Hashing and Hash Tables

Introduction

- ***hashing***: a technique used for storing and retrieving information as quickly as possible.

- used to perform optimal searches and is useful in implementing symbol tables.

- Why use hashing?
  - balanced binary trees (Red-Black Trees or AVL Trees) give O(logn) time for *insert*, *delete*, and *search*.
  - if you need these operations to be in O(1) time hashing gives this performance increase.
  - worst case for hashing is O(n), but on average it gives O(1).

Understanding Hashing: An Example

Hashing Example

- We can use an array has a *hash table*.

- Example: Write an algorithm which prints the first repeated character in a String if it contains duplicate characters.

Brute Force Approach

- Take the string and check each character to see if that character repeats or not.
  - What would be the time complexity of this approach?
  - $O(n^2)$
  - General Algorithm:

```
for each character in the string:
   get the current character
   for each index i from 0 to length - 1:
      count how many times the current character appears
      if count > 1:
         print out the letter and end
```

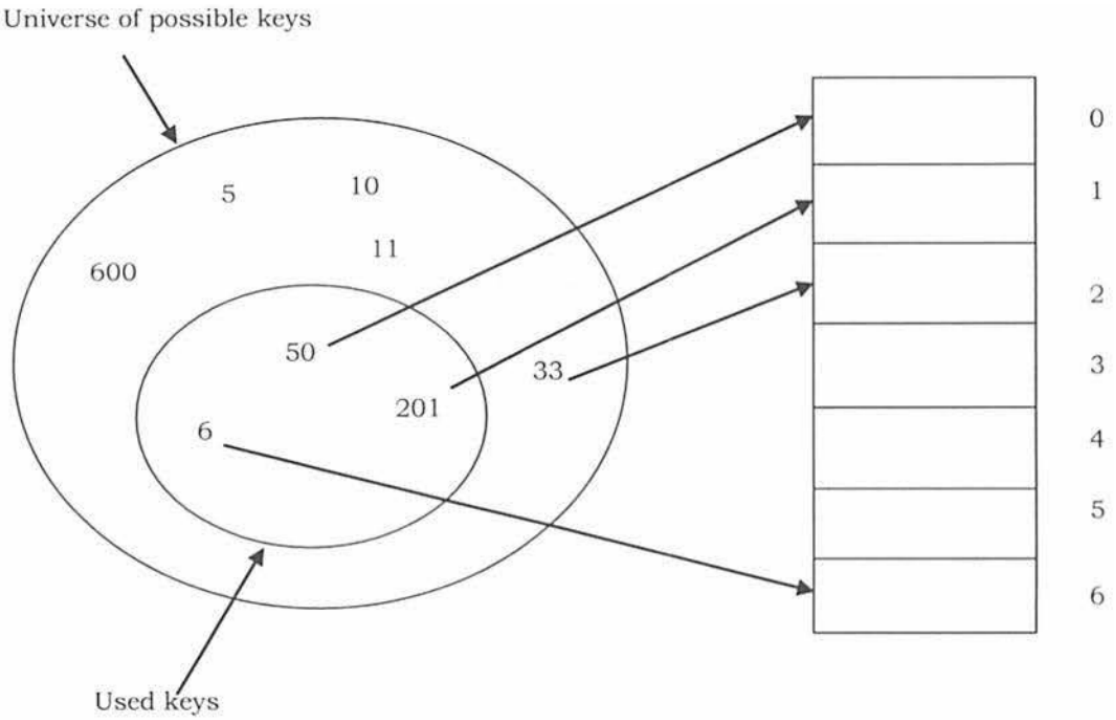
A Better Solution (Still Without Hashing)

- We can remember previous characters by using an array.

- For simplicity sake let's assume we are only using the ASCII characters.
  - We have 256 possible characters, we can create an array of size 256.
  - Initialize the array with all 0s.
  - For each character, go to its corresponding index position and increase the count.
    - The index corresponds to its ASCII value.
    - Example: If we find an @ symbol this has ASCII value 64 so increase the count at index 64 by 1.
  - Continue until you find a count greater than 1.

Why Not Arrays?

- In the previous implementation we used an array of exact size for our problem.
  - What if we had used numbers instead of characters?
    - Infinite possibilities for the set of values.
    - What if the size of our array has to be limited?



- We need to find a way to map the keys to one of the available locations in our array.
  - *hashing*: the process of mapping keys to locations

Components of Hashing

- There are four key components of any hashing algorithm:

  1) Hash Table

  2) Hash Function

  3) Collisions

  4) Collision Resolution Techniques

Hash Table

Hash Table

- Recall:
  - An array stores an element whose key is *k* at index *k.*
  - We find the element with key *k* at index *k.*
  - This technique is called ***direct addressing***.

- Direct addressing only works when you have the resources to allocate an array with one position for every possible key.

- ***hash tables*** or ***hash maps***: are data structures that store the keys and their values by using a hash function to map keys to their associated values.

- use a hash table when the number of keys stored is relative to the number of possible keys.

Hash Functions

Hash Function

- used to transform the key into the correct index of the array.
  - in theory, the hash function should map each possible key to a unique index.
  - in practice, this is difficult to achieve

- Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a ***perfect hash function***.
  - If you know the elements will never change, finding a perfect hash function is easy.

- Given an arbitrary collection of elements, there is no easy way to construct a perfect hash function, but we do not need perfection to achieve a performance efficiency.

