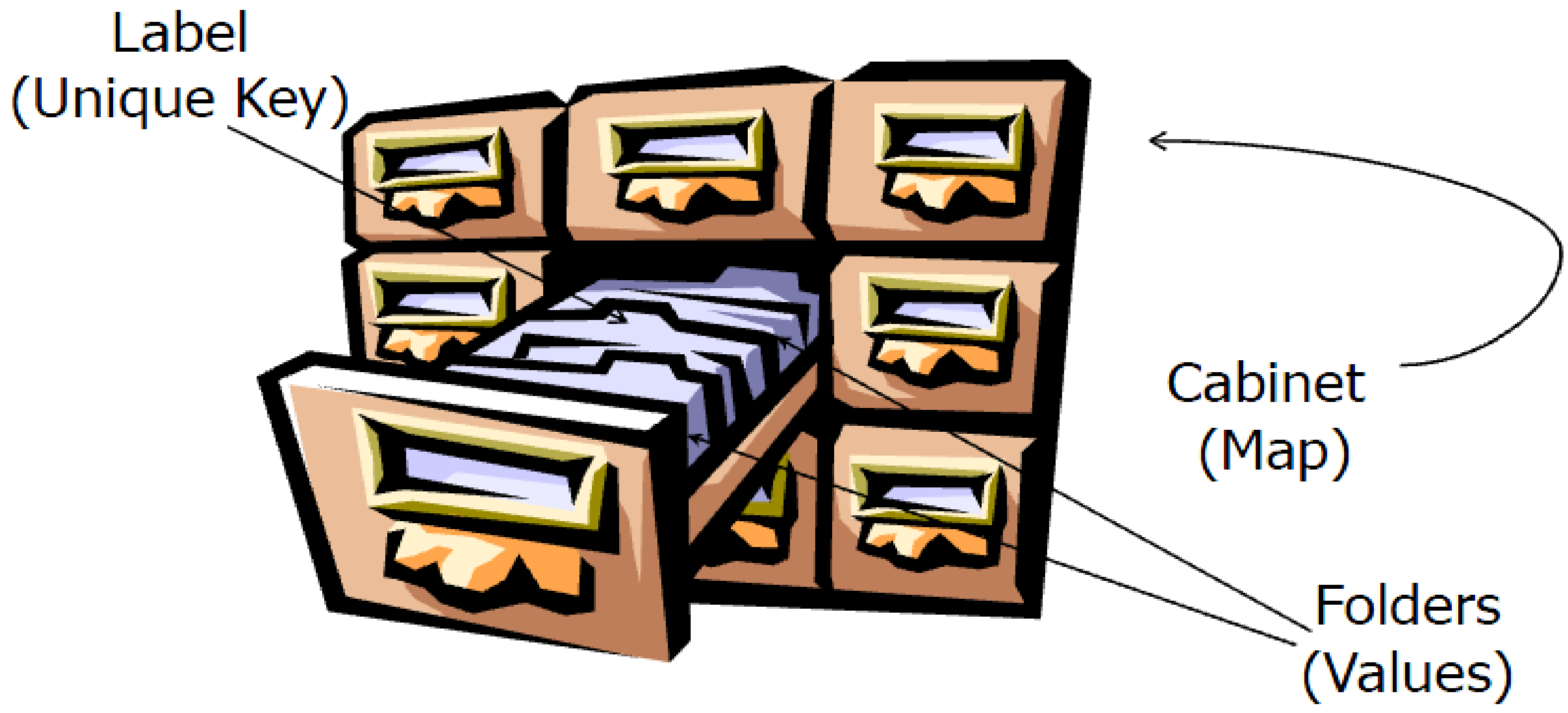
COMP6481 Programming and Problem Solving

1. Map
2. Hash Table

Chap 10

- Map

Today



Maps

- A map models a searchable collection of key-value entries.
- A map allows us to store elements in a way that speeds up locating them through the utilization of keys.
- The main operations of a map are searching, inserting, and deleting items.
- Multiple entries with the same key are not allowed. In other words, the keys in a map are unique.
- Applications:
  - address book
  - student-record database

The Map ADT

A map supports the following methods:

- **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, replace the value associated with k with v and return that old value
- **remove(k):** if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- **entrySet():** return an iterable collection containing all the key-value entries in M

The Map ADT

A map supports the following methods (continues):

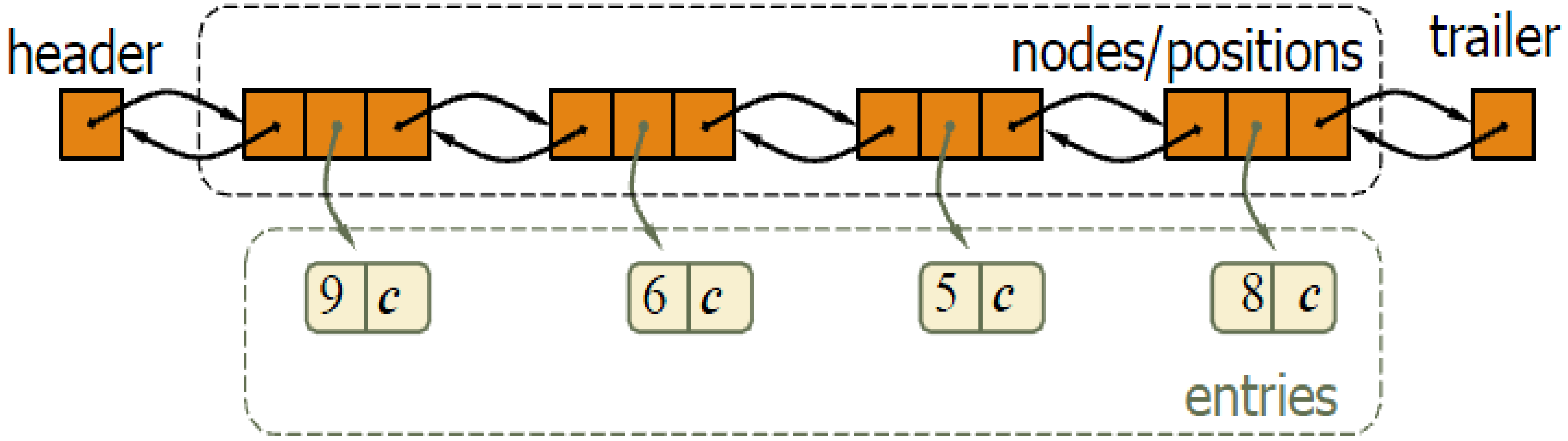
- keySet(): return an iterable collection of all the keys in M
- values(): return an iterable collection of all the values in M
- size(): return the number of entries in M
- isEmpty(): test whether M is empty

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 efficiently implement a map using an unsorted list
  - We store the items of the map in a list *S* (based on a doubly-linked 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 }
```

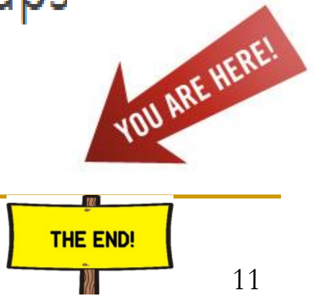
The remove(k) Algorithm

```
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 $O(n)$ time since we need to find out before adding the item if an entry with the key exists (in such case we only replace the value)
  - 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 sizes.



- Ordered Map

Ordered Maps

- In some applications, simply looking up values based on associated keys is not enough.
- The entries may need to be sorted in the map according in **total order**.
- An ordered map would have the usual operations that a map has, but also maintains an order relation for the keys.
- Keys are assumed to come from a total order; a comparator can be used to provide the needed order between the keys.

Ordered Maps

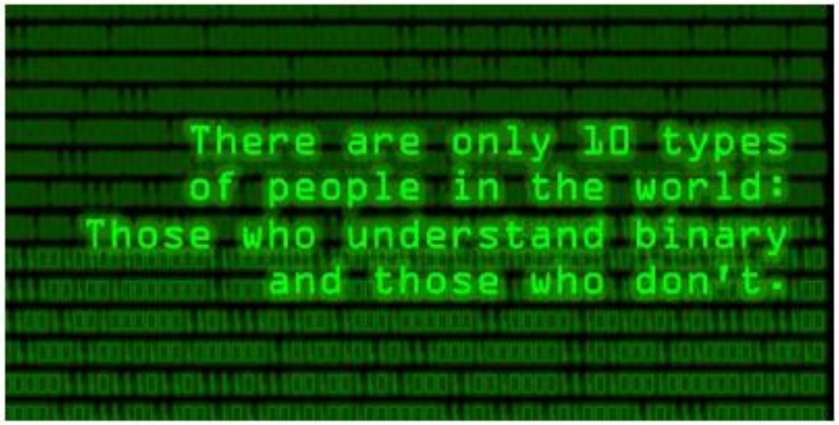
Since the entries are sorted, some additional methods can efficiently be provided:

- firstEntry(): return the entry with smallest key, or null if the map is empty
- lastEntry(): return the entry with largest key, or null if the map is empty
- floorEntry(k): return the entry with the largest key $\leq k$
- ceilingEntry(k): return the entry with the smallest key $\geq k$
  - These two operations also return null if the map is empty

Binary Search

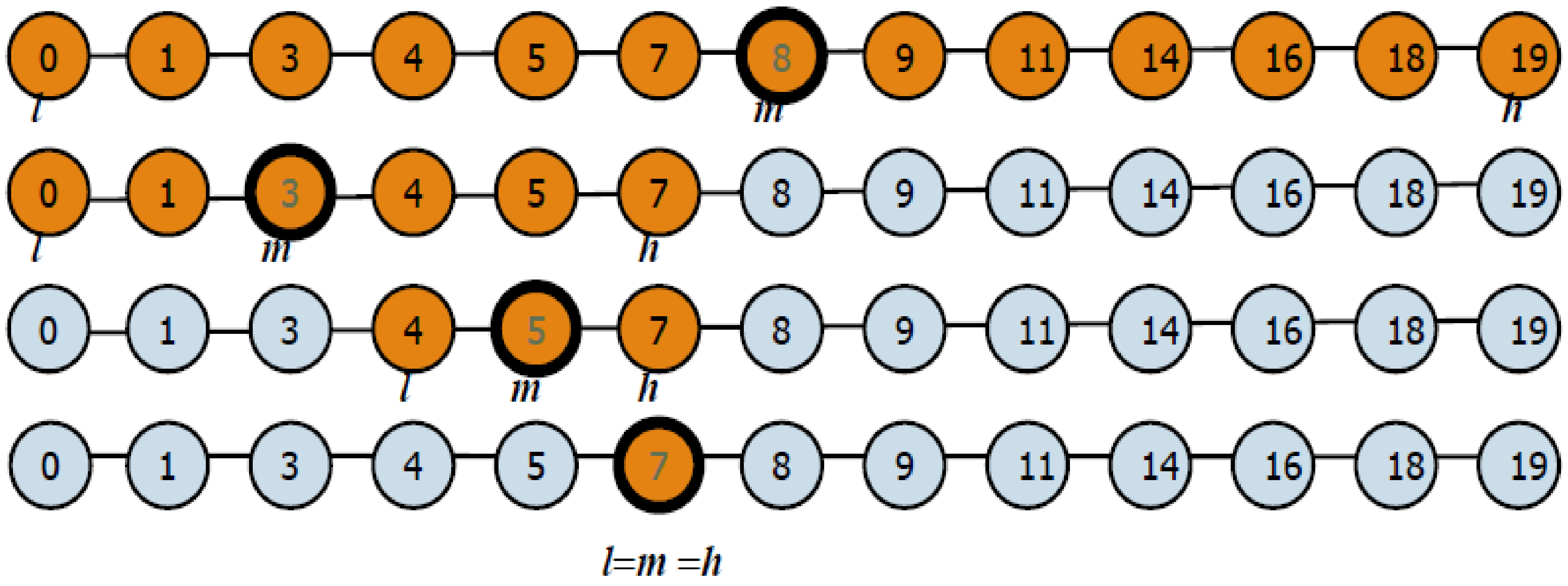
Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map implemented by means of an array-based sequence, sorted by key

- similar to the high-low game
- at each step, the number of candidate items is halved
- terminates after O(log n) steps



Binary Search

Example: find(7)



Performance of Ordered Maps

- We can store the items in an array-based sequence, sorted by key.
- Performance:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- Ordered maps are effective only for small size maps or for maps on which searches are the most common operations, while insertions and removals are rarely performed.

- Dictionary

Dictionary ADT

- Like a map, a dictionary stores *key-value* entries.
- Similarly, a dictionary allows the keys and the values to be of any object types.
- However, in contrast to maps, where the keys must be unique, a dictionary allows for multiple entries to have the same keys.
- This is very much like an English dictionary, for instance, which allows for a multiple definitions for the same word.
- The main operations of a dictionary are searching, inserting, and deleting items.

Dictionary ADT

As an ADT, an (unordered) ***dictionary D*** supports the following methods:

- get(***k***): if the dictionary has an entry with key ***k***, return it, else, return null. If more than one entry exists with key ***k*** then *arbitrarily* return one of them.
- getAll(***k***): return an iterable collection of all entries with key ***k***.
- put(***k***, ***v***): insert an entry with key ***k*** and value ***v*** and return that entry.
- remove(***e***): remove the entry ***e*** from the dictionary and return it; error occurs if ***e*** is not in the dictionary.
- entrySet(): return an iterable collection of the entries in the dictionary.
- size(), isEmpty()

Example

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	null	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
remove(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(5)	null	(7,B),(2,C),(8,D),(2,E)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order

- 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

A List-Based Dictionary …

- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

The getAll and put Algorithms

```
Algorithm getAll(k)
 Create an initially-empty list L
 for e: D do
  if e.getKey() = k  then
   L.addLast(e)
 return L

Algorithm put(k,v)
   Create a new entry e = (k,v)
   S.addLast(e)  {S is unordered}
 return e
```

The remove Algorithm

```
Algorithm remove(e):
 { We don’t assume here that e stores its
 position in S }
   B = S.positions()
   while B.hasNext() do
      p = B.next()
      if p.element() = e then
         S.remove(p)
   return e

return null      {there is no entry e in D}
```

Hash Table Implementation

- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.
- In that case, the dictionary methods can be achieved in $O(1)$.

