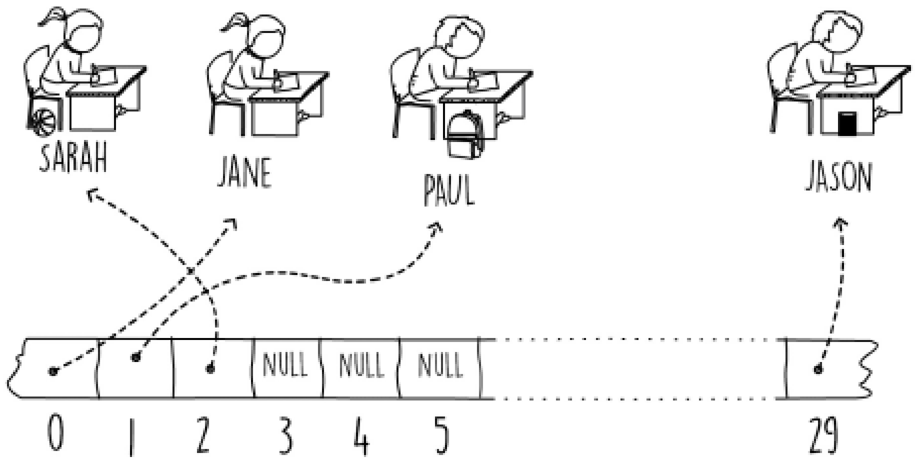
Java Implementation, 3

```java
/** Removes all entries with the given key. */
Iterable<V> removeAll(K key) {
  List<V> secondary = map.get(key);
  if (secondary != null) {
    total -= secondary.size();
    map.remove(key);
  } else
    secondary = new ArrayList<>();      // return empty list of removed values
  return secondary;
}
/** Returns an iteration of all entries in the multimap. */
Iterable<Map.Entry<K,V>> entries() {
  List<Map.Entry<K,V>> result = new ArrayList<>();
  for (Map.Entry<K,List<V>> secondary : map.entrySet()) {
    K key = secondary.getKey();
    for (V value : secondary.getValue())
      result.add(new AbstractMap.SimpleEntry<K,V>(key,value));
  }
  return result;
}
}
```

Hash functions – a simple view

Direct Addressing

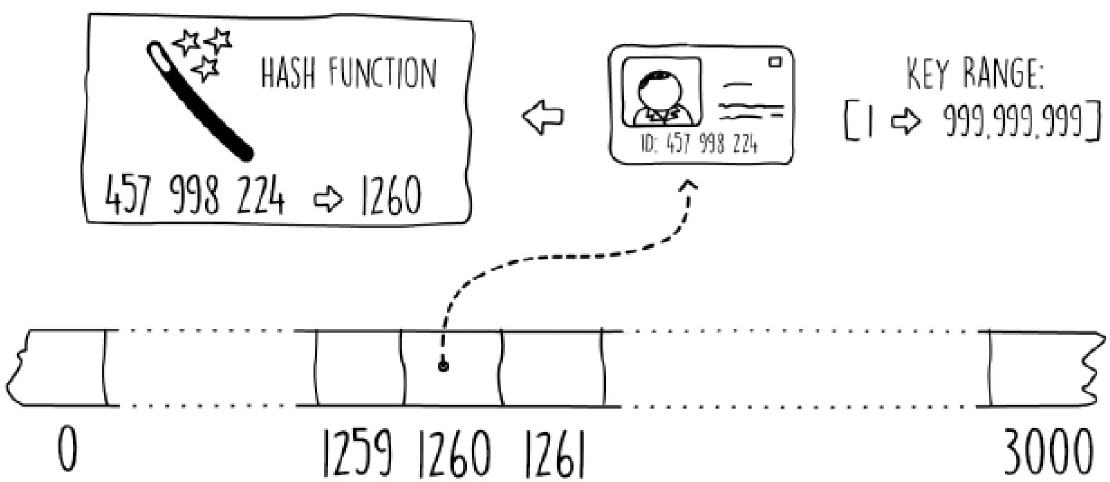
- Access student record using a key (desk number)
- Each array position can contain one student record



- arrayIndex= deskNumber – 1
- arrayStudent[arrayIndex) returns student record
- The **load factor** is a metric showing how fully utilized our
- data structure is. Load factor is 1 when fully utilized.