Hash Functions

- A way to always have a perfect has function is to increase the size of the hash table so that each possible value has a space.
  - Not always feasible.

- Example: We want to store social security numbers so we would need 1 billion positions in our array. If we only wanted to store the numbers for 25 students in a class, this is impractical and a complete waste of space.

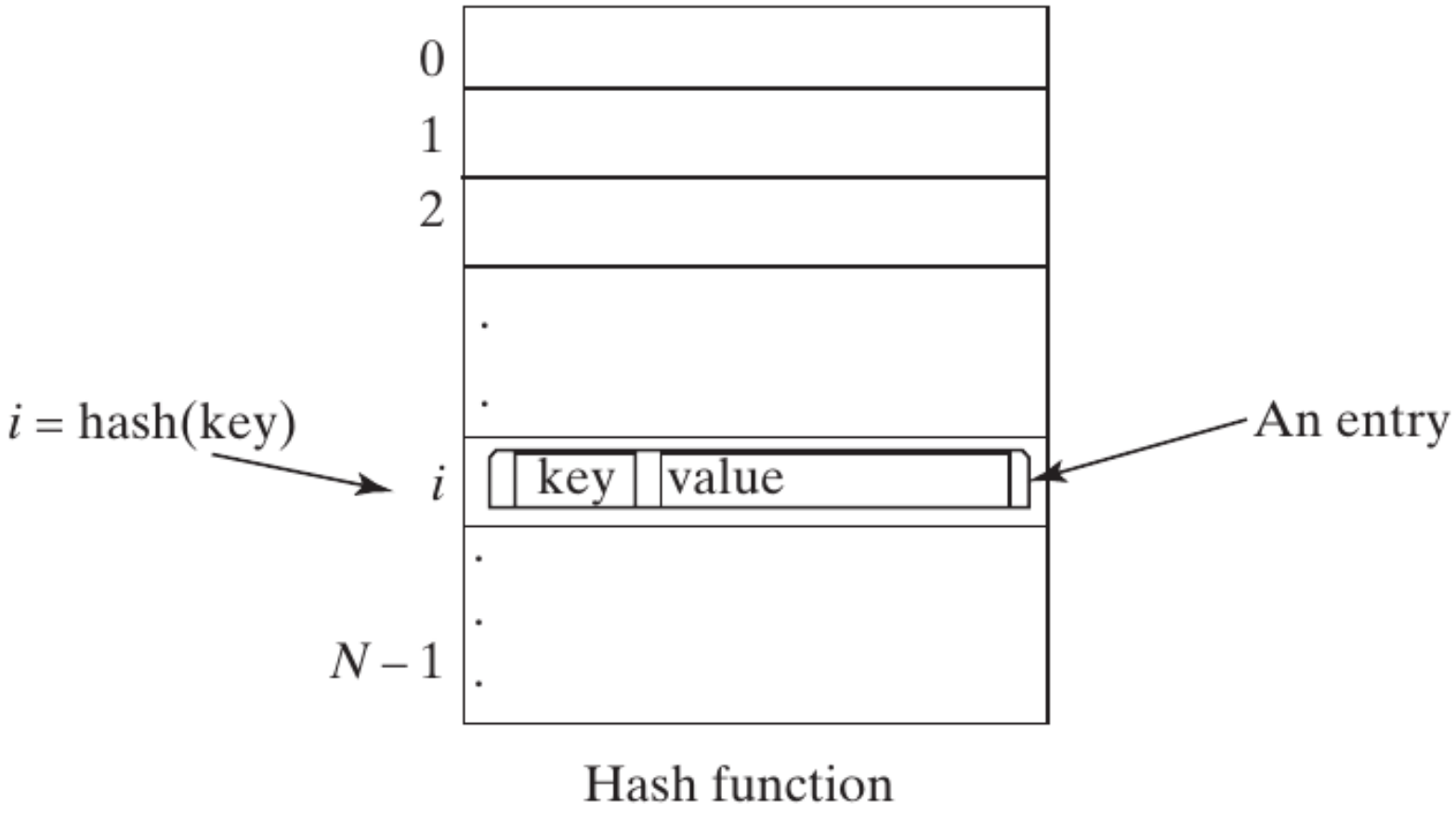


FIGURE 27.1 A hash function maps a key to an index in the hash table.

Side Note: Hash Functions in Java

- `Object` has the **`hashCode()`** method, which returns an integer hash code (by default the method returns the memory address for the object.)

- Rules for **`hashCode()`**
  - override the **`hashCode()`** method whenever the equals method is overridden to ensure that two equal objects return the same hash code.
  - invoking the **`hashCode()`** method multiple times on the same object should always return the same integer (as long as the object's data has not been changed).
  - Two unequal objects may have the same hash code, but you should implement the **`hashCode()`** method to avoid too many such cases.

Hashing Primitive Type Keys: Byte, Short, Int, Char, and Float

- for **`byte`**, **`short`**, **`int`**, and **`char`** just cast them to an **`int`**
  - two different search keys of any of these types will have different hash codes.

- for **`float`** use **`Float.floatToIntBits(key)`** as the hash code
  - returns an **`int`** value whose bit representation is the same as the bit representation for the floating number
  - two different search keys of the **`float`** type will have different hash codes.