Search Table

- A search table is a dictionary implemented by means of a <u>sorted array</u>
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - get takes $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove takes $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal

Search Table

- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed.

- Hash Table

Recall the Map ADT

A map supports the following methods:

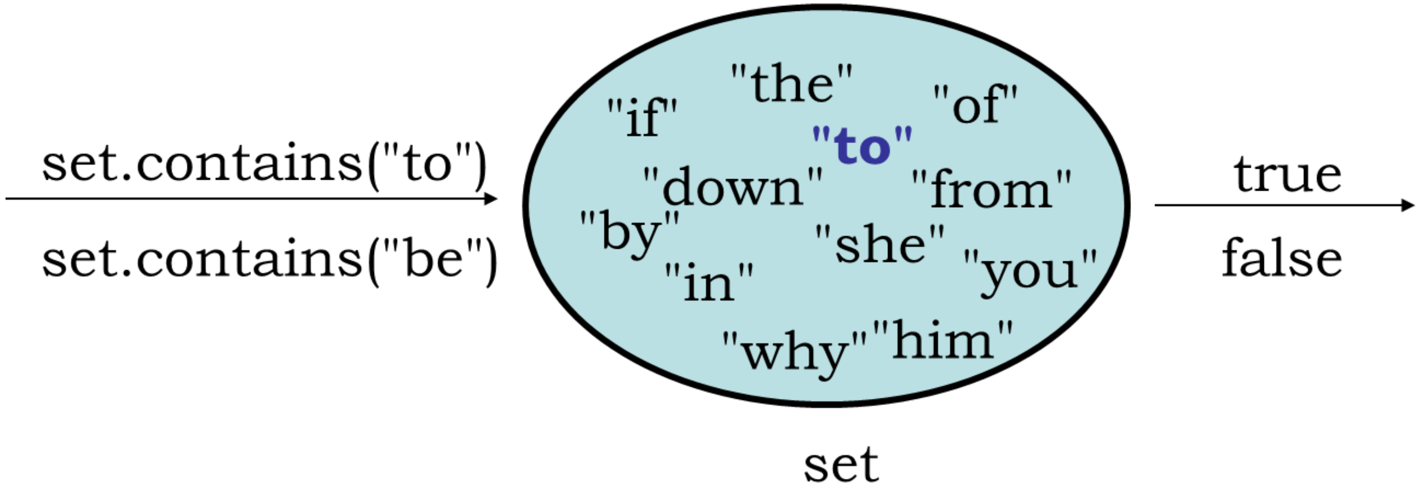
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, replace the value associated with k with v and return that old value
- remove(k): if the map M has an entry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

- Performance:
  - put takes $O(n)$ time since we need to find out before adding the item if an entry with the key exists (in such case we only replace the value)
  - 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 sizes.

SET ADT

- set: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - The client doesn't think of a set as having indexes; we just add things to the set in general and don't worry about order



SET ADT

Integer Set ADT

- Let's think about writing our own implementation of a set
  - To simplify the problem, we only store int in our set for now
  - As is (usually) done in the Java Collection Framework, we will define sets as an ADT by creating a Set interface
  - Core operations are: add, contains, remove.

```
public interface IntSet {
        void add(int value);
        boolean contains(int value);
        void clear();
        boolean isEmpty();
        void remove(int value);
        int size();
    }
```

SET ADT

Unfilled Array Set

- Consider storing a set in an unfilled array.
  - It doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly
  - What would make a good ordering for the elements?
- If we store them in the next available index, as in a list, ...
  - set.add(9);
    set.add(23);
    set.add(8);
    set.add(-3);
    set.add(49);
    set.add(12);

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	9	23	8	-3	49	12	0	0	0	0
size	6									

  - How efficient is add? contains? remove? (1, N, N)

SET ADT

Sorted Array Set

- Suppose we store the elements in an unfilled array, but in *sorted* order rather than order of insertion
  - set.add(9);
    set.add(23);
    set.add(8);
    set.add(-3);
    set.add(49);
    set.add(12);

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	-3	8	9	12	23	49	0	0	0	0
size	6									

  - How efficient is add? contains? remove? (N, logN, N)
  - (You can do a O(log *N*) binary search to find elements in contains, and to find the proper index in add/remove; but add/remove still need to shift elements right/left to make room, which is O(*N*) on average.)

Definition

- Hashing: a technique that determines this index using only an entry's search key
- Hash function
  - Takes a search key and produces the integer index of an element in the hash table
  - Search key—maps, or hashes, to the index
- Example: Emergency telephone 911
- If you call from a land line, your tel. num. is the key in a search map of street addresses. Obviously, you want this search to find your location immediately---efficiency is critical
- Here we will introduce a technique called *hashing* that ideally can result in O(1) search times.

Hash Functions and Hash Tables

- To speed up searching for an entry, we can create an array of holders/buckets of these entries.
- The resulting data structure is hence a table.
- If insertion of entries is made such that each entry can be placed at only one possible location (bucket) in this data structure, then searching for the entry is consequently restricted to searching that particular bucket instead of searching the entire data structure.

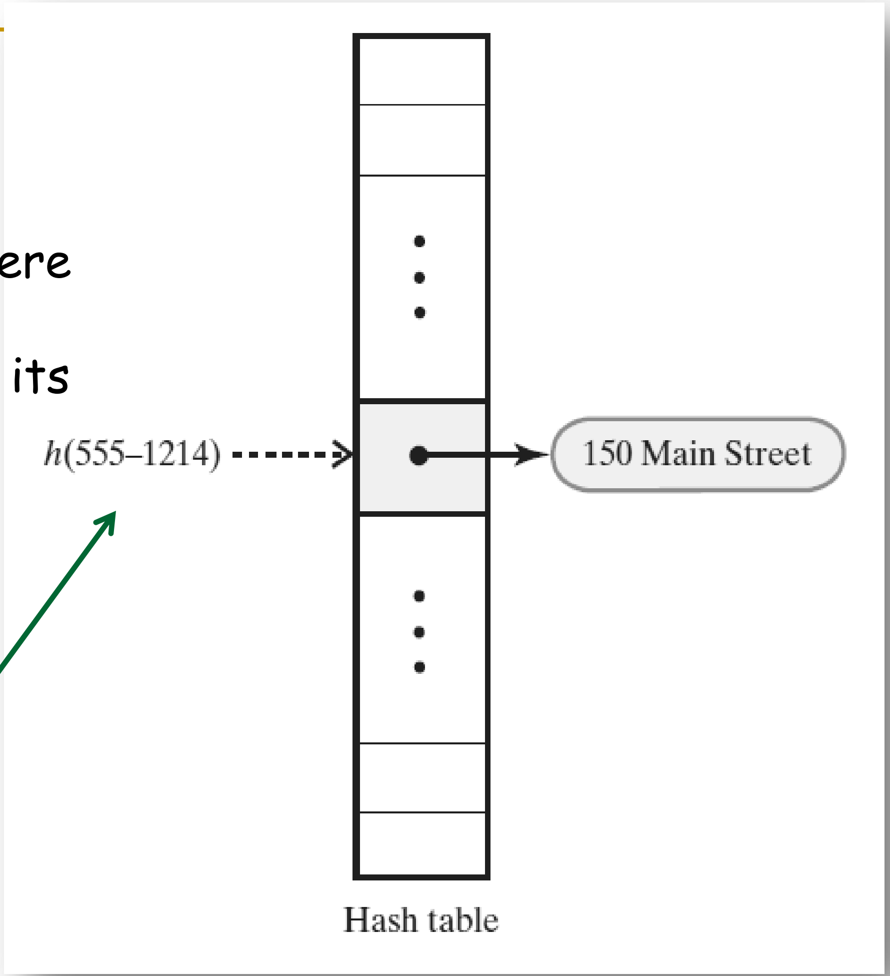
Ideal Hashing

- An emergency system for small town where every tel. num. begins with 555

Hash function *h* that convert a tel. num. to its last four digits

Example:

h(555-1214)= 1214



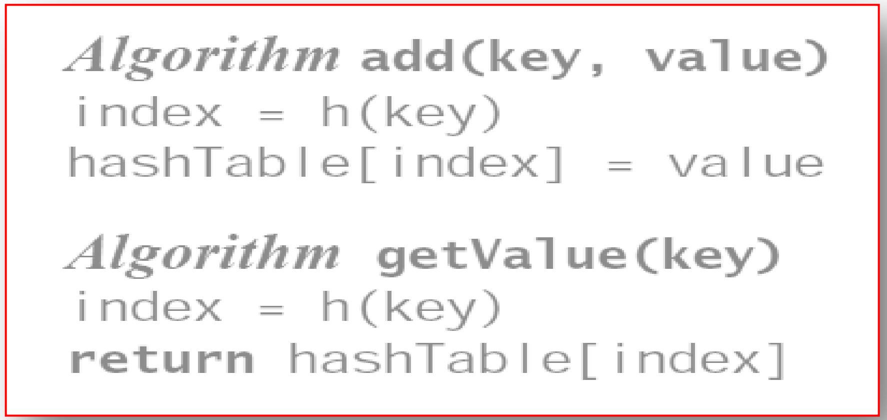
hash function indexes its hash table

If `hashTable` is the hash table, we would place a reference to the street address associated with this tel. num. in `hashTable[1214]`.

If the cost of evaluating the hash function is low, adding an entry to the array hashTable is an O(1) operation. We did not search the array `hashTable`

Ideal Hashing (con't)

- Simple algorithms for the Map operations that add and retrieve



```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value

Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

- Will this algorithm always work?
  - If we know all possible search keys: e.g. search key from: 555-0000 to 555-9999,
  - so the hash will produce indices from 0-9999
  - If the array `hashTable` has 10, 000 elets, each tel.num. will correspond to one unique elet. in `hashTable`.
  - That elet. references the appropriate street address.
- This scenario describes the ideal case of hashing, and the hash function is a perfect hash function

- ✓ A perfect hash function maps each search key into a different integer that is suitable as an index to the hash table

Hash Functions and Hash Tables

- In a best scenario, where each bucket can contain only one entry, searching for any entry can be done in $O(1)$ time.
- That can be achieved if the keys are unique integers in the range [0 .. N-1], where N is the size of the array.

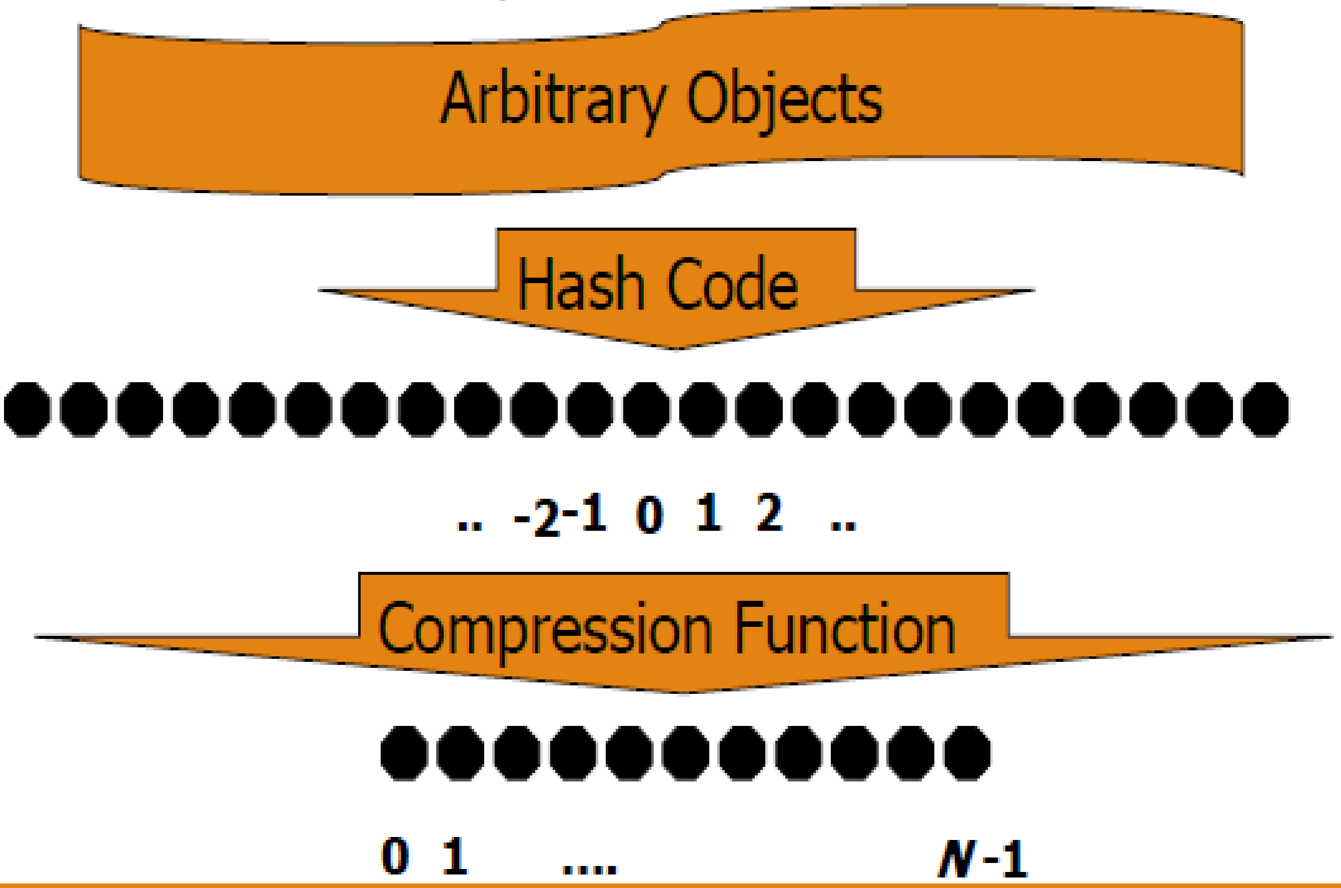
- This however have two **drawbacks**:
  - If N is actually much larger than the actual number of stored elements, then a lot of space is wasted
  - Keys must be integers, which may not be the case all the time

Hash Functions and Hash Tables

- Instead, have an array of limited size N, then somehow convert the keys to a value between 0 and N-1.
- The method/function performing the conversion is referred to as "hash function".
- The entire data structure (that is the array and the holders/buckets) are referred to as "hash table".
- Notice that now multiple keys may be converted to the same index value, which may result in multiple entries being placed in the same bucket. This is referred to as "collision".

Hash Functions

The two parts of a hash function: a hash code and a compression function to a hash value, can be illustrated as follows:

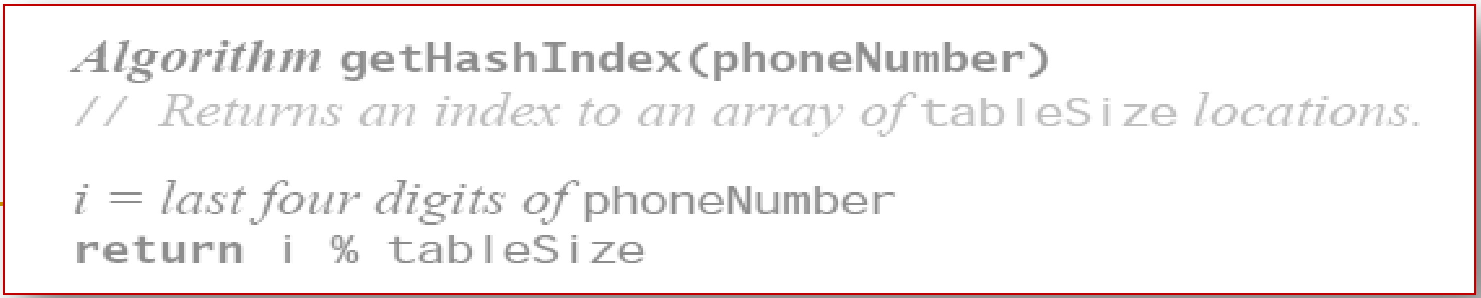


Typical Hashing

- The previous perfect hash function needs a hash table large enough to produce 10, 000 diff. indices between 0 and 9999 from 10,000 possible search keys. This hash table is always full if every tel. num. in the 555 exchange is assigned
- Although a full hash table is quite reasonable for this example, most has tables are not full and can even be sparse, e.i. few of their elets. Actually in use; e.g. small town with 700 tel. num., so most of the 10,000 location hash table would be unused, waste of space
  - if 700 num. were not sequential, we would need a different hash function if we want to use a smaller hash table.

Solution:

- Given a nonnegative int i, and hash table with n locations, the value of *i* modulo *n* ranges from 0 to n-1, since *i* is nonnegative, *i* modulo *n* is the integer remainder after dividing *i* by *n*. This value is a valid index for the hash table.



