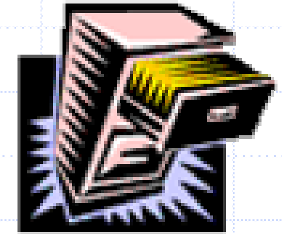
Maps and hashing

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



- Map ADT 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, 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*)	*It returns null because it is not already in M*
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*)	*Because 4 is not in M*
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 5 and return A
remove(2)	*E*	(7,*B*),(8,*D*)	
get (2)	null	(7,*B*),(8,*D*)	*Because 4 is not in M*
isEmpty()	false	(7,*B*),(8,*D*)	*Because there are 2 in M*

Comparison to java.util.Map

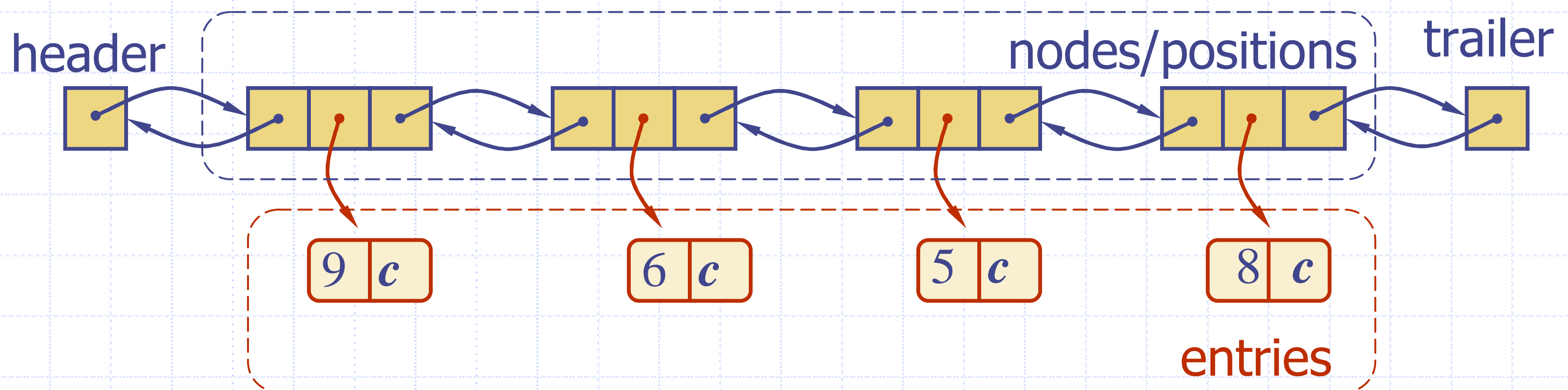
Map ADT Methods	java.util.Map Methods
size()	size()
isEmpty()	isEmpty()
get(*k*)	get(*k*)
put(*k*,*v*)	put(*k*,*v*)
remove(*k*)	remove(*k*)

All same except:

Map ADT Methods	java.util.Map Methods
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()  if the next position in Bg
        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)

Hashing

Content

- Idea: when using various operations on binary search trees, we use hash table ADT
- Hashing: implementation of hast tables for insert and find operations is called hashing.
- Collision: when two keys hash to the same value
- Resolving techniques for collision using linked lists
- Rehashing: when a hash table is full, then operations will take longer time. Then we need to build a new double sized hash table. (HWLA??? Why double sized..)
- Extendible hashing: for fitting large data in main memory

Hashing as a Data Structure

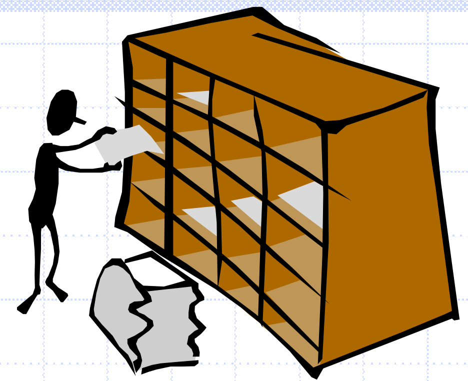
- Performs operations in O(1)
  - Insert
  - Delete
  - Find
- Is not suitable for
  - FindMin
  - FindMax
  - Sort or output as sorted

