Map , HashTable & Dictionary

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

The Map ADT (§ 8.1)

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated 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()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

Example

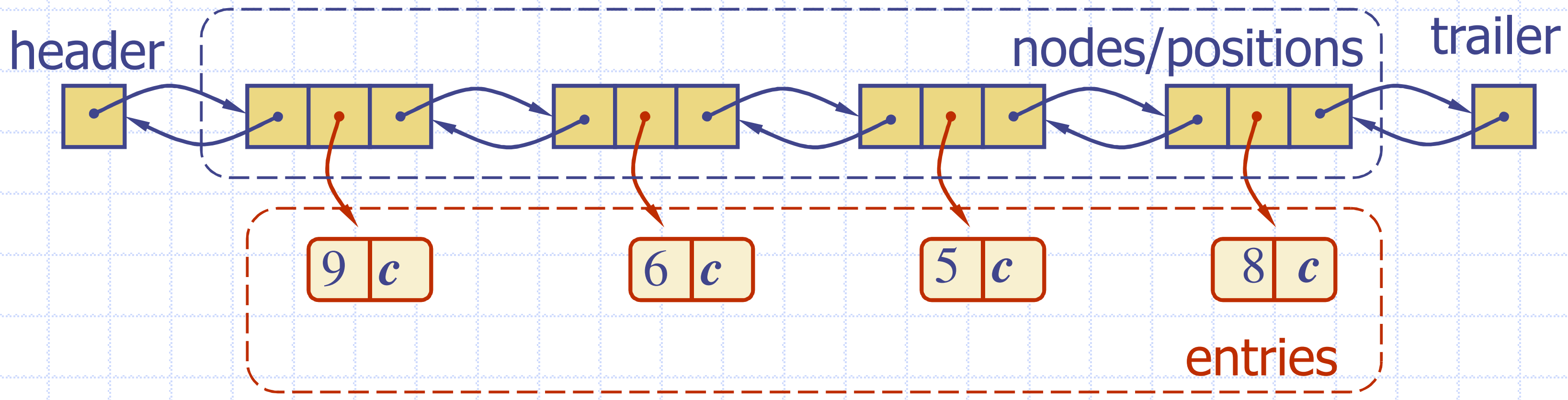
Operation	***Output***	***Map***
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
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*)

Comparison to java.util.Map

Map ADT Methods	java.util.Map **Methods**
size()	size()
isEmpty()	isEmpty()
get(*k*)	get(*k*)
put(*k*,*v*)	put(*k*,*v*)
remove(*k*)	remove(*k*)
keys()	keySet().iterator()
values()	values().iterator()

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().key() = k    then
            return p.element().value()
    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().key() = $k$ **then**
        $t$ = $p$.element().value()
        $B$.replace($p$,($k$,$v$))
        **return** $t$            {return the old value}
$S$.insertLast(($k$,$v$))
$n$ = $n$ + 1          {increment variable storing number of entries}
**return null**        {there was no previous entry with key equal to $k$}
```

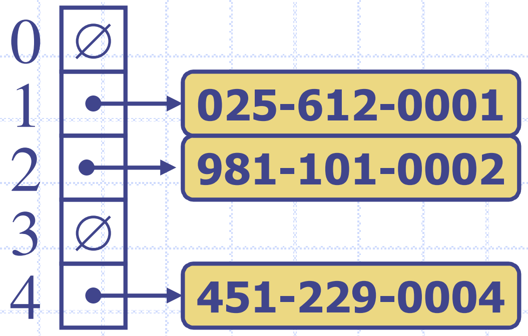
The remove(k) Algorithm

```
**Algorithm** remove($k$):
$B = S$.positions()
while $B$.hasNext() do
    $p = B$.next()
    if $p$.element().key() = $k$ then
        $t = p$.element().value()
        $S$.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



Recall the Map ADT (§ 8.1)

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated 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()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