```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array of tableSize locations.

i = last four digits of phoneNumber
return i % tableSize
```

Hashing

HashSet Implementation

```
public class HashIntSet implements IntSet {
    private int[] elements;
    ...
    public void add(int value) {
        elements[hash(value)] = value;
    }
    public boolean contains(int value) {
        return elements[hash(value)] == value;
    }
    public void remove(int value) {
        elements[hash(value)] = 0;
    }
}
```

– Runtime of add, contains, and remove: O(1) !!

Any problems with this approach?

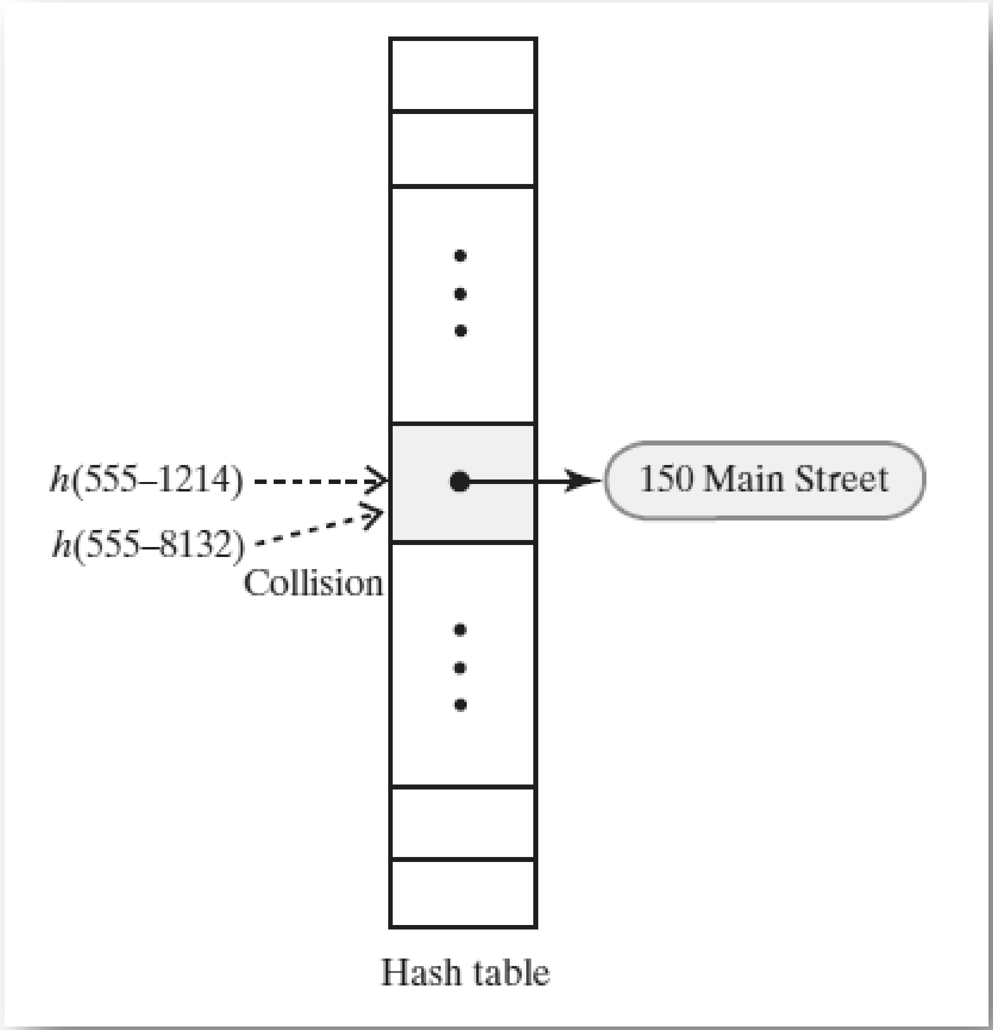
Typical Hashing

- Typical hash functions—perform two steps:
  1. Convert search key to an integer called the hash code.
  2. Compress hash code into the range of indices for hash table.
- Often search key is not integer, and frequently it is a string, so
  1. hash function first converts the key to integer has code,
  2. next it transforms that integer into one that suitable as index to the particular hash table

- Typical hash functions are not perfect,
  - Can allow more than one search key to map into a single index
  - Causes a collision in the hash table
- Example
  - Consider tableSize = 101
  - `getHashIndex(555-1214)` = 52
  - `getHashIndex(555-8132)` = 52 also!!!

Typical Hashing

collision caused by the hash function *h*



Hash Functions

- A good hash function should
  - Minimize collisions
  - Be fast to compute
- To reduce the chance of a collision
  - Choose a hash function that distributes entries uniformly throughout hash table.
- First consider how to convert a search key to *int*. Realize that a search key can be either a primitive or an instance of class

Computing Hash Codes

- Java's base class `Object` has a method `hashCode` that returns an integer hash code
- Since every class is a subclass of `Object`, all classes inherent this method. But unless a class overrides `hashCode`, the method will return an int value based on memory address of the object used to invoke it. This default hash code is usually not appropriate for hashing, because equal but distinct objects will have different hash codes.
- To be useful as a map implementation, hashing must map equal objects into same location in a hash table
  - A class should define its own version of `hashCode`

Guidelines for method `hashCode`

- If a class overrides the method `equals`, it should override `hashCode`.
- If the method `equals` considers two objects equals, `hasCode` must return the same value for both objects
- If you call an object's `hashCode` more than once during the execution of a program, and if the object's data remains the same during this time, `hashCode` must return the same hash code
- An object's hash code during one execution of a program can differ from its has code during another execution of the same program.

Computing Hash Codes (1/2)

- A hash code for a string
  - Using a character's Unicode integer is common--→see slide
  - Better approach: multiply Unicode value of each character by factor based on character's position, then sum

- Hash code for a string example:

$$u_0 g^{n-1} + u_1 g^{n-2} + \ldots + u_{n-2} g + u_{n-1}$$

- Java code to do this:

```
int hash = 0;
int n = s.length();
for (int i = 0; i < n; i++)
   hash = g * hash + s.charAt(i);
```

Hash Codes (cont.)

Polynomial accumulation:

- We then evaluate the polynomial

  $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{k-1} z^{k-1}$

  for some fixed value z.

- This would effectively take care of the order of the bits.

- *Horner's rule*: the polynomial

  $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{k-1} z^{k-1}$

  can actually be written as:

  $p(z) = a_0 + z(a_1 + z(a_2 + \ldots + z(a_{k-3} + z(a_{k-2} + za_{k-1})) \ldots))$

Hash Codes (cont.)

Polynomial accumulation:

- Studies suggested that a good value of z (to reduce collisions) would be 33, 37, 39 or 41.
- For instance, 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 (n successive computation, each from the previous one in $O(1)$ time).
  - Many Java implementations use polynomial accumulation as the default hash function for strings.
  - Some Java implementations only apply polynomial accumulation to a fraction of the characters in long strings.

Computing Hash Codes (2/2)

- Basic hash code for string: Sum the unicode values for each letter in search key: works for two different search keys that never contains same letters, but search keys that consist of different arrangement of same letters, e.g: DUB, BUD, would have the same hash code.
- A better hash code for string: multiply the Unicode value of each character by factor based character's position within the string. The hash code is then the sum of these products

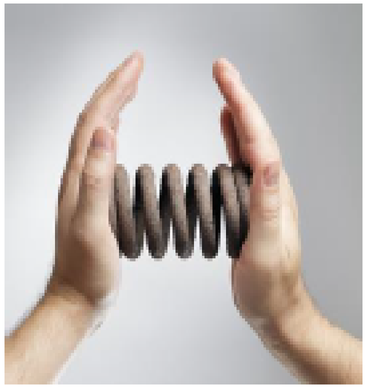
Example

- Calculate the hash code for the string *Java* when *g* is 31, and compare your result with the value of the expression `“Java”.hashCode()`

Compression Functions

- Hash codes may be out of the indices of the hash table's array, so compression is needed to convert these codes to integers within the [*0* .. *N-1*] range.

- A good compression function is the one that minimizes the possible number of collisions in a gives set of hash codes.



Compression Functions

- A simple compression function is the ***division method***.
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
    - This helps spread out the distribution of hashed values.
    - For instance, assume hash codes {200, 205, 210, 215, 220, ..., 600}. If *N* is chosen as 100, then each hash code will collide with three others. If however *N* is 101, then there will be no collisions.
    - The specifics have to do with number theory and belongs to the scope of other courses

Compression Functions

- A more sophisticated compression function is multiply-add-and-divide (MAD).

- Multiply, Add and Divide (MAD):
  - This method maps an integer y as follows:

    $$h_2(y) = (ay + b) \bmod N$$

    - a and b are nonnegative integers such that

      $$a \bmod N \neq 0$$

    - Otherwise, every integer would map to the same value b

Compression Functions

- Actually, a better MAD compression would be as follows:
  $h_2(y) = [(ay + b) \bmod p\ ] \bmod N$
  - p is a prime number larger than N
  - a and b are nonnegative integers randomly chosen from $[0 .. p\text{-}1]$ with $a > 0$

- The above MAD compression gets us closer to having a good hash function.
- A "good" hash function should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$. In other words, the entries will be "thrown" into the array uniformly.

Hash Code for a Primitive type

- If data type is `int`,
  - Use the key itself
- For `byte`, `short`, `char`:
  - Cast as `int` to get a hash code
- Other primitive types
  - Manipulate internal binary representations
    - If a search key is an integer of type `long`, it contains 64 bits, An *int* has 32bit, simply casting the 64 bits search key to an *int* or performing modulo 2^{32}, would lose its first 32 bits, so all keys that differ in only their first 32 bits will have the same hash code and collide.
    - Ignoring part of the search key can be a problem

Derive the hash code from the entire key. Do not ignore part of it

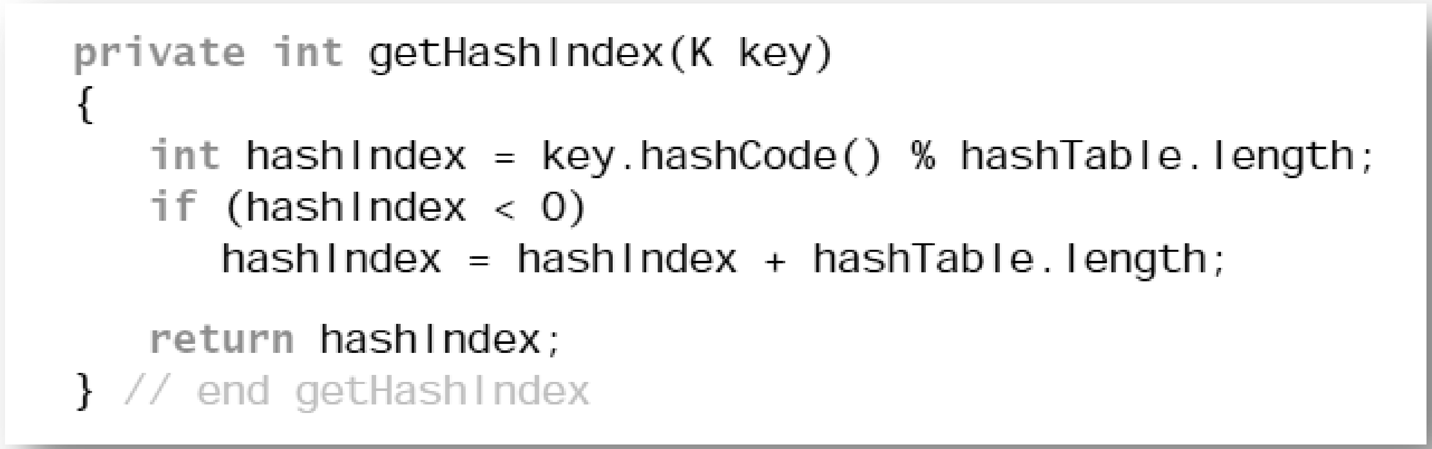
Compressing a Hash Code

- Common way to scale an integer
  - Use Java % operator, `code % n`

  `n` equals the size of the hash table, but not any n will do.
- Best to use an odd number for n
- Prime numbers often give good distribution of hash values

The size of a hash table should be prime number n greater that 2. Then if you compress a positive hash code *c* into an index for the table by using c% n, the indices will be distributed uniformly between 0 and n-1.

Compressing a Hash Code