General Idea

- Array of Fixed Size (*TableSize*)
- Search is performed on some part of the item (*Key*)
- Each key is mapped into some number between 0 and (TableSize-1)
- Mapping is called a *hash function*

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash Functions and Hash Tables

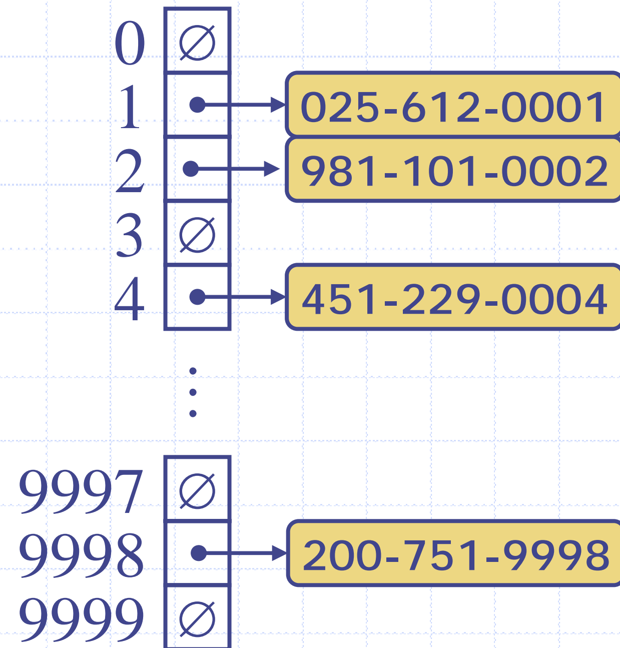


- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $$h(x) = x \bmod N$$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

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

Example

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



Hash Functions

- Easy to compute
  - Key is of type Integer
  - Key is of type String

Hash Function (Integer)

- Simply return *Key* % *TableSize*
- Choose carefully TableSize
  - TableSize is 10, all keys end in zero???
- To avoid such pitfalls, choose TableSize a prime number

Hash Function I (String)

- Adds up ASCII values of characters in the string
- Advantage: Simple to implement and computes quickly
- Disadvantage: If TableSize large, function does not distribute keys well
  - Example: Keys are at most 8 characters. Maximum sum (8*256 = 2048), but TableSize 10007. Only 25 percent could be filled.

Hash Function II (String)

- Assumption: *Key* has at least 3 characters
- Hash Function: (26 characters + blank)

  key[0] + 27 * key[1] + 27^2 * key[2]
- Advantage: Distributes well than Hash Function I
- Disadvantage:
  - 26^3 = 17,576 possible combinations
  - English has only 2,851 different combinations by a dictionary check

Hash Function III (String)

- Idea: Computes a polynomial function of *Key*'s characters

P(*Key* with n+1 characters) =

$Key[0]+37Key[1]+37^2Key[2]+...+37^nKey[n]$

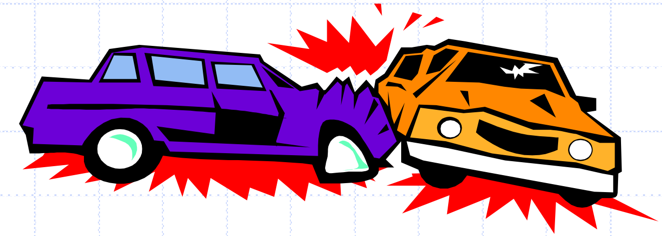
- If find 37^n then sum up complexity $O(n^2)$
- Using Horner's rule complexity drops to O(n)