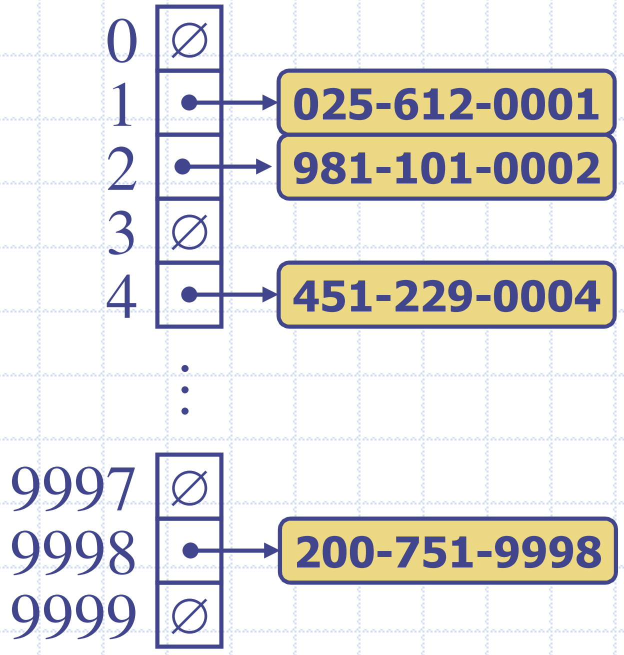
Hash Functions and Hash Tables (§ 8.2)

- 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 $h(x) =$ last four digits of x



Hash Functions (§ 8.2.2)

- 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 Codes (§ 8.2.3)

- 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 $p(z) = p_{n-1}(z)$

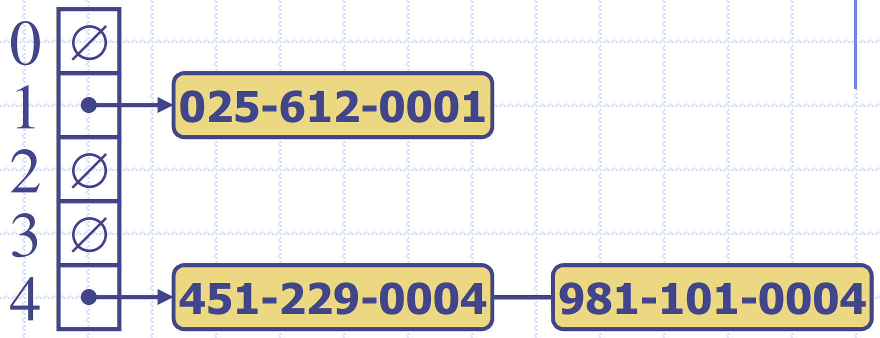
Compression Functions (§ 8.2.4)

- 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

Collision Handling (§ 8.2.5)

- 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 Methods with Separate Chaining used for Collisions

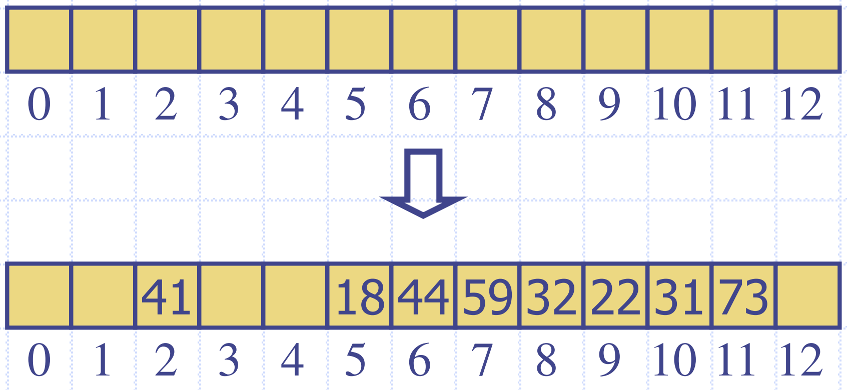
- Delegate operations to a list-based map at each cell:

```
Algorithm get(k):
Output: The value associated with the key k in the map, or null if there is no
    entry with key equal to k in the map
return A[h(k)].get(k)          {delegate the get to the list-based map at A[h(k)]}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we
    return its value (replacing it with v); otherwise, we return null
t = A[h(k)].put(k,v)           {delegate the put to the list-based map at A[h(k)]}
if t = null then               {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or null if there
    is no entry with key equal to k in the map
t = A[h(k)].remove(k)       {delegate the remove to the list-based map at A[h(k)]}
if t ≠ null then            {k was found}
    n = n - 1
return t
```