```java
private int getHashIndex(K key)
{
   int hashIndex = key.hashCode() % hashTable.length;
   if (hashIndex < 0)
      hashIndex = hashIndex + hashTable.length;

   return hashIndex;
} // end getHashIndex
```

Hash function for the ADT Map

Hash Table: Collision handling

- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
- Separate Chaining

Resolving Collisions

- Definition: hash function maps search key into a location in hash table already in use
- Two choices:
  - Use another location in the hash table (open addressing) --sounds simple, but can lead to several complications.
  - Change the structure of the hash table so that each array location can represent more than one value--not as difficult as it might sound and can be a better choice for resolving collisions than open addressing

Open Addressing with

Linear probing

  - Resolves a collision during hashing by examining consecutive locations in hash table
  - Beginning at original hash index
  - Find the next available one

- Table locations checked make up *probe sequence: if a collision occurs at* `hashTable[k]`*, we see whether* `hashTable[k+1]` *is available, if not, we look at* `hashTable[k+2]`*, and so on, the table location that we consider in this search make the probe sequence.*
- If probe sequence reaches end of table, go to beginning of table (circular hash table)

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
    - All 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

- Deletion of entries would require considerable amount of shifting to adjust the array; however this can be avoided.
- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements.
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, v) is found, we replace it with the special item *AVAILABLE* and we return element v
  - Else, we return ***null***

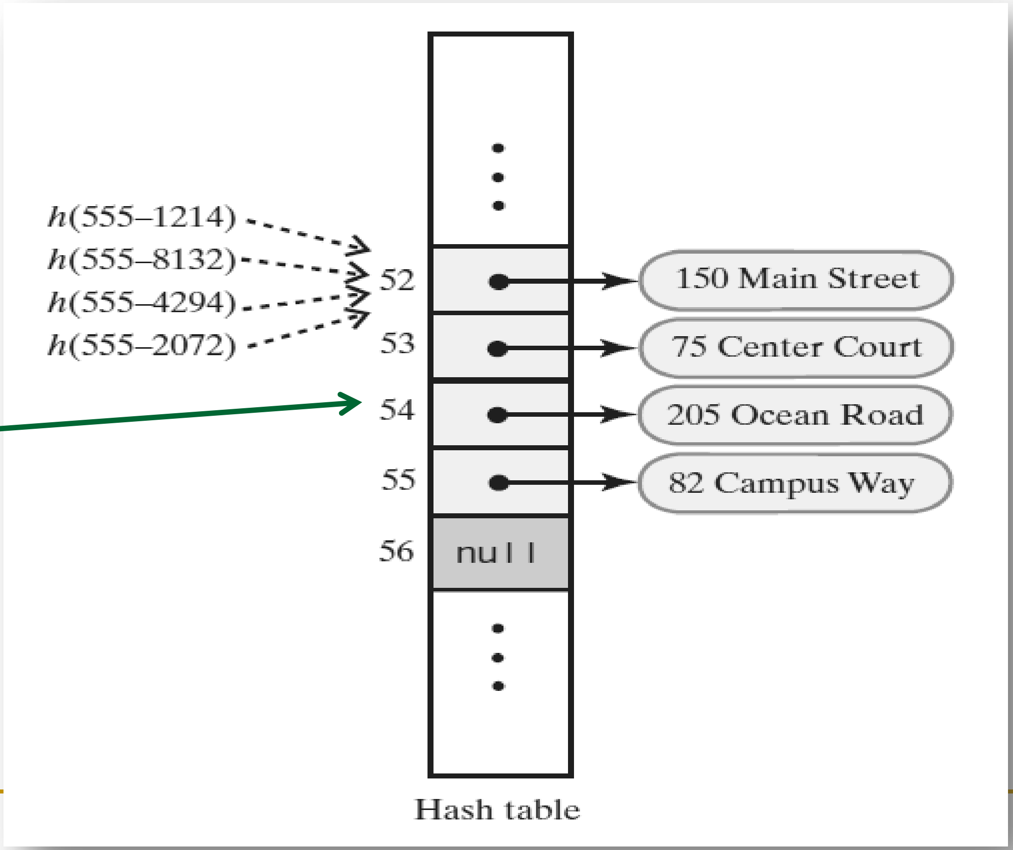
- put(k, v)
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - If a cell i is found, we store (k, v) in that cell i

Example Resolving Collisions

Additions that collide:

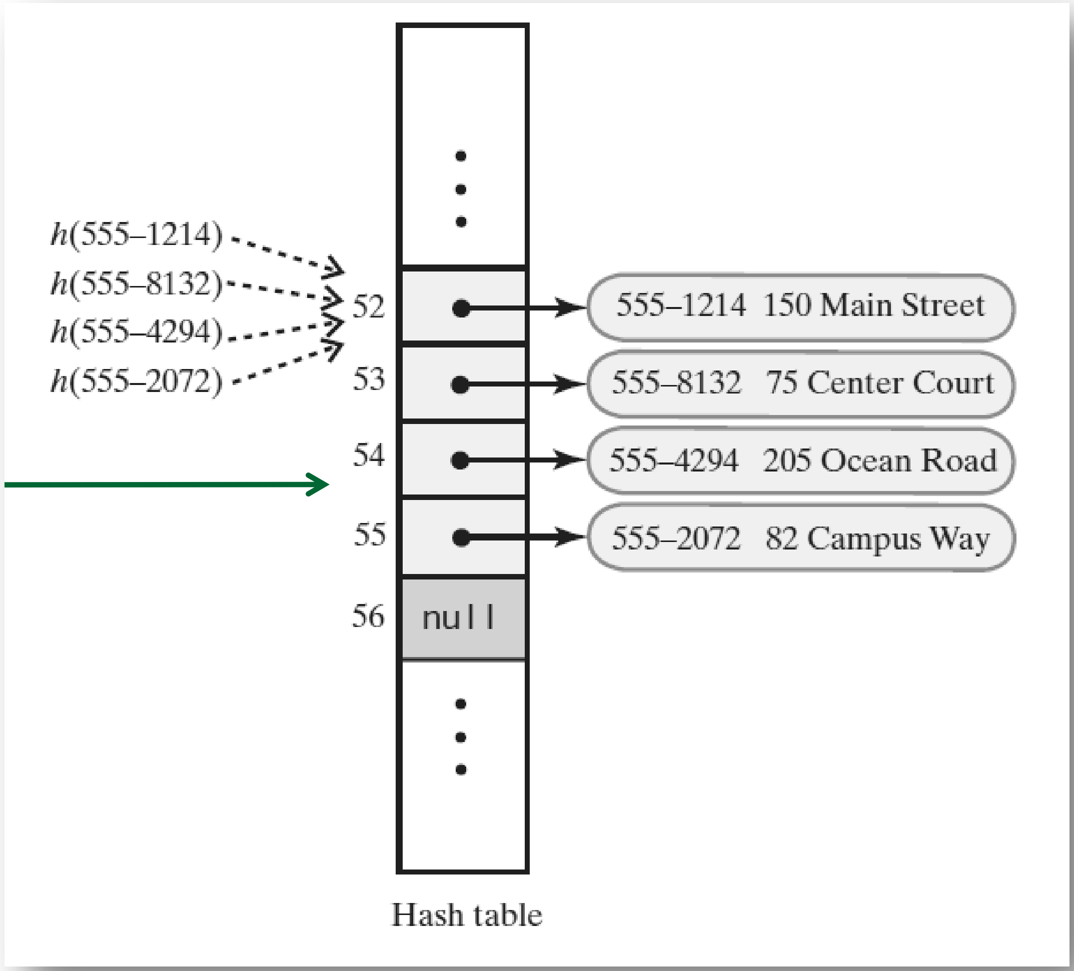
```
AddressBook. Add(555-1214", "150 Main Street");
AddressBook. Add(555-8132", "75 Center Court");
AddressBook. Add(555-4294", "205 Ocean Road");
AddressBook. Add(555-2072", "82 Campus Way");
```

The effect of linear probing after adding four entries whose search keys hash to the same index



Resolving Collisions

revision of the hash table shown in the previous figure when linear probing resolves collisions; each entry contains a search key and its associated value



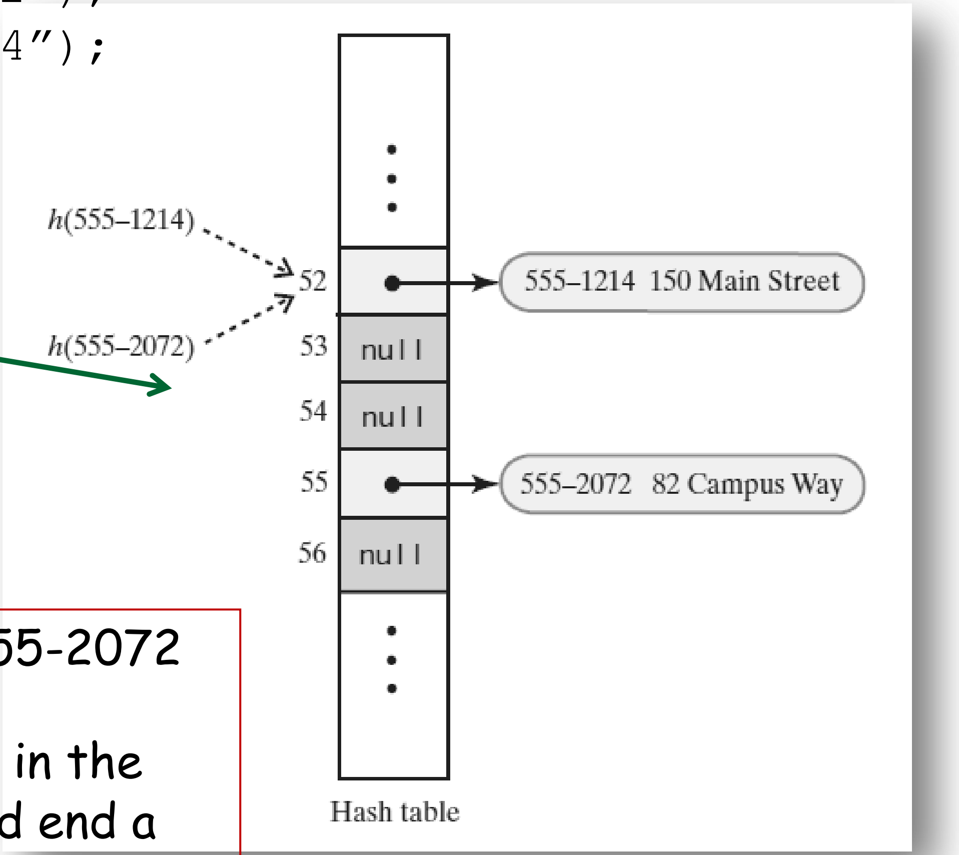
Resolving Collisions

Removals: we remove two entries by executing:

```
addressBook. remove("555-8132");
addressBook. remove("555-4294");
```

hash table if **remove** used **null** to remove entries.

What problem do you see?



An attempt to find the search key 555-2072 will terminates unsuccessfully at `hashTable[53]`. Although a location in the hash table that was never used should end a search, a location that had been used and is now available again for use should not.

Resolving Collisions

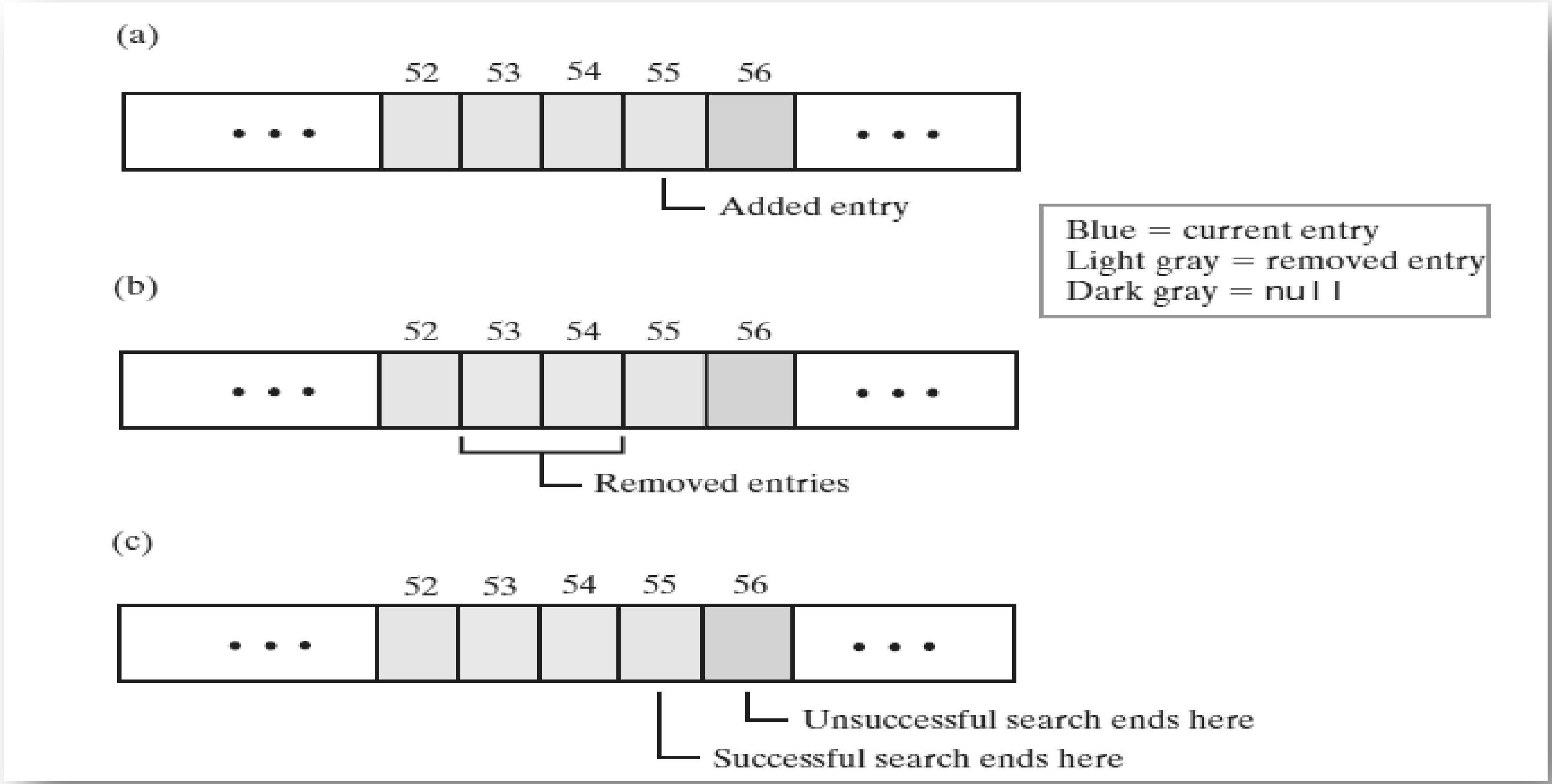
- Need to distinguish among three kinds of locations in the hash table
  - Occupied—the location references an entry in the dictionary
  - Empty—the location contains `null` and always has
  - Available—the location's entry was removed from the dictionary

- Accordingly, the method `remove` should not place `null` into the hash table, instead encode the location as available.
- The search during the retrieval should then continue if its encounters an available location and should stop only if it is successful or reach a `null` location.
- A search during a removal behaves in the same way.

Clustering

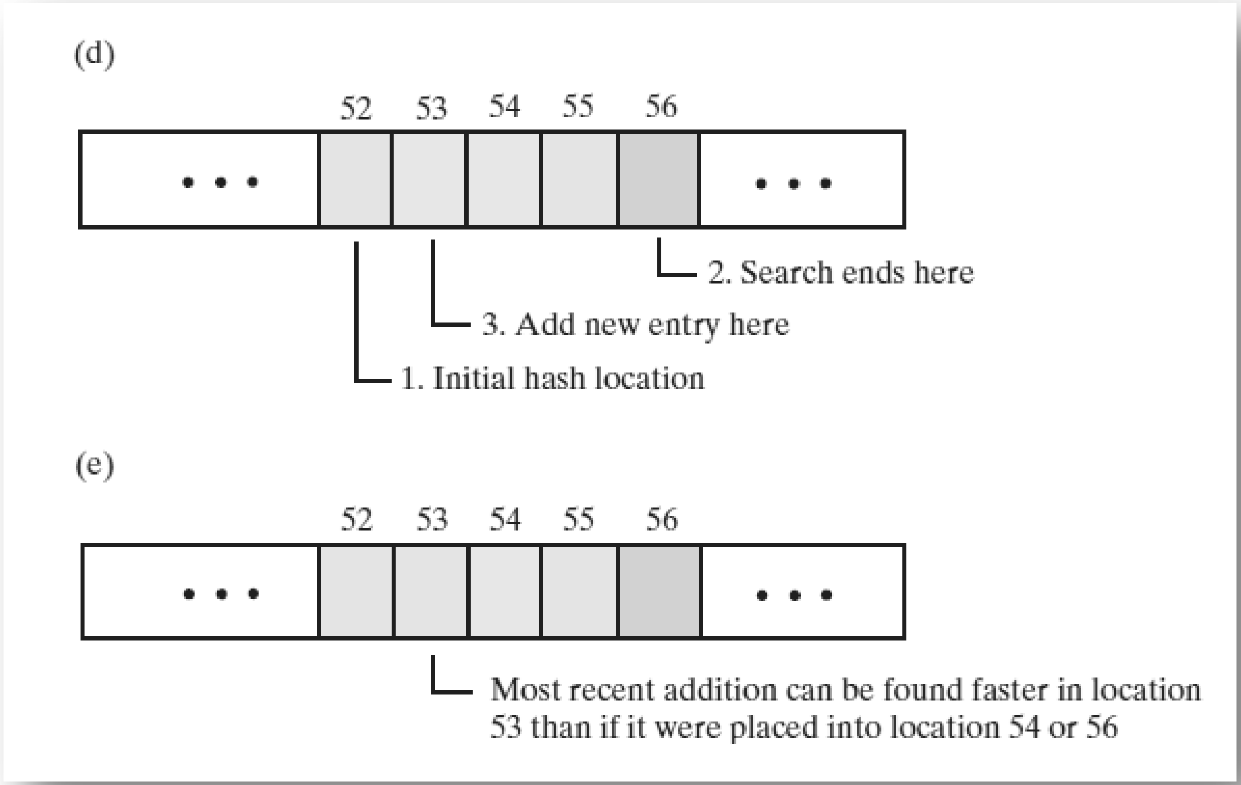
- Collisions resolved with linear probing cause groups of consecutive locations in hash table to be occupied
  - Each group is called a *cluster*
- Bigger clusters mean longer search times following collision

Clustering


(a)
52
53
54
55
56
Added entry
Blue = current entry
Light gray = removed entry
Dark gray = null
(b)
Removed entries
(c)
Unsuccessful search ends here
Successful search ends here

A linear probe sequence (a) after adding an entry; (b) after removing two entries;
(c) after a search;

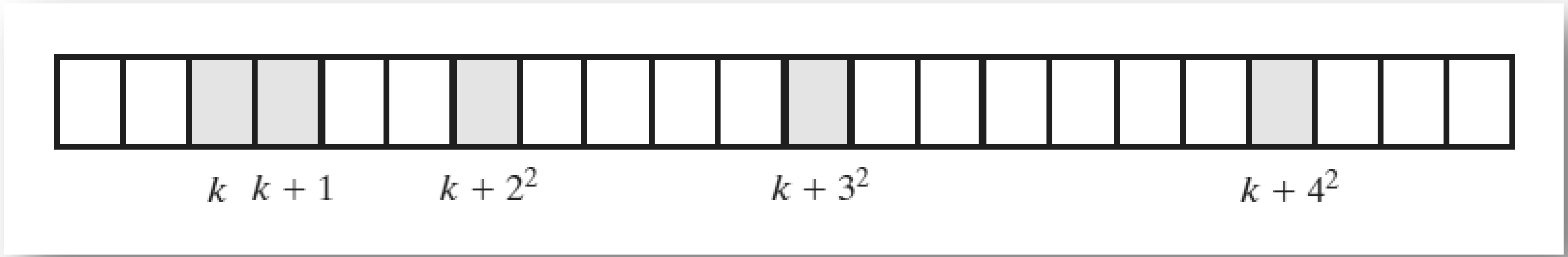
Clustering



A linear probe (d) during the search while adding an entry; (e) after an addition to a formerly occupied location

Open Addressing with Quadratic Probing

- Linear Probing looks at consecutive locations beginning at index *k*
- Quadratic Probing, considers the locations at indices k + j^2
  - Uses the indices *k*, *k* + 1, *k* + 4, *k* + 9, …



- L.P. :can reach every location in hash table: guarantees the success of `add` operation when the hash table is not full.
- Q.P.:can also guarantee a successful add operation as long as the hash table is at most half full and its size is prime number.
- Q.P. :requires more effort to compute the indices for the probe sequence than the L.P

Open Addressing with Double Hashing

- Linear probing and quadratic probing add increments to k to define a probe sequence
  - Both are independent of the search key
- Double hashing uses a second hash function to compute these increments
  - This is a key-dependent method.
    - This way, double hashing avoids both primary and secondary clustering.
    - Produce a probe sequence that reaches the entire table
    - The second hash function must be different from the original hash table and must never be zero value, as zero is not an appropriate increment.

Example: Open Addressing with Double Hashing

Consider the pair of hash functions for a hash table whose size 7.

$h_1(key)$= *key modulo 7*

$h_2(key)$= *5 - key modulo 5*

Let us study the behavior of probe sequence

For a search key of 16 we have:

$h_1(16)= 2$

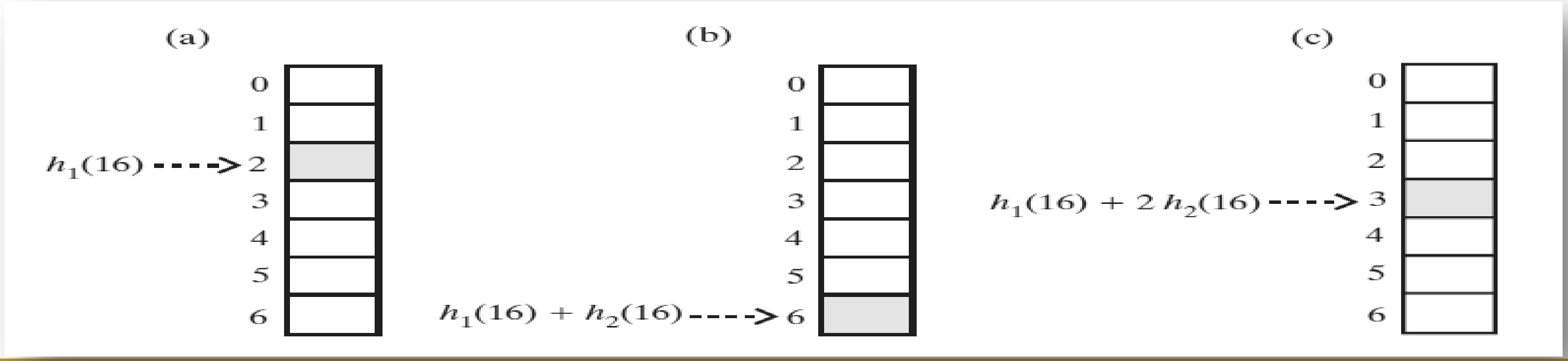
$h_2(16)= 4$

- *The probe sequence begins at 2 and probes locations at increments of 4.*
- *Remember when probing reaches the end of the table, it continues at the table's beginning.*

The table locations in the probe sequence will have the following indices:, 2, 6,3,0,4,1,5,2…

this sequence reaches all locations in the tables and repeats itself.

Notice that the table size is 7, is a prime number.



The first three locations in a probe sequence generated by double hashing for the search key 16

Recap: Double Hashing

- Resolves a collision during hashing by examining locations in the hash table at the original hash index plus an increment defined by second hash function.
  - The second hash function should:
    - Differ from the first hash function
    - Depend on the search key
    - Have a nonzero value
- Reaches every location in the hash table if the size of the table is a prime number
- Avoids both primary clustering and secondary clustering

Questions

Q1: What size hash table should we use with double hashing
when the hash functions are:

$$h_1(key) = key \text{ modulo } 13$$
$$h_2(key) = 7 - key \text{ modulo } 7$$

Why?

Q2: What probe sequence is define by hash functions given in the previous question when the search key is 16

Potential Problem with Open Addressing

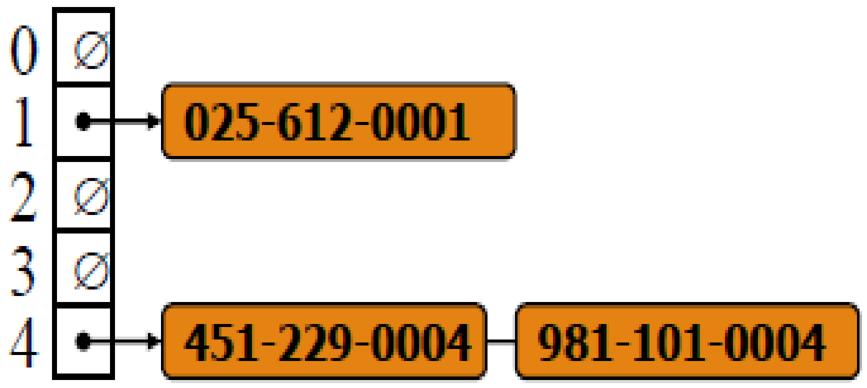
- Recall each location is either occupied, empty, or available
  - Frequent additions and removals can result in *no* locations that are null.
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Separate Chaining

- Alter the structure of the hash table
  - Each location can represent more than one value.
  - Such a location is called a *bucket*
- Decide how to represent a bucket
  - list, sorted list
  - array
  - linked nodes
  - vector

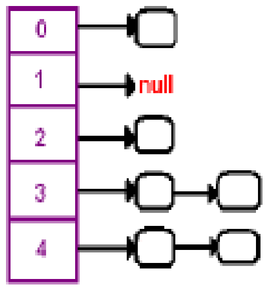
Collision Handling

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



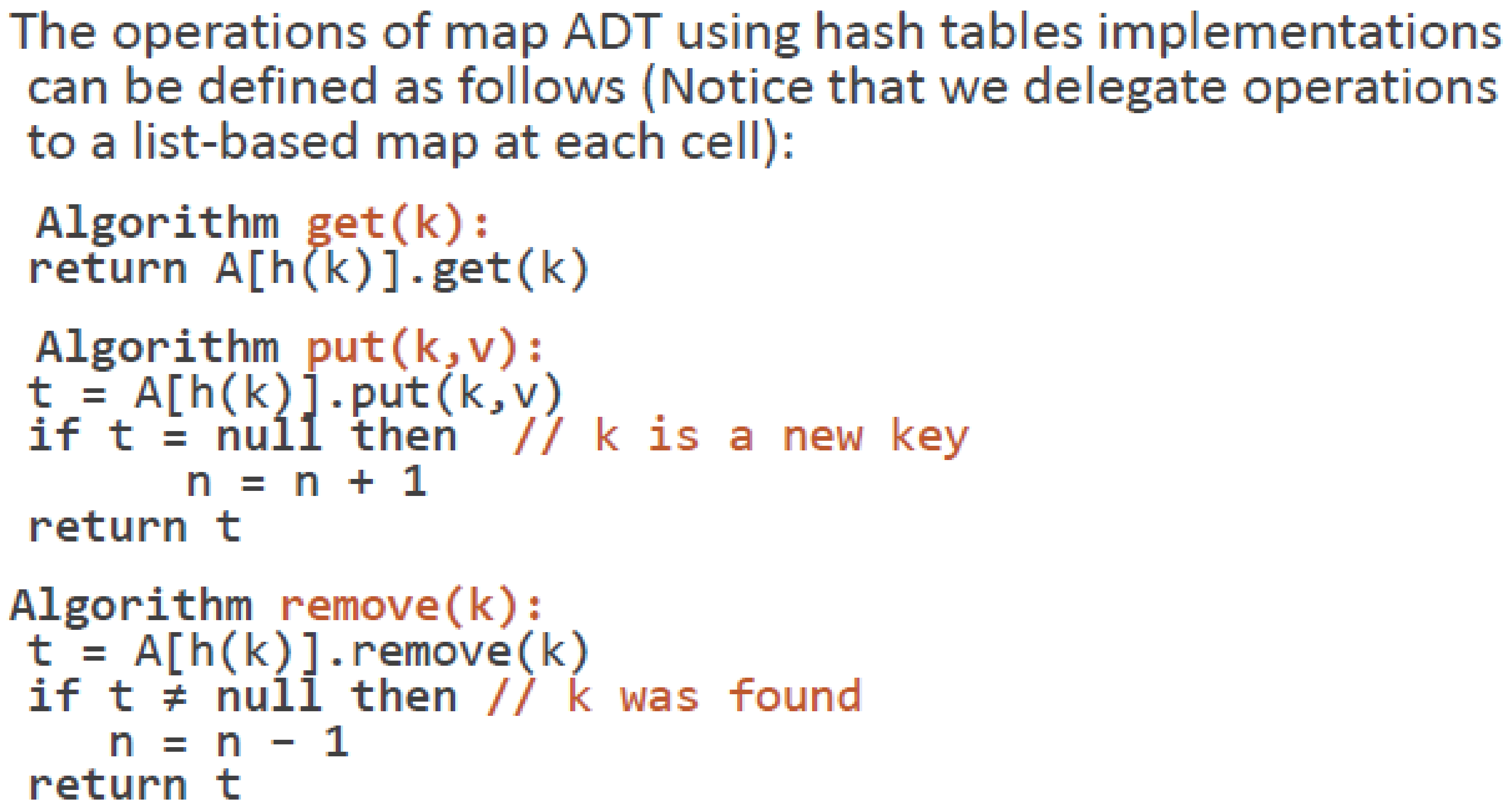
- Separate Chaining: let each cell in the table point to a map implemented as a linked list of entries.
- Separate chaining is simple, but requires additional memory outside the table.

Separate Chaining



- The array indices point to maps implemented using linked list. The operations of these individual maps are defined as usual:
  - 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, replace the value associated with k with v and return that old value
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null

Map with Separate Chaining



The operations of map ADT using hash tables implementations can be defined as follows (Notice that we delegate operations to a list-based map at each cell):

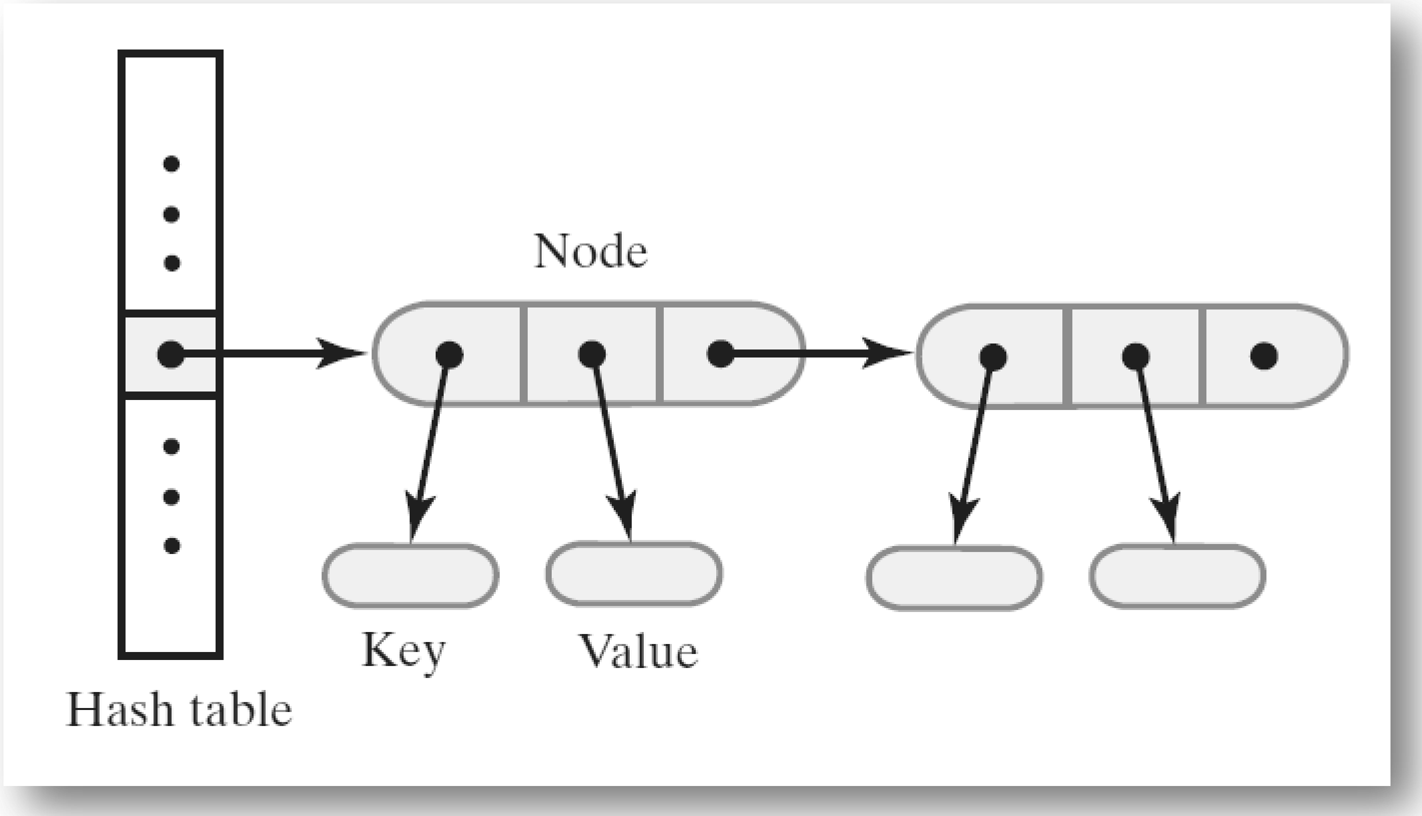
```
 Algorithm get(k):
return A[h(k)].get(k)