Direct Addressing

- Direct addressing not efficient for large arrays
- For example, to address passport number, a nine-digit numeric range, array's size would be 1,000,000,000 - low load factor.
- Solution: mapping our nine-digit numeric range into a four-digit one.

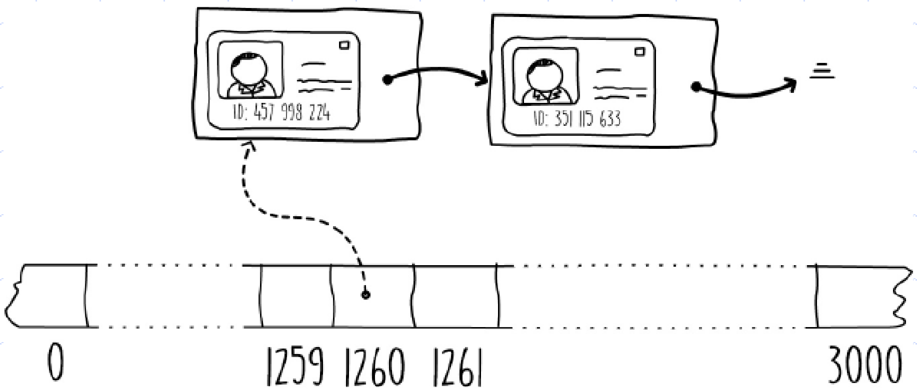


Hash Function

- Mapping can be done by a **hash function**.
- A hash function would **accept a key** (our passport number) and **return an array index** within the size of our array.
- Using a hash function enables us to use a much smaller array and saves us a lot of memory.
- However, there is a catch - forcing a bigger key space into a smaller one, there is a risk that multiple keys map to the same hashed array index.
- This is what is called a **collision**; we have a key hash to an already filled position.

Chaining

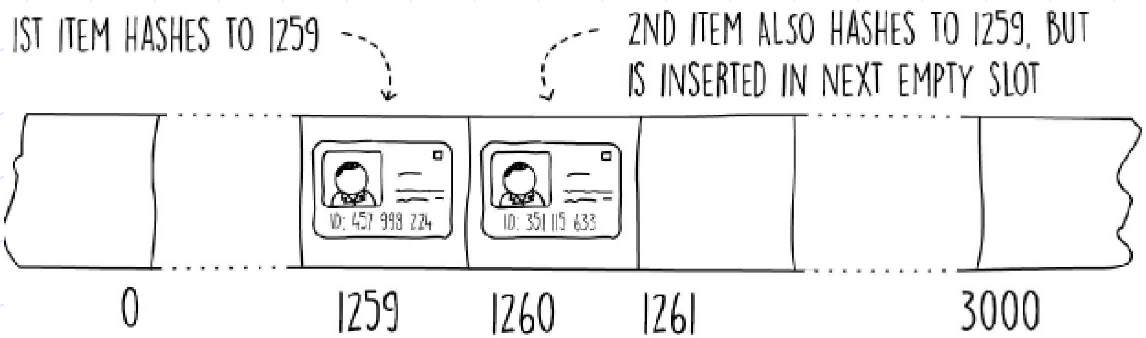
- One common solution to deal with collisions is a technique called chaining - the hash table data is stored outside the actual array itself.
  - Linked lists are used to chain multiple entries in one hash slot.



- Searching for a particular key requires first locating the array slot, and then traversing the linked list, one item at a time, looking for the required key until there is a match or the end of the list is reached. – One average the performance is close to O(1).