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



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)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return null
        else if $c.key() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return null
```

Updates with Linear Probing

- 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, o) is found, we replace it with the special item $AVAILABLE$ 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 $AVAILABLE$, or
    - N cells have been unsuccessfully probed
  - We store entry (k, o) in cell i

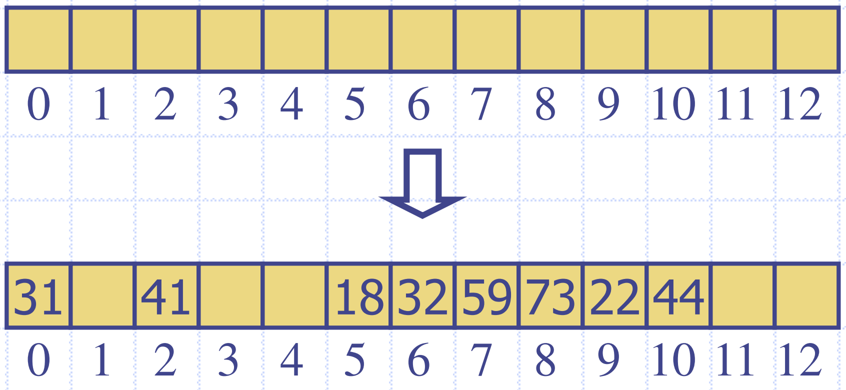
Double Hashing

- 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		



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

Java Example

```
/** A hash table with linear probing and the MAD hash function */
public class HashTable implements Map {
  protected static class HashEntry implements Entry {
    Object key, value;
    HashEntry () { /* default constructor */ }
    HashEntry(Object k, Object v) { key = k; value = v; }
    public Object key() { return key; }
    public Object value() { return value; }
    protected Object setValue(Object v) {  // set a new value, returning old
      Object temp = value;
      value = v;
      return temp;  // return old value
    }
  }
  /** Nested class for a default equality tester */
  protected static class DefaultEqualityTester implements EqualityTester {
    DefaultEqualityTester() { /* default constructor */ }
    /** Returns whether the two objects are equal.  */
    public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
  }
  protected static Entry AVAILABLE = new HashEntry(null, null); // empty
        marker
  protected int n = 0;                    // number of entries in the dictionary
  protected int N;                        // capacity of the bucket array
  protected Entry[] A;                                  // bucket array
  protected EqualityTester T;            // the equality tester
  protected int scale, shift;   // the shift and scaling factors
  /** Creates a hash table with initial capacity 1023. */
  public HashTable() {
    N = 1023; // default capacity
    A = new Entry[N];
    T = new DefaultEqualityTester(); // use the default equality tester
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