Hashing Primitive Type Keys: Long

- for **`long`** types:
  - casting to an int will not work
    - possible for some keys to share the same hash value.
  - bitwise operations are useful here:
    - these are operations performed on the bits of a number, not the actual value of the number.
    - **>>** is the right-shift operator that shifts the bits of a value n positions to the right.
      - 1010110 >> 2 yields 0010101.
    - **^** is the bitwise exclusive-or operator. It operates on two corresponding bits of the binary operands.
      - 1010110 ^ 0110111 yields 1100001.
  - divide the 64 bits of a long into two halves and use the XOR (^) operation to combine the two halves (this is called ***folding***)

```
int hashCode = (int)(key ^ (key >> 32));
```

Hashing Primitive Types: Double

- For double types convert the double to a long and then perform the previous folding calculation.

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

Hashing Strings: Unicode Summation

- One approach is to sum the Unicode values of the characters as the hash code
  - only works if two search keys don't contain the same letters
  - eat and tea would cause a collision

Hashing Strings: Polynomial Formula

- a better method is to take the position of the characters into consideration:

$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \ldots + s_{n-1}$$

  - s_i is s.charAt(i)
  - n is the number of characters in the string.
  - this is a polynomial hash code.
  - This computation can cause an overflow for long strings, but arithmetic overflow is ignored in Java.
    - If the value ends up being negative, take the absolute value.
  - choose an appropriate value b to minimize collisions.
  - Experiments show that good choices for b are 31, 33, 37, 39, and 41.
  - In the String class, the hashCode is over-ridden using the polynomial hash code with b being 31.

Hash Code Compression

- it's possible for a hash code to be out of range of the possible indexes of the hash-table

- have to scale down the code to fit a particular index range
  - assume the index has to be between 0 and N – 1
  - you can use **`hashCode % N`**

- To ensure that the indexes are spread evenly, choose N (the size of your array) to be a prime number greater than 2.
  - This also helps to avoid clustering.

Hash Code Compression

- `java.util.HashMap` makes N a power of 2
  - When N is a power of 2:
    - we can use hashCode % N
    - or we can use hashCode & (N – 1)
  - the ampersand here (&) is the bitwise AND operator, performs an AND operation on the bits of a value.

- Example: assume N = 4 and hashCode = 11
  - 11 % 4 = 3 which is the same as 01011 & 00011 = 3.
  - NOTE: The & operation is performed faster than the % operation.

Designing a Hash Function

- How to design a hash function?
  - the function should be able to map a key to a different index in the hash table (***perfect hash function***)
  - perfect hashing is difficult
  - two or more keys mapped to the same hash value is called ***collision***
  - therefore, design a fast and easy-to-compute hash function that minimizes collisions.

Collisions and Collision Resolution

Collisions

- ***collision***: When a hash function assigns the same hash code to two or more items even if they have different values.

- Two records will have to be stored at the same location.

- ***collision resolution***: The process of finding an alternate location when a collision occurs.
  - Two popular techniques:
    - Separate Chaining
    - Open Addressing

Collision Resolution: Separate Chaining

- separate chaining scheme puts all entries with the same hash index in the same location

- Each location uses a bucket to hold multiple entries.
  - buckets can be implemented using an array, ArrayList, or LinkedList
  - LinkedList is used in the example, so each cell of the hash table is a reference to the head of the LinkedList