 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
```

Performance of Separate Chaining

- Assume n entries are inserted into the hash with Array of size N.
- If the hash function is good, then the entries are uniformly distributed; that is the size of each bucket is n/N.
- This value n/N is called the *load factor* of the hash table. This load factor should generally be bounded by some small constant, preferably below 1.
- Thus, with a good hash function, the expected time of get(), put() and remove() is $O[n/N]$.

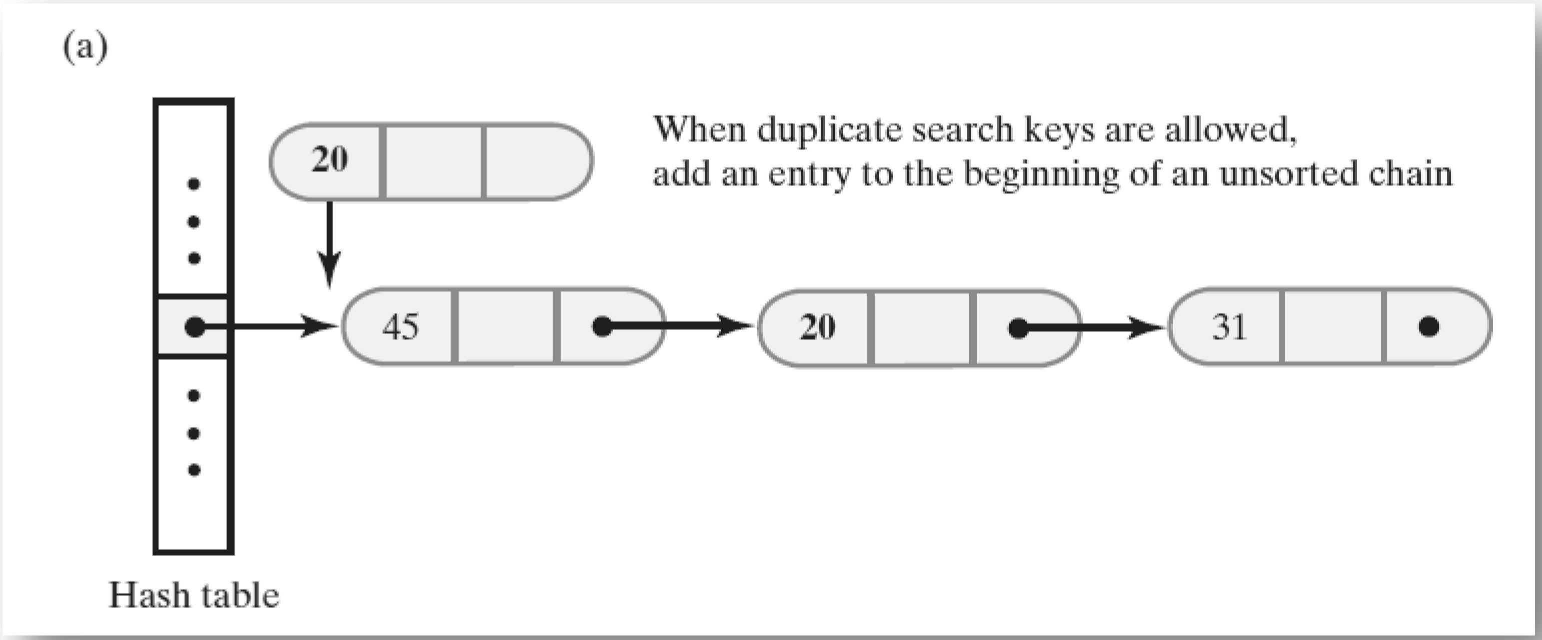
➔ **provided that n is $O(N)$, the operations can run in $O(1)$ time.**

Separate Chaining



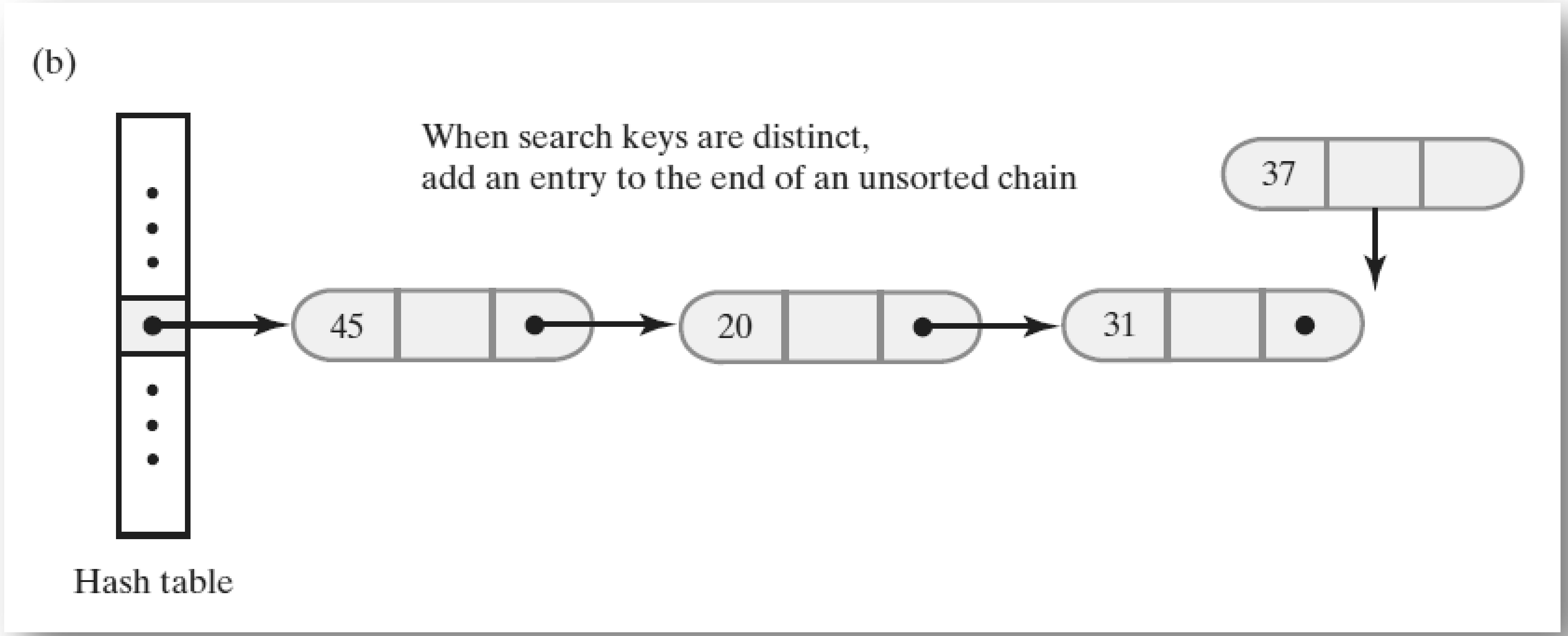
A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining


(a)
20
When duplicate search keys are allowed,
add an entry to the beginning of an unsorted chain
45
20
31
Hash table

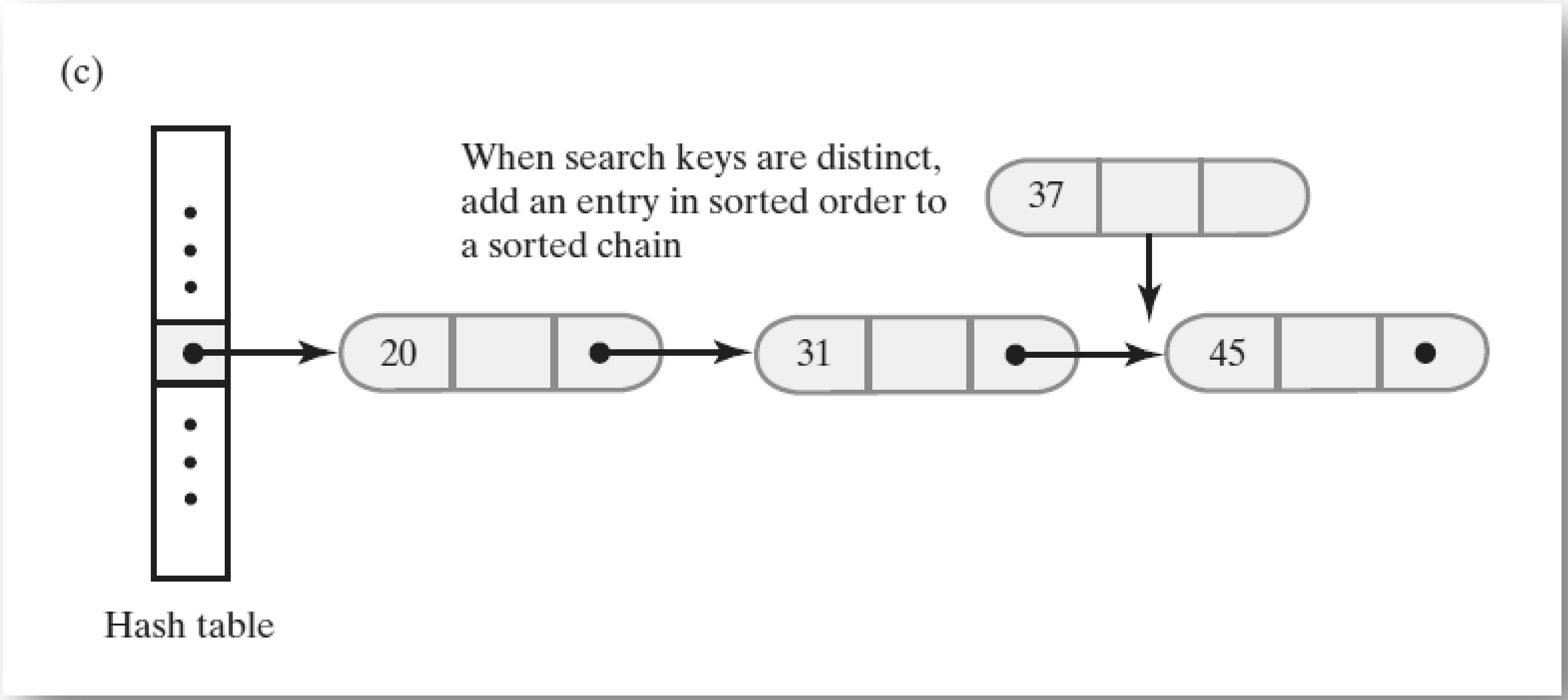
Where to insert a new entry into a linked bucket when the integer search keys are (a) unsorted and possibly duplicate;

Separate Chaining


(b)
When search keys are distinct,
add an entry to the end of an unsorted chain
37
45
20
31
Hash table

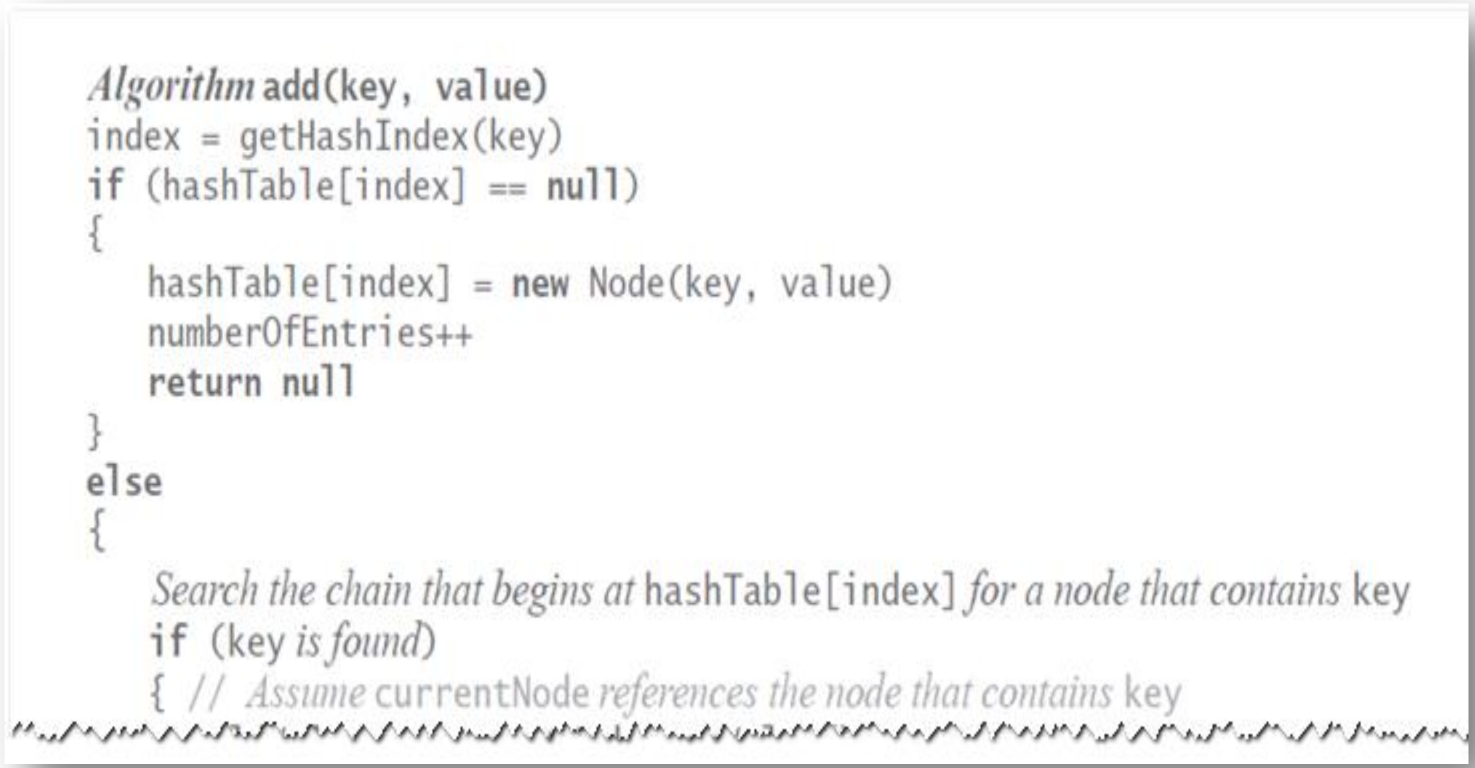
Where to insert a new entry into a linked bucket when the integer search keys are
(b) unsorted and distinct;

Separate Chaining


(c)
When search keys are distinct, add an entry in sorted order to a sorted chain
37
20
31
45
Hash table

Where to insert a new entry into a linked bucket when the integer search keys are
(c) sorted and distinct

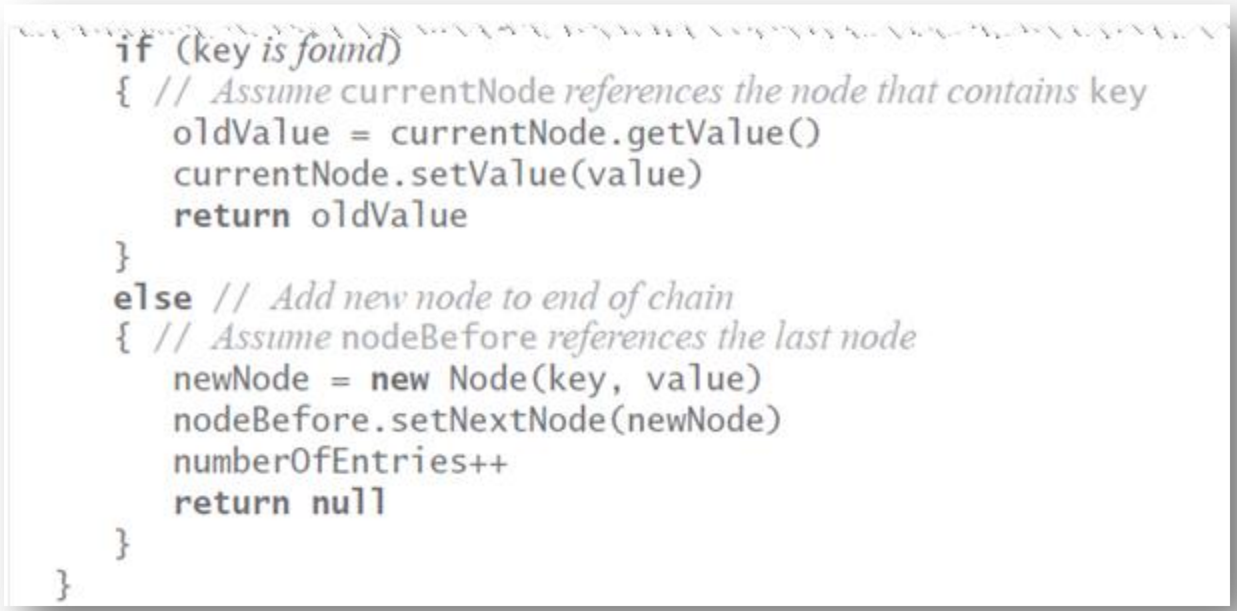
Separate Chaining