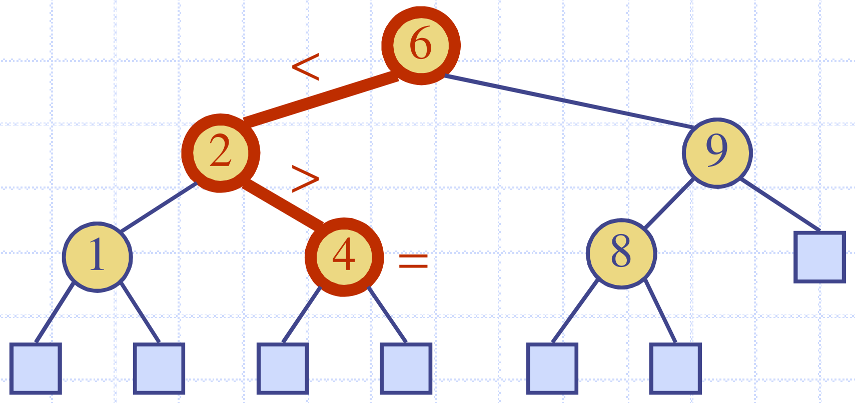
```
/** Creates a hash table with the given capacity and equality tester. */
  public HashTable(int bN, EqualityTester tester) {
    N = bN;
    A = new Entry[N];
    T = tester;
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

Java Example (cont.)

```
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
  if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashValue(Object key) {
  return Math.abs(key.hashCode()*scale + shift) % N;
}
/** Returns the number of entries in the hash table. */
public int size() { return n; }
/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }
/** Helper search method - returns index of found key or -index-1,
 * where index is the index of an empty or available slot. */
protected int findEntry(Object key) throws InvalidKeyException {
  int avail = 0;
  checkKey(key);
  int i = hashValue(key);
  int j = i;
  do {
    if (A[i] == null)   return -i - 1;  // entry is not found
    if (A[i] == AVAILABLE) {              // bucket is deactivated
        avail = i;                        // remember that this slot is available
        i = (i + 1) % N;                  // keep looking
    }
    else if (T.isEqualTo(key,A[i].key()))  // we have found our entry
        return i;
    else // this slot is occupied--we must keep looking
        i = (i + 1) % N;
  } while (i != j);
  return -avail - 1;  // entry is not found
}
/** Returns the value associated with a key. */
public Object get (Object key) throws InvalidKeyException {
  int i = findEntry(key);  // helper method for finding a key
  if (i < 0) return null;  // there is no value for this key
  return A[i].value();     // return the found value in this case
}
```

```
/** Put a key-value pair in the map, replacing previous one if it exists. */
public Object put (Object key, Object value) throws InvalidKeyException {
  if (n >= N/2) rehash(); // rehash to keep the load factor <= 0.5
  int i = findEntry(key); //find the appropriate spot for this entry
  if (i < 0) {      // this key does not already have a value
    A[-i-1] = new HashEntry(key, value); // convert to the proper index
    n++;
    return null;    // there was no previous value
  }
  else                                  // this key has a previous value
    return ((HashEntry) A[i]).setValue(value); // set new value & return old
}
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  N = 2*N;
  Entry[] B = A;
  A = new Entry[N]; // allocate a new version of A twice as big as before
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(N-1) + 1;                    // new hash scaling factor
  shift = rand.nextInt(N);                          // new hash shifting factor
  for (int i=0; i&ltB.length; i++)
    if ((B[i] != null) && (B[i] != AVAILABLE)) { // if we have a valid entry
      int j = findEntry(B[i].key());  // find the appropriate spot
      A[-j-1] = B[i];                 // copy into the new array
    }
}
/** Removes the key-value pair with a specified key. */
public Object remove (Object key) throws InvalidKeyException {
  int i = findEntry(key);           // find this key first
  if (i < 0) return null;           // nothing to remove
  Object toReturn = A[i].value();
  A[i] = AVAILABLE;                                 // mark this slot as
      deactivated
  n--;
  return toReturn;
}
/** Returns an iterator of keys. */
public java.util.Iterator keys() {
  List keys = new NodeList();
  for (int i=0; i&ltN; i++)
    if ((A[i] != null) && (A[i] != AVAILABLE))
      keys.insertLast(A[i].key());
  return keys.elements();
}
} // ... values() is similar to keys() and is omitted here ...
```

Dictionaries



Dictionary ADT (§ 8.3)

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

- Dictionary ADT methods:
  - find(k): if the dictionary has an entry with key k, returns it, else, returns null
  - findAll(k): returns an iterator of all entries with key k
  - insert(k, o): inserts and returns the entry (k, o)
  - remove(e): remove the entry e from the dictionary
  - entries(): returns an iterator of the entries in the dictionary
  - size(), isEmpty()

Example

Operation	***Output***	***Dictionary***
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*),(7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(7)	(7,*B*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(4)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(2)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
findAll(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(find(5))	(5,*A*)	(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(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:
  - insert takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find 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 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 findAll(k) Algorithm