$((Key[n]*37+Key[n-1])*37+...+Key[1])*37+Key[0]$

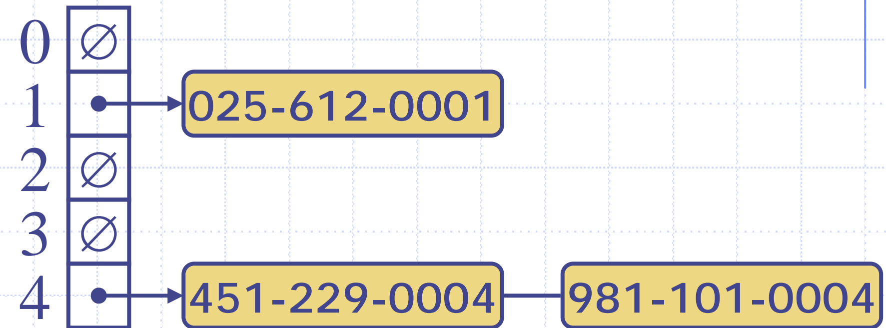
Hash Function III (String)

```
public static int hash( String key, int tableSize )
        {
            int hashVal = 0;
            for( int i = 0; i < key.length( ); i++ )
                hashVal = 37 * hashVal + key.charAt( i );
            hashVal %= tableSize;
            if( hashVal < 0 )
                hashVal += tableSize;
            return hashVal;
        }
```

Collision

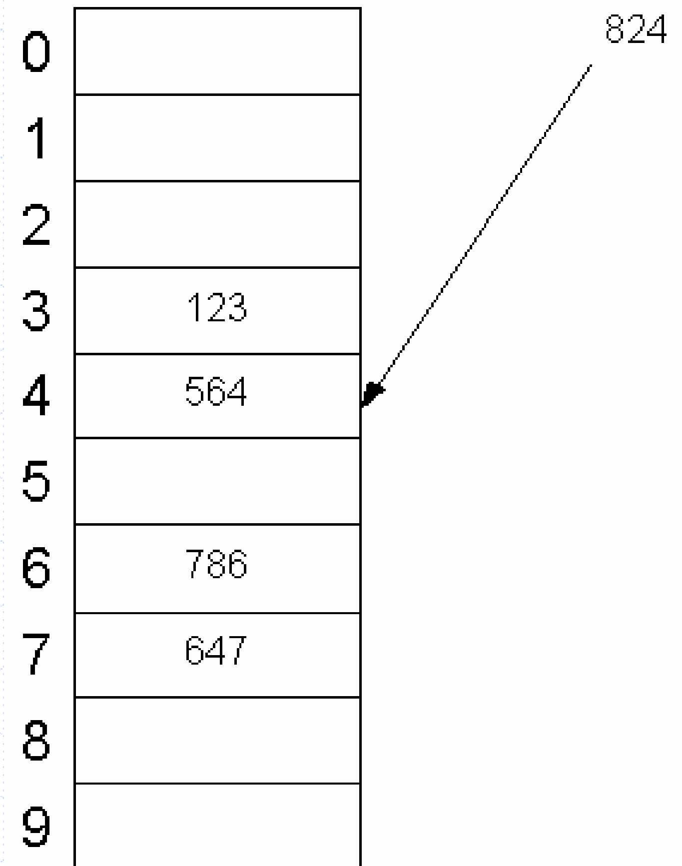


- Collisions occur when different elements are mapped to the same cell



Collision

- When an element is inserted, it hashes to the same value as an already inserted element we have *collision*.
- Example: Hash Function (Key % 10)