```
Algorithm add(key, value)
index = getHashIndex(key)
if (hashTable[index] == null)
{
    hashTable[index] = new Node(key, value)
    numberOfEntries++
    return null
}
else
{
    Search the chain that begins at hashTable[index] for a node that contains key
    if (key is found)
    { // Assume currentNode references the node that contains key
```

Algorithm for the Map's **add** method.

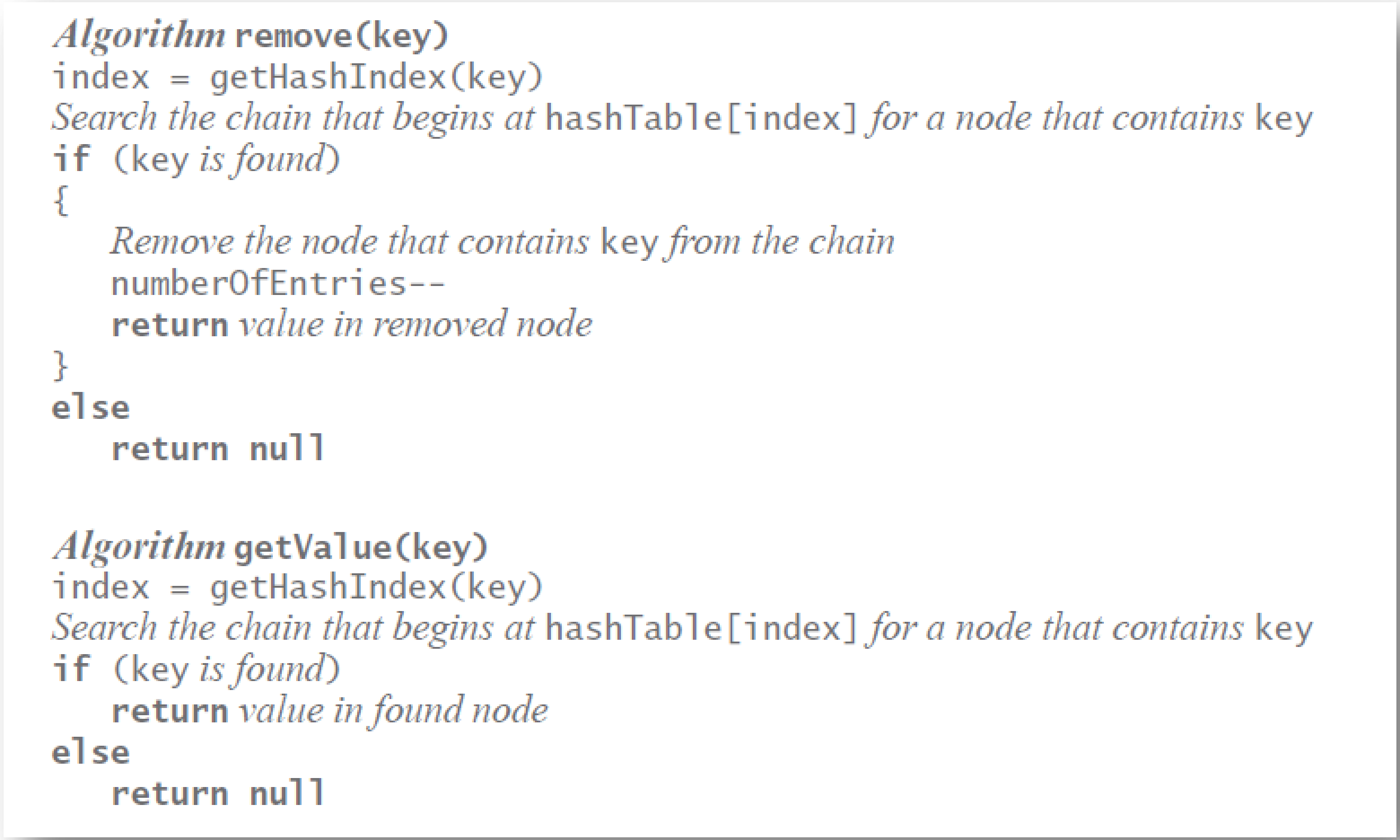
Separate Chaining