Collision Resolution: Separate Chaining

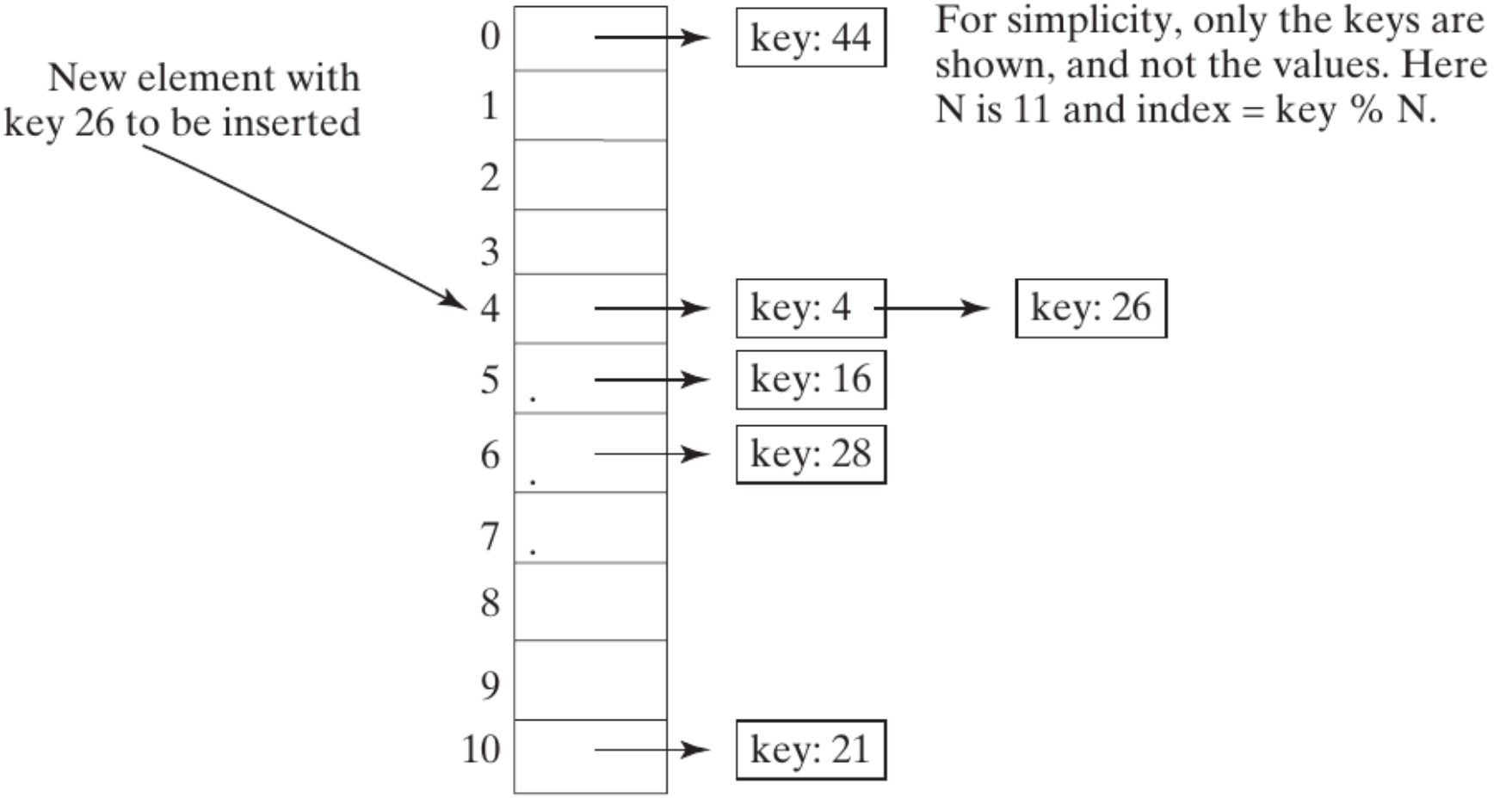


FIGURE 27.7 Separate chaining scheme chains the entries with the same hash index in a bucket.

Collision Resolution: Open Addressing

- ***open addressing***: the process of finding an open location in the hash table when a collision occurs

- there several variations:
  - linear probing
  - quadratic probing
  - double hashing.

Notation

- Notation used on the following slides:
  - *k* is the key that you want to insert
  - *h* is the hashed key value
    - *h = hash(k)*
  - *j* is the jump size (how far do we jump ahead to look for an empty spot in the table).
  - N is the capacity of the hash table.

Linear Probing

- when there is a collision, finds the next available location sequentially.

- basically we keep searching through the array one by one starting from the initial hash value position until we find an empty location.

- with linear probing, the value of *j* starts at 1 and increases by 1 for all j >= 0 will collisions are found.

Linear Probing Algorithm

- The following is an algorithm for insertion which uses linear probing:
  - Compute the hashcode of the key. The result of hash mod N is the current index of the table.
    - h = hash(k)
    - index = h % N
    - j = 1
  - If the position at index is empty, place the item at that position, finish
  - else repeat until no collision:
    - index = (h + j) % N
    - j++
    - if no collision found at new index, place item at index and done.