Open Addressing – Linear Probing

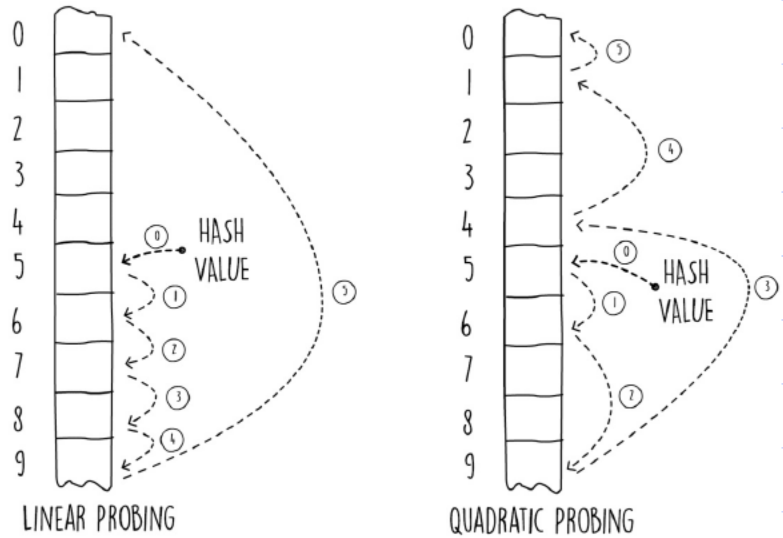
- In open addressing, all items are stored in the array itself, making the structure static with a maximum load factor limit of 1.
- To insert in an open-addressed hash table, we hash the key and simply insert the item in the hash slot, the same as a normal hash table.
  - If the slot is already occupied, we search for another empty slot and insert the item in it.
  - The manner in which we search for another empty slot is called the **probe sequence**.
  - **Linear probing** strategy looks for the **next available slot**.



Open Addressing – Quadratic Probing

- Linear probing suffers from a problem called clustering.
  - This occurs when a long succession of non-empty slots develop, degrading the search and insert performance.
- One way to improve this is to use a technique called **quadratic probing** - probe for the next empty slot using a quadratic formula of the form $h + (ai + bi^2)$, where h is the initial hash value, and a and b are constants.

```
array[(hashValue + a*i + b*i^2) mod s]
```



Implementing Hash Functions

- A simple technique to implement a hash function is what is known as the **remainder method**.
- The hash function simply takes in any numeric key, divides it by the table size (size of the array), and **uses the resultant remainder as the hash value**.
- This value can then be **used as an index on the array**.

```
public int hashKey(Integer key, int tableSize) {
    return key % tableSize;
}
```

- The reminder method might result in many collisions if care is not taken when choosing an appropriate table size.
- A better choice of a table size is to **use a prime number**, ideally not too close to the power of 2.

Multiplication method

- In this method, we multiply the key by a constant double value, k, in the range 0 < k < 1.
- We then extract the fractional part from the result and multiply it by the size of our hash table.
- The hash value is then the floor of this result:

$$hash(x) = \lfloor s(xk \bmod 1) \rfloor$$

- Where:
  - k is a decimal in the range between 0 and 1
  - s is the size of the hash table
  - x is the key

Multiplication method

- The following code shows an implementation for the multiplication hash function:

```
private double k;
public MultiplicationHashing(double k) {
this.k = k;
}
public int hashKey(Integer key, int tableSize) {
    return (int) (tableSize * (k * key % 1));
}
```

Universal Hashing

- Universal hashing works by choosing **a random function** from a universal set of hash functions at the start of execution.
  - the same sequence of keys will produce a **different sequence of hash values** on every execution.
  - A set of hash functions, H, with size n, where each function maps a universe of keys ∪ to a fixed range of [0, s), is said to be universal for all pairs, where a, b ∈ ∪, a ≠ b and the probability that h(a) = h(b), h ∈ H is less than or equal to n/s.
  - We can construct our set of universal hash functions by using two integer variables, **i** in a range of [1, p), and **j** in a range of [0, p), where **p** is a prime number larger than any possible value of the input key universe.
  - We can then generate any hash function from this set using:

$$h_{ij}(x) = ((ix + j) mod\, p) mod\, s$$

  - Where s is the size of the hash table and x is the key.

Universal Hashing

```
public UniversalHashing() {
    j = BigInteger.valueOf((long) (Math.random() * p));
    i = BigInteger.valueOf(1 + (long) (Math.random() * (p - 1L)));
}

public int hashKey(Integer key, int tableSize) {
    return i.multiply(BigInteger.valueOf(key)).add(j)
    .mod(BigInteger.valueOf(p))
    .mod(BigInteger.valueOf(tableSize))
    .intValue();
}
```

- Universal hashing provides us with good results, **minimizing collisions**, and is **immune to malicious attacks**, since the function parameters are chosen at random.