```
Algorithm findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k

Create an initially-empty list L
B = D.entries()
while B.hasNext() do
    e = B.next()
    if e.key() = k then
        L.insertLast(e)
return L.elements()
```

The insert and remove Methods

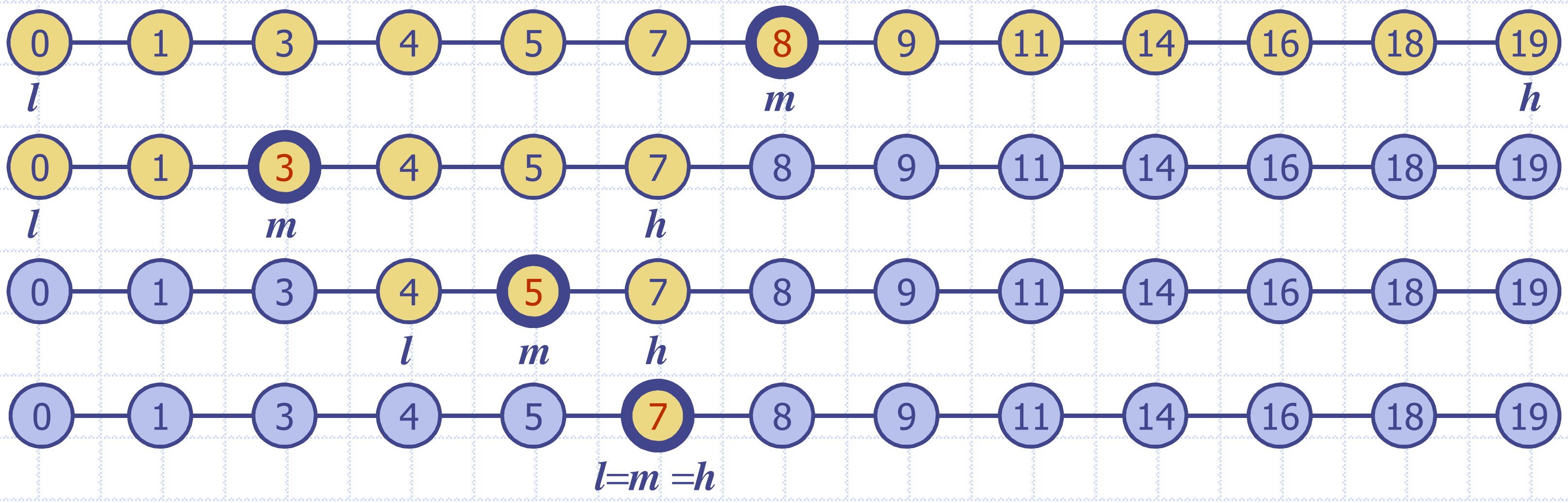
```
Algorithm insert(k,v):
Input: A key k and value v
Output: The entry (k,v) added to D
Create a new entry e = (k,v)
S.insertLast(e)          {S is unordered}
return e

Algorithm remove(e):
Input: An entry e
Output: The removed entry e or null if e was not in D
{We don’t assume here that e stores its location 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.

Binary Search

- Binary search performs operation find(k) on a dictionary 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 a logarithmic number of steps
- Example: find(7)



Search Table

- A search table is a dictionary implemented by means of a sorted array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - find takes $O(\log n)$ time, using binary search
  - insert 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
- 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 (e.g., credit card authorizations)