Linear Probing

Note

When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

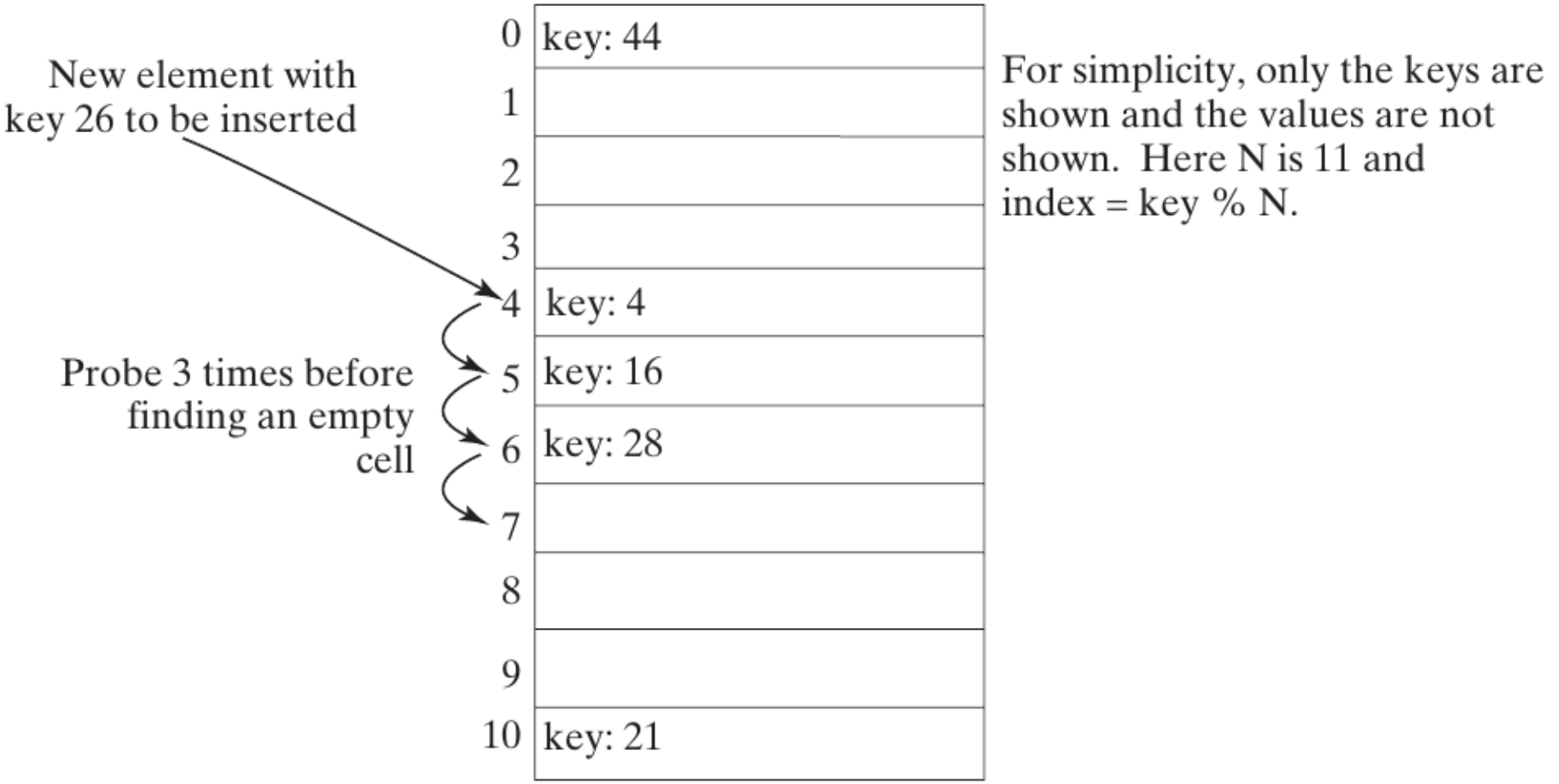


FIGURE 27.2 Linear probing finds the next available location sequentially.

Linear Probing

- Searching for a key in the hash table:
  - compute the index = hash(key) % N
  - Check if **`hashTable[index]`** contains the entry you are looking for. If so, return the item. If not, probe until item is found or you reach an empty space.
    - You may also need to check whether the table was filled and you have searched all positions, (this can be avoided however...)

- Removing an entry from the hash table:
  - search the entry that matches the key according to the above algorithm.
  - If the entry is found set the position to be null (empty).

Linear Probing

- Linear probing can cause groups of consecutive cells in the hash table to be occupied.
  - Each group is called a ***cluster***.
  - Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry.
  - As clusters grow in size, they may merge into even larger clusters, further slowing down the search time.
  - This is a big disadvantage of linear probing.

Quadratic Probing

- Quadratic probing can help avoid clustering
- Linear probing looks at consecutive cells
- Quadratic probing looks at the cells that can be jumped to by squaring j each time:
  - $(h + j^2) \% N$, for $j >= 0$
  - i.e. it looks at $h \% N$, $(h + 1) \% N$, $(h + 4) \% N$, $(h + 9) \% N$, and so on

Quadratic Probing – Insertion Algorithm

- The following is an algorithm for insertion which uses quadratic probing:
  - Compute the hashcode of the key. The result of hash mod N is the current index of the table.
    - h = hash(k)
    - index = h % N
    - j = 1
  - If the position at index is empty, place the item at that position, finish
  - else repeat until no collision:
    - index = (h + j^2) % N
    - j++
    - if no collision found at new index, place item at index and done.