```
    if (key is found)
    { // Assume currentNode references the node that contains key
       oldValue = currentNode.getValue()
       currentNode.setValue(value)
       return oldValue
    }
    else // Add new node to end of chain
    { // Assume nodeBefore references the last node
       newNode = new Node(key, value)
       nodeBefore.setNextNode(newNode)
       numberOfEntries++
       return null
    }
 }
```

Algorithm for the Map's `add` method.

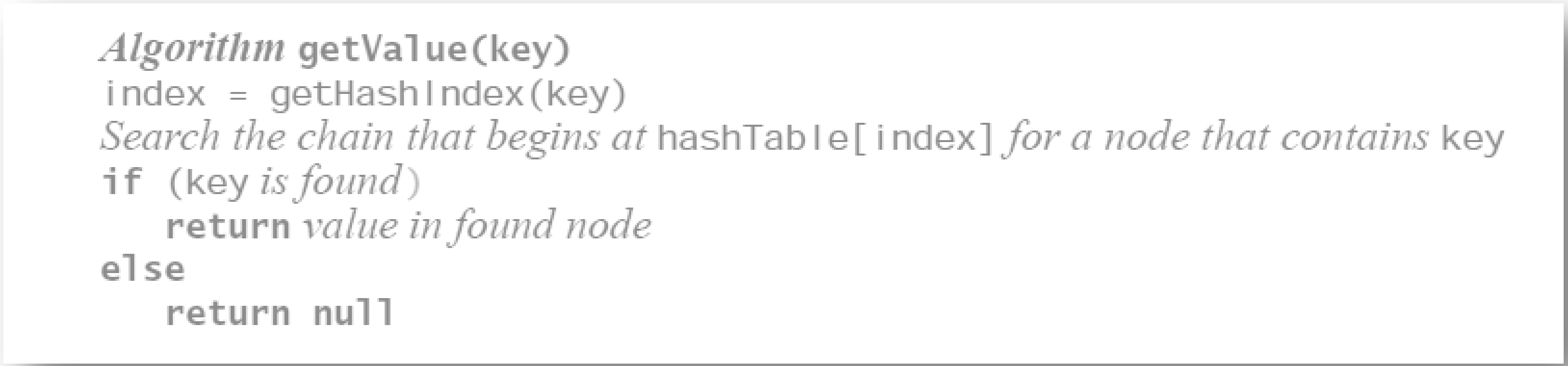
Separate Chaining



```
Algorithm remove(key)
index = getHashIndex(key)
Search the chain that begins at hashTable[index] for a node that contains key
if (key is found)
{
    Remove the node that contains key from the chain
    numberOfEntries--
    return value in removed node
}
else
    return null

Algorithm getValue(key)
index = getHashIndex(key)
Search the chain that begins at hashTable[index] for a node that contains key
if (key is found)
    return value in found node
else
    return null
```

Algorithm for the Map's `remove` method

Separate Chaining



```
Algorithm getValue(key)
index = getHashIndex(key)
Search the chain that begins at hashTable[index] for a node that contains key
if (key is found)
    return value in found node
else
    return null
```

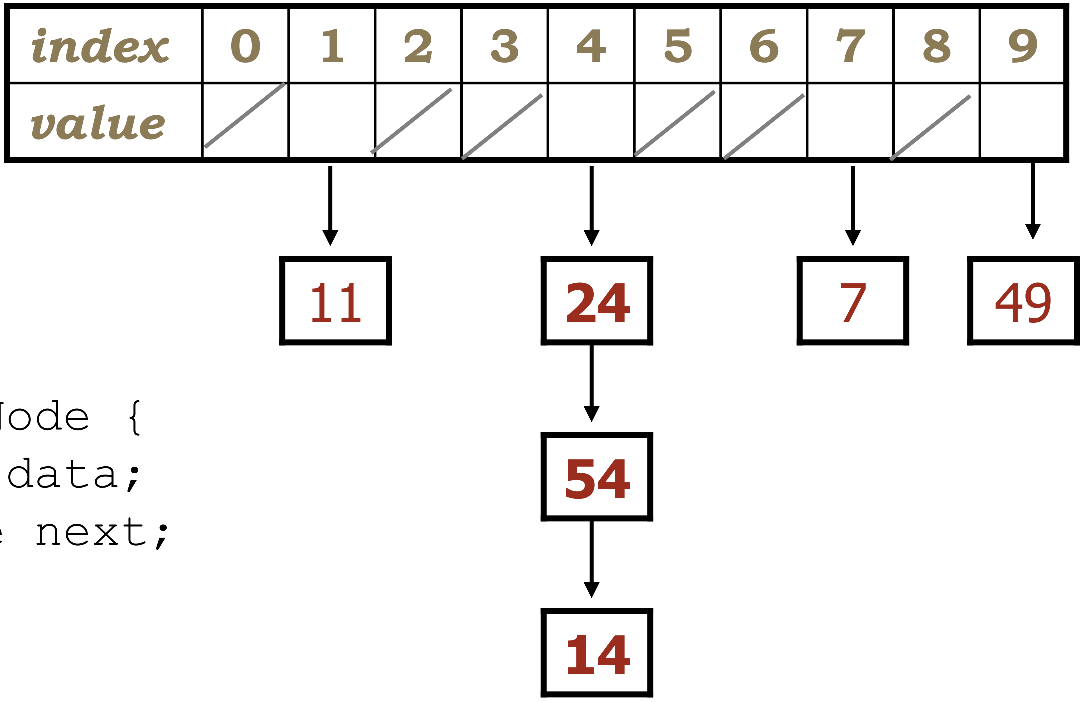
Algorithm for the Map's **getValue** method

Hashing

Separate Chaining

Solving collisions by storing a list at each index

- add/contains/remove must traverse lists, but the lists are short
- impossible to "run out“ of indexes, unlike with probing



```
private class Node {
    public int data;
    public Node next;
    ...}
```

Analysis of Open Addressing

- Open addressing schemes save space over separate chaining.
- However, they are not necessarily faster; in experimental and theoretical analysis , separate chaining is either competitive or faster depending on the load factor of the table.
- Additionally, the number of entries that can be inserted in the hash table using separate chaining is not restricted to the size of the array bucket.
- Consequently, if space is not a significant issue, separate chaining is the preferred collision-handling method.

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 $1/(1-\alpha)$
  - In other words, the smaller the load factor is, the smaller the number of probes will be. Larger load factors will result in larger probes.

Performance of Hashing

- The expected running time of all the dictionary ADT (to be discussed next) operations in a hash table is $O(1)$.
- In practice, hashing is very fast provided the load factor is not close to 100%.
- In specific, experiments and average-case analysis suggested that we should maintain the load factor, α, at a value < 0.5 for open addressing and < 0.9 for separate chaining.
  - For instance, Java uses a threshold of 0.75 as the maximum load factor, and rehashes if the load factor exceeds that.

Performance of Hashing

- If this value significantly exceeds such limits, then it might be better to resize the entire table.
- We refer to this as ***rehashing*** to a new table.
- In such case, it is advisable to double the size of the array so we can amortize the cost of rehashing all the entries against the time used to insert them.
- It is also advisable to redefine a new hash function to go with the new array.

➔ Even with periodic rehashing, hast tables are efficient means of implementing maps.

Hashing

Rehashing

- rehash: Growing to a larger array when table is too full
  - Cannot simply copy the old array to a new one (Why not?)

- load factor: ratio of (*# of elements*) / (*hash table length*)
  - many collections rehash when load factor ≅ .75

index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
value	95	11	0	0	24	54	14	37	66	48
size	8									

index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
value	0	0	0	0	24	0	66	0	48	0	0	11	0	0	54	95	14	37	0	0
size	8																			

Hashing

Rehash Method

```
// Grows hash table to twice its original size.
   private void rehash() {
      int[] old = elements;
      elements = new int[2 * old.length];
      size = 0;
      for (int value : old) {
         if (value != 0 && value != REMOVED) {
            add(value);
         }
      }
   }
   public void add(int value) {
      if ((double) size / elements.length >= 0.75) {
         rehash();
      }
      ...}
```

Hashing

HashIntSet Implementation

- Let's implement a hash set of ints using separate chaining

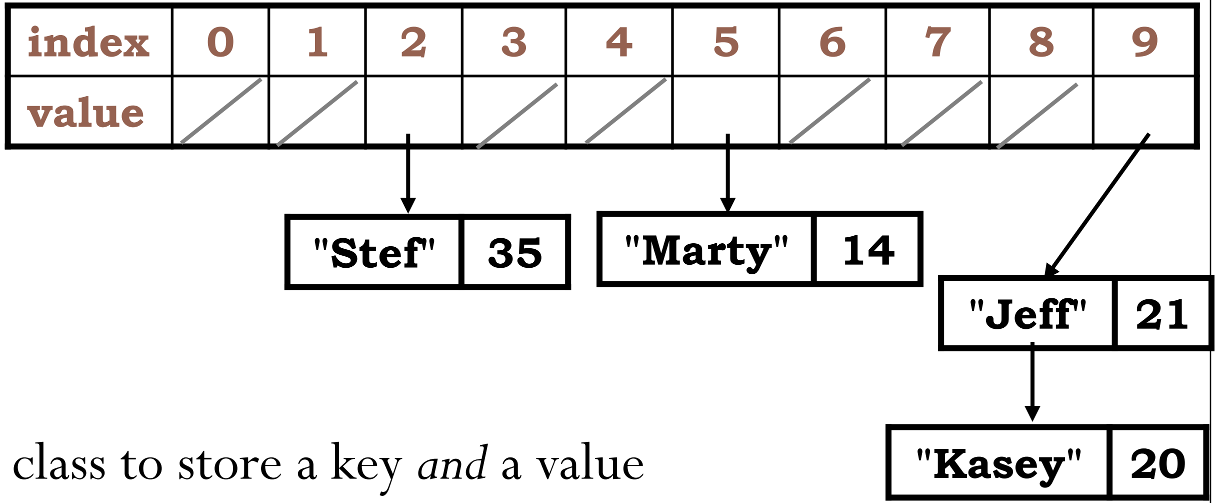
```
public class HashIntSet implements IntSet {
    // array of linked lists;
    // elements[i] = front of list #i (null if empty)
    private Node[] elements;
    private int size;

    // constructs new empty set
    public HashIntSet() {
        elements = new Node[10];
        size = 0;
    }

    // hash function maps values to indexes
    private int hash(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

Hashing

Hash Map

- A hash map is like a set where the nodes store key/value pairs:

```
public class HashMap<K,V> implements Map<K,V> {
    ...
}

//       key     value
map.put("Marty", 14);
map.put("Jeff",  21);
map.put("Kasey", 20);
map.put("Stef",  35);
```



  – Must modify your Node class to store a key *and* a value

Hashing

Hash Map

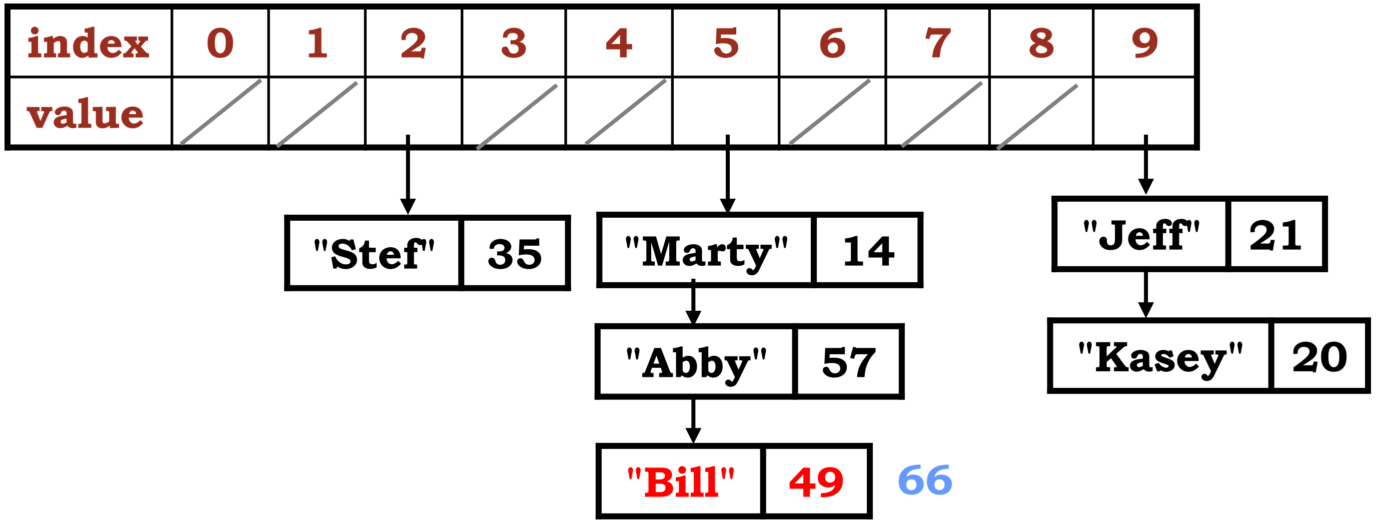
- Let's think how to write our implementation of a map
  - As is (usually) done in the Java Collection Framework, we will define map as an ADT by creating a Map interface.
  - Core operations: put (add), get, contains key, remove

```
public interface Map<K,V> {
    void clear();
    boolean containsKey(K key);
   V get(K key);
    boolean isEmpty();
    void put(K key, V value);
    void remove(int value);
    int size();
}
```

Hashing

Hash Map vs. Hash Set

- The hashing is always done on the keys, *not* the values
- The contains method is now containsKey; and in remove, you search for a node whose key matches a given key
- The add method is now put; if the given key is already there, you must replace its old value with the new one
  - map.put("Bill", 66);   // replace 49 with 66



ADT

- abstract data type (ADT): A specification of a collection of data and the operations that can be performed on it
  - Describes *what* a collection does, not *how* it does it.
- Java's collection framework describes ADTs with interfaces:
  - Collection, Deque, List, Map, Queue, Set, SortedMap
- An ADT can be implemented in multiple ways by classes:

– ArrayList and LinkedList	implement List
– HashSet and TreeSet	implement Set
– LinkedList , ArrayDeque, etc.	implement Queue

References

- Textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
- Some slides from ©University of Washington