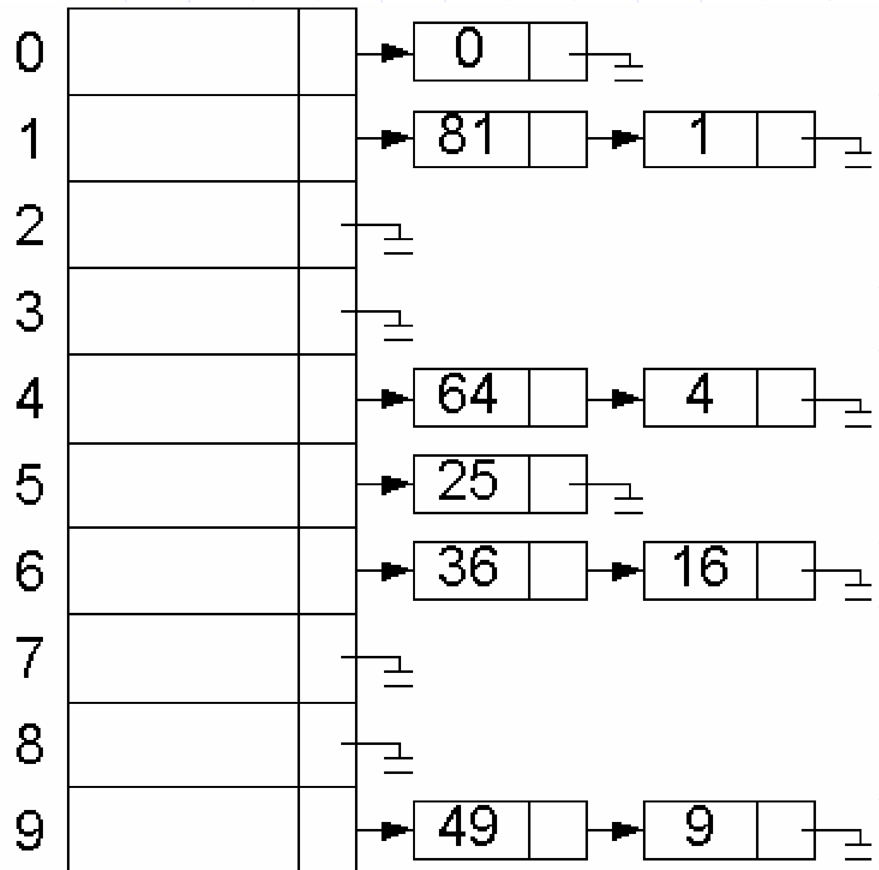
Solving Collision

- Separate Chaining:
  - keep a list of all elements hashing to the same value, and traverse the list to find corresponding hash. Hint: lists should be large and kept as prime number table size to ensure a good distribution. (Limited use due to space limitations of lists, and needs linked lists!!!)
- Open Addressing:
  - at a collision, search for alternative cells until finding an empty cell.
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- HWLA: Search Internet for other techniques (is there any without using a linked list)

Separate Hashing

- Separate Chaining: let each cell in the table point to a linked list of entries that map there
- Keep a list of all elements that hash to the same value
- Each element of the hash table is a Link List
- Example: x^2 % 10

- Separate chaining is simple, but requires additional memory outside the table



Separate Hashing

```
/**
 * Construct the hash table.
 */
public SeparateChainingHashTable( ){
    this( DEFAULT_TABLE_SIZE );
}

/**
 * Construct the hash table.
 * @param size approximate table size.
 */
public SeparateChainingHashTable( int size ){
    theLists = new LinkedList[ nextPrime( size ) ];
    for( int i = 0; i < theLists.length; i++ )
        theLists[ i ] = new LinkedList( );
}
```

Separate Hashing

- Find
  - Use hash function to determine which list to traverse
  - Traverse the list to find the element

```
public Hashable find( Hashable x ){
  return (Hashable)theLists[ x.hash( theLists.length ) ]
.find(x).retrieve( );
       }
```

Separate Hashing

- Insert
  - Use hash function to determine in which list to insert
  - Insert element in the header of the list

```
public void insert( Hashable x ){
            LinkedList whichList = theLists[x.hash(theLists.length) ];
            LinkedListItr itr = whichList.find( x );
            if( itr.isPastEnd( ) )
                whichList.insert( x, whichList.zeroth( ) );
        }
```

Separate Hashing

- Delete
  - Use hash function to determine from which list to delete
  - Search element in the list and delete

```
public void remove( Hashable x ){
          theLists[ x.hash( theLists.length ) ].remove( x );
       }
```

Separate Hashing

- Advantages
  - Solves the collision problem totally
  - Elements can be inserted anywhere
- Disadvantages
  - All lists must be short to get O(1) time complexity