Quadratic Probing

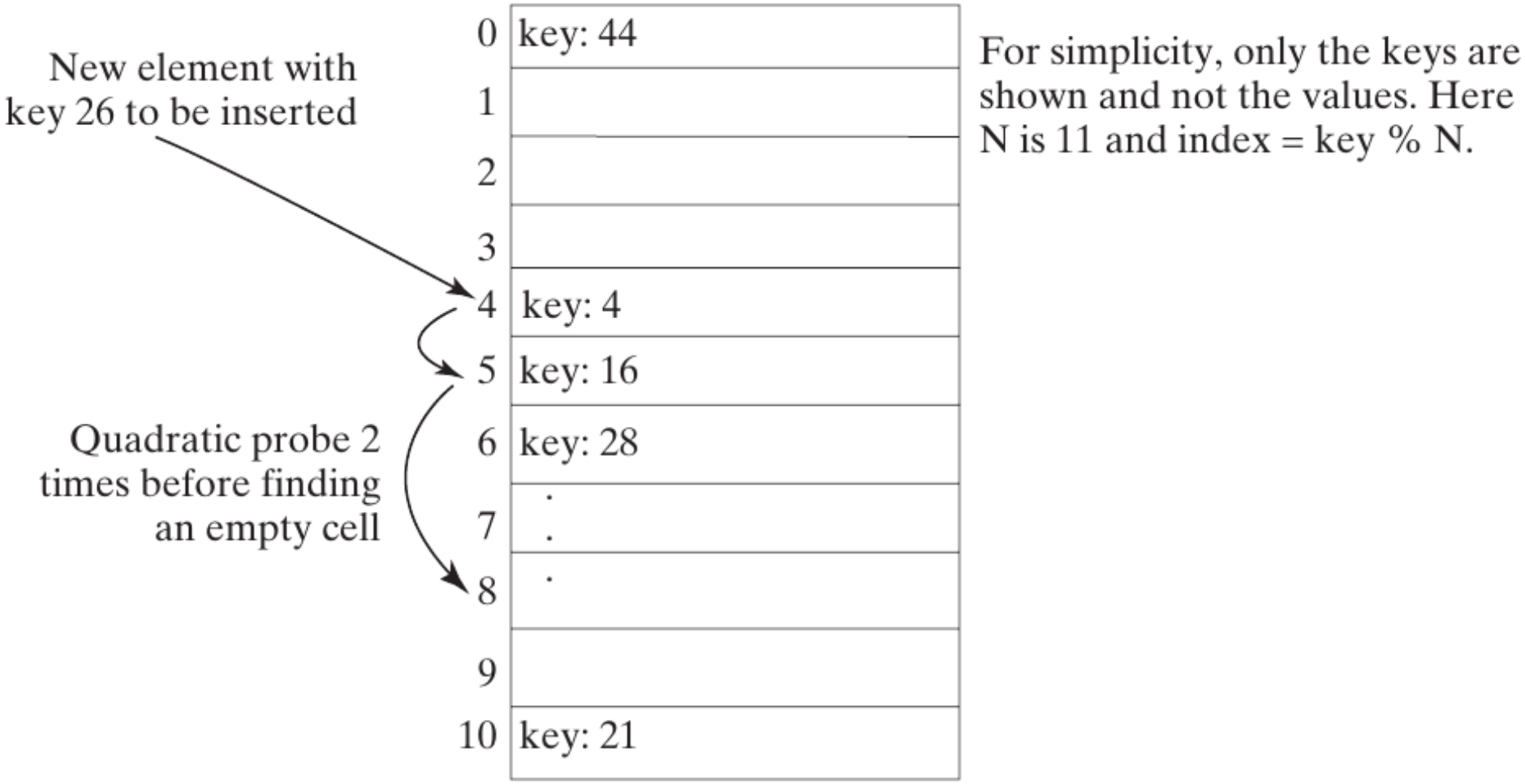


FIGURE 27.4 Quadratic probing increases the next index in the sequence by j^2 for $j = 1, 2, 3, \ldots$.

Quadratic Probing

- works in the same way as linear probing except for a change in the search sequence.

- avoid the linear probing clustering, but has its own clustering problem (called ***secondary clustering***)
  - the entries collide when they use the same probe sequence

- Linear probing guarantees that an available cell can be found for insertion as long as the table is not full.

- However, there is no such guarantee for quadratic probing.

Quadratic Probing - Example

- Consider the following:
  - Table size is 16, and the first 5 items all hash to index 2
  - First item goes to index 2.
  - Second item goes to 3 ((2 + 1)%16
  - Third item goes to 6 ((2+4)%16
  - Fourth item goes to 11 ((2+9)%16
  - Fifth item does not get inserted
    - (2+16)%16 = 2 is full
    - we end up back at the beginning
    - not all empty spaces were searched.

- To guarantee quadratic probing will reach every available spot your table size must:
  - be a prime number.
  - never be more than half full (even by one element)

Double Hashing

- also avoid clustering

- uses a secondary hash function h'(key) to determine the increments to avoid clustering.

- Double Hashing looks at the cells that can be jumped to by multiplying j by the result of the secondary hash function:
  - (h + j* hash′(k)) % N, for j >= 0,
  - i.e. it looks at h % N, (h + hash′(k))% N, (h + 2* hash′(k)) % N, (h + 3* hash′(k)) % N, and so on

Double Hashing

- The following is an algorithm for insertion which uses double hashing:
  - Compute the hashcode of the key. The result of hash mod N is the current index of the table.
    - h = hash(k)
    - index = h % N
    - j = 1
  - If the position at index is empty, place the item at that position, finish
  - else repeat until no collision:
    - index = (h + j * hash'(k)) % N
    - j++
    - if no collision found at new index, place item at index and done.
- NOTE: if k is a string, then hash'() function can take the result of hash(string) as input.

For example, let the primary hash function h and secondary hash function h' on a hash table of size 11 be defined as follows:

```
h(key) = key % 11;
h'(key) = 7 - key % 7;
```

For a search key of 12, we have

```
h(12) = 12 % 11 = 1;
h'(12) = 7 - 12 % 7 = 2;
```

- Suppose the elements with the keys 45, 58, 4, 28, and 21 are already placed in the hash table.
- to insert the element with key 12:
  - The probe sequence for key 12 starts at index 1
  - Since index 1 is already occupied, search the next cell at index 3 (1 + 1 * 2).
  - Since index 3 is already occupied, search the next cell at index 5 (1 + 2 * 2).
  - Since the index 5 is empty, the element for key 12 is now inserted at this cell.
- NOTE: The indexes of the probe sequence are as follows: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10. (it reaches all of the indexes of the table).
- The probe sequence should always produce every possible index of the table and the second function should never have a zero value, since zero is not an increment.

Double Hashing

- If your keys are integers, a secondary hash function you could use is:

  hash'(k) = 1 + ((k / N) mod (N – 1))

- Another popular secondary hash function is:
  - hash'(k) = P - (k % P), where P is a prime number less than the size of the table (N)

- NOTE: For either of these secondary hash functions, you still need to mod by the capacity of the Hash Table.

- NOTE: Integer keys are always easiest to work with when you want to use open addressing. You can always design your object to implement an integer key when you want to store your objects in a hash table.

- If you are using double hashing with Strings, you can hash the String using the hash function that Java uses (see section on hash functions) and then take that result and pass it to one of the two functions above.

Double Hashing

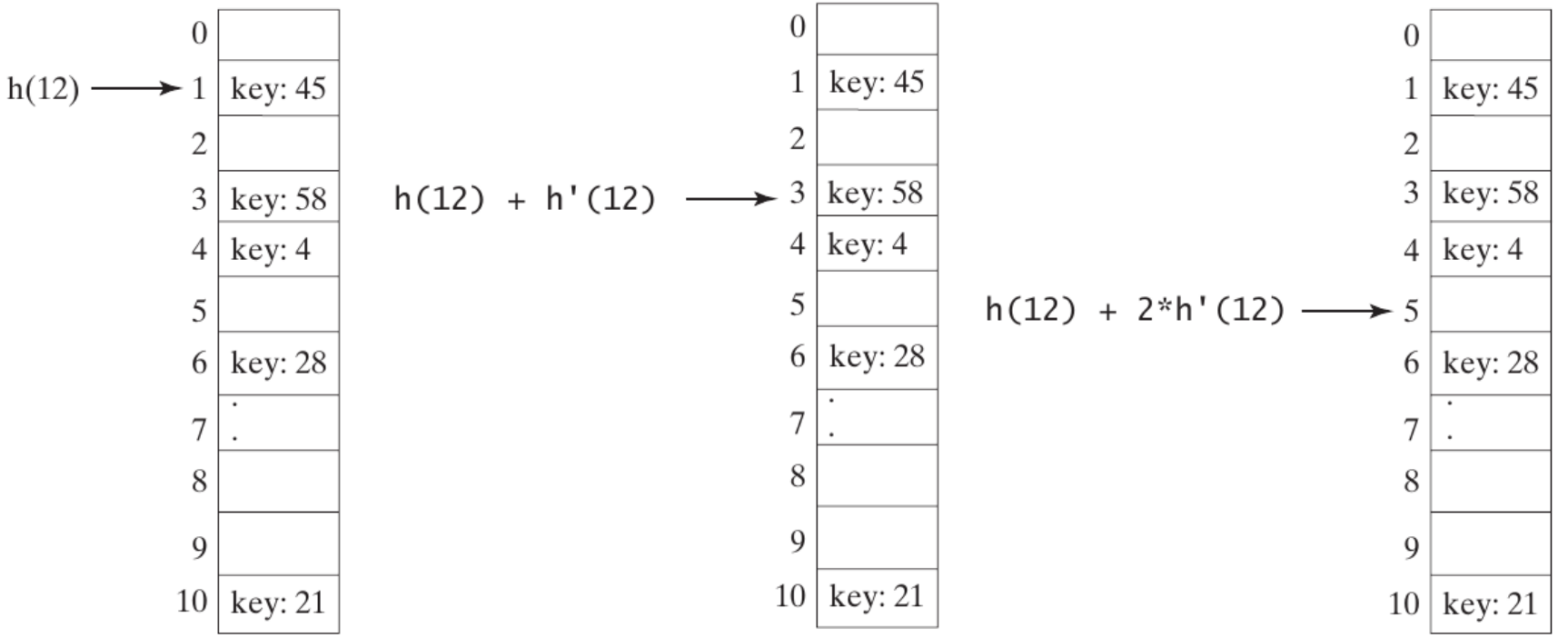


FIGURE 27.6 The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.

Load Factor and Rehashing

Load Factor and Rehashing

- ***load factor*** measures how full a hash table is.
  - If the load factor is exceeded, increase the hash-table size and reload the entries into a new larger hash table (called ***rehashing***).
- assume load factor (LF) is the ratio of the number of elements to the size of the hash table i.e. LF = n / N where n is the number of elements and N is the number of locations in the hash table.
- LF is 0 if the hash table is empty
- For open addressing LF is between 0 and 1 (1 if the hash table is full)
- for separate chaining LF can be any value

Load Factor and Rehashing

- As LF increases the probability of a collision increases
  - think about it...as the table becomes more and more filled that's less places to put hash codes, so of course a collision will happen.
- Research shows you should maintain a load factor under 0.5 for open addressing, and 0.9 for separate chaining.
  - 0.75 for Linear Probing
  - 0.5 for Quadratic Probing
- HashMap uses a load factor of 0.75 and separate chaining.
- if the load factor exceeds the threshold increase the hashtable size and rehash all the entries into a new larger hash table.
  - this means you will have to change the hash functions since they are based on the size of the hash table
  - To reduce the likelihood of rehashing (rehashing is very costly) at least double the hash-table size and find the nearest prime number greater than the doubled size.
- Despite having to rehash periodically, hashing is still efficient for implementing a map.

Sets

Sets

- ***set***: an efficient data structure for storing and processing non-duplicate elements.
  - any list type data structure can be converted into a set
  - simply have your data structure enforce that there can be no duplicate items.

- Sets in Java can be created using three of the concrete Set classes:
  - **`HashSet`**
  - **`LinkedHashSet`**
  - **`TreeSet`**

HashSet

- Has no guaranteed ordering to the elements.

- Constructors:
  - no-arg: empty HashSet
  - HashSet from an existing collection
  - default initial capacity: 16
  - default load factor: 0.75
    - ***load factor*** is a value between 0.0 and 1.0 and measures how full a set is allowed to be before its capacity is increased
    - capacity is usually doubled when expansion is needed
    - high load factors decrease the space costs but increase the search time, generally 0.75 is a good tradeoff between time and space costs.

HashSet

- To make a **`HashSet`** efficient, objects added to a **`HashSet`** need to implement the **`hashCode()`** method in a way which properly disperses the hash code.

- The result of **`hashCode()`** is used to place objects into the set.
  - HashSets do not have any kind of ordering, but have very fast access times.

LinkedHashSet

- extends **`HashSet`** with a linked-list implementation that supports an ordering of the elements in the Set.
  - generally this is insertion order

TreeSet

- **`SortedSet`** is a subinterface of **`Set`**
  - guarantees the elements in the **`Set`** are sorted
  - provides methods **`first()`** and **`last()`** to get the first and last elements in the set
  - provides **`headSet(toElement)`** and **`tailSet(fromElement)`** for returning a portion of the set whose elements are less than **`toElement`** and greater than or equal to **`fromElement`**.

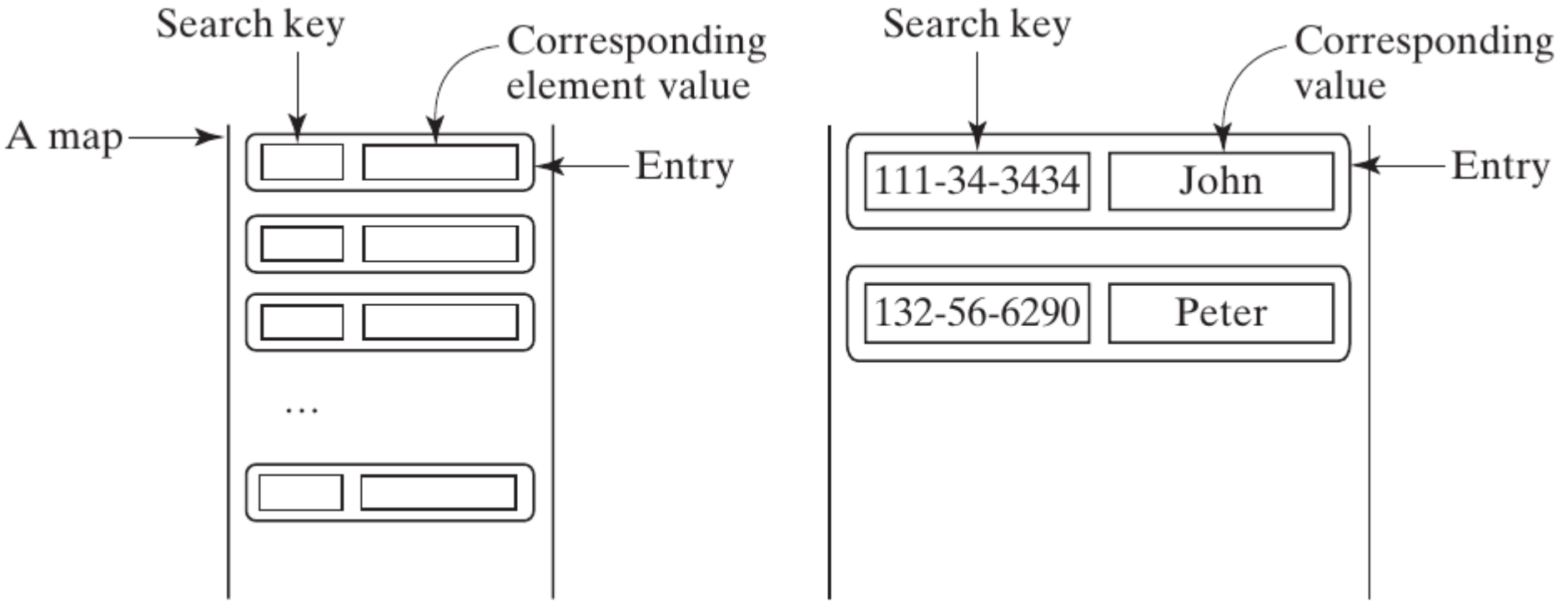
- NavigableSet extends SortedSet:
  - provides navigation methods **`lower(e)`**, **`floor(e)`**, **`ceiling(e)`** and **`higher(e)`** which return elements less than, less than or equal, greater than or equal, and greater than a given element or **`null`** if there is no such element
  - **`pollFirst()`** and **`pollLast()`** remove and return the first and last elements in the **`TreeSet`**
  -

- A TreeSet can be sorted according to Comparable or using a Comparator

Maps

Maps

- ***map***: like a dictionary that provides a quick lookup to retrieve a value using a key (stores key/value pairs)
  - keys are like indexes but keys could really be any data type, not just integer
  - maps cannot contain duplicate keys
  - sort of like the primary key of a database if you took CS-1222



Map Types

- **`HashMap`**
  - efficient for locating a value, inserting an entry, deleting an entry
  - entries are not ordered
- **`LinkedHashMap`**
  - extends **`HashMap`** with linked-list implementation
  - supports an ordering of the entries in the map
  - generally insertion order or the order they were last accessed from least recently accessed to most recently accessed
    - this depends on which Constructor was used to make the Map
- **`TreeMap`**
  - efficient for traversing the **keys** in sorted order
  - keys can be sorted using Comparable or Comparator.