Separate Hashing

- Alternatives to Link Lists
  - Binary Trees
  - Hash Tables

Open Addressing

- Solving collisions without using any other data structure such as link list
- If collision occurs, alternative cells are tried until an empty cell is found
- Cells $h_0(x)$, $h_1(x)$, ..., are tried in succession
- $h_i(x) = (hash(x) + f(i)) \% \mathit{TableSize}$

Open Addressing

- Linear Probing
  - $f(i) = i$
- Quadratic Probing
  - $f(i) = i^2$
- Double Hashing
  - $f(i) = i \, hash_2(x)$

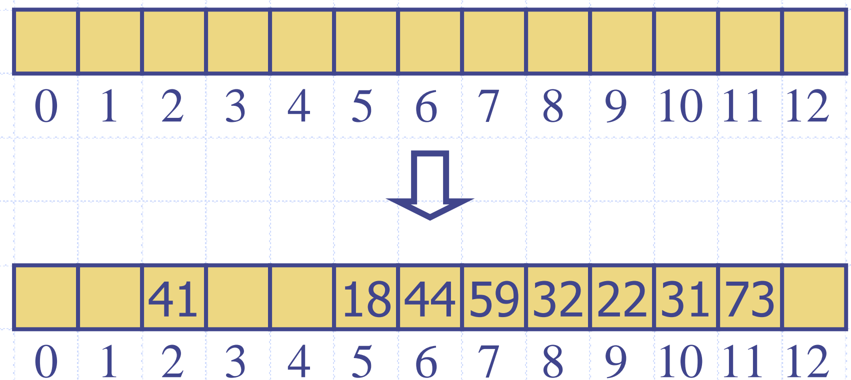
Linear Probing

- Advantages
  - Easy to compute
- Disadvantages
  - Table must be big enough to get a free cell
  - Time to get a free cell may be quite large
  - *Primary Clustering*
    - Any key that hashes into the cluster will require several attempts to resolve the collision

Example: 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



Example in the book: Linear Probing

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing

- Eliminates *Primary Clustering* problem
- Theorem: If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty
- *Secondary Clustering*
  - Elements that hash to the same position will probe the same alternative cells

Quadratic Probing

$$H_i(x) = x \% 10 + i^2$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

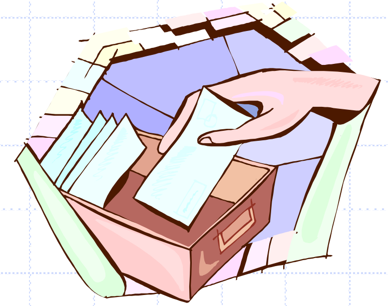
Quadratic Probing

```
/**
 * Construct the hash table.
 */
public QuadraticProbingHashTable( )
{
    this( DEFAULT_TABLE_SIZE );
}
/**
 * Construct the hash table.
 * @param size the approximate initial size.
 */
public QuadraticProbingHashTable( int size )
{
    allocateArray( size );
    makeEmpty( );
}
```

Quadratic Probing

```
/**
     * Method that performs quadratic probing resolution.
     * @param x the item to search for.
     * @return the position where the search terminates.
     */
    private int findPos( Hashable x )
    {
/* 1*/      int collisionNum = 0;
/* 2*/      int currentPos = x.hash( array.length );
/* 3*/      while( array[ currentPos ] != null &&
                    !array[ currentPos ].element.equals( x ) )
            {
/* 4*/          currentPos += 2 * ++collisionNum - 1;
/* 5*/          if( currentPos >= array.length )
/* 6*/              currentPos -= array.length;
            }
/* 7*/      return currentPos;
    }
```

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 $j = 0,\ 1,\ \dots\ ,\ N - 1$

- The secondary hash function $/(k)$ 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:

$$/_{\in}(k) = q - k \bmod q$$

where

  - $q < N$
  - q is a prime

- The possible values for $/_{\in}(k)$ are

$$1,\ 2,\ \dots\ ,\ q$$

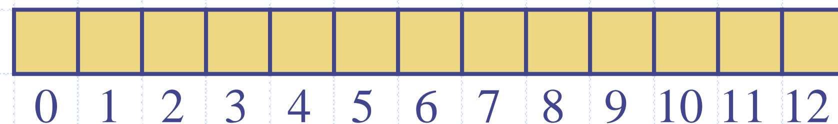
Double Hashing

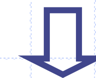
- Poor choice of $hash_2(x)$ could be disastrous
- $hash_2(x) = R - (x \% R)$
- R a prime smaller than *TableSize*
- If double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy

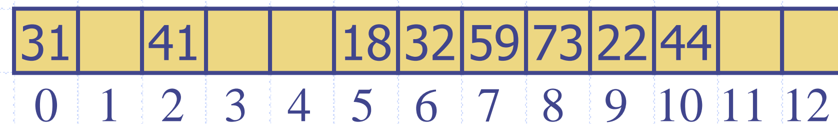
Example of Double Hashing

- 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		







Example in the book Double Hashing

$H_i(x) = x \% 10 + i\,(R - (x \bmod R)) \quad R = 7$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing

- If Hash Table gets too full
- Running time for the operations will start taking too long time
- Insertions might fail for open addressing with quadratic probing
- Solution: Rehashing

Rehashing

- Build another table that is about twice as big
- Associate a new hash function
- Scan down the entire original hash table
- Compute the new hash value for each element
- Insert it in the new table

Rehashing

- Expensive operation O(N)
- Not bad, occurs very infrequently
- If data structure is part of the program, effect is not noticeable

Rehashing

- When to apply rehashing?
  - As soon as the table is half full
  - Only when an insertion fails
  - When the table reaches a certain load factor

Rehashing

- Use 17

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing

```
private void allocateArray( int arraySize )
       {
           array = new HashEntry[ arraySize ];
       }
private void rehash( )
       {
           HashEntry [ ] oldArray = array;
               // Create a new double-sized, empty table
           allocateArray( nextPrime( 2 * oldArray.length ) );
           currentSize = 0;
               // Copy table over
           for( int i = 0; i < oldArray.length; i++ )
               if( oldArray[ i ] != null && oldArray[ i ].isActive)
                   insert( oldArray[ i ].element );
           return;
       }
```

Extendible Hashing (Why?)

- Amount of data is too large to fit in main memory
- Main consideration is the number of disk accesses required to get data

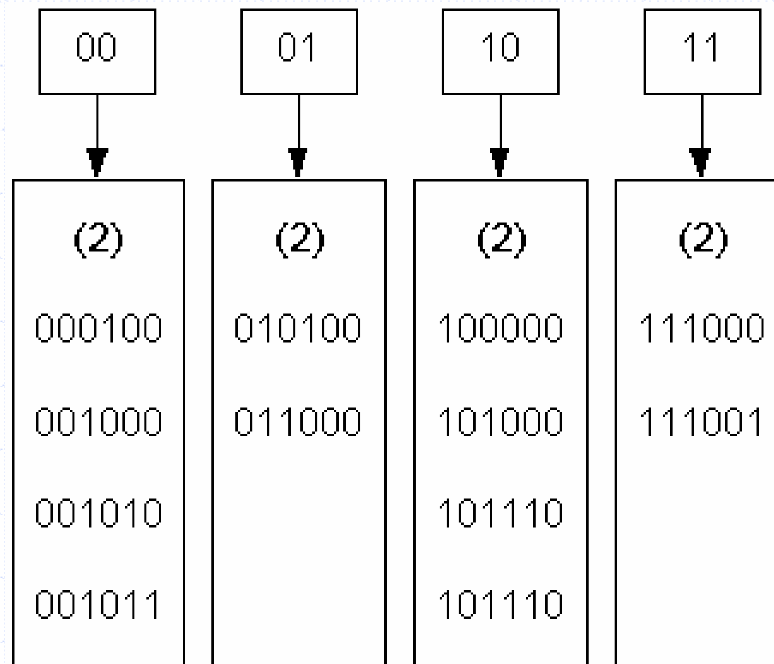
Extendible Hashing (Why?)

- Open addressing or separate chaining is used, collisions could cause several blocks to be examined
- When the table gets too full, rehashing step requires O(N) disk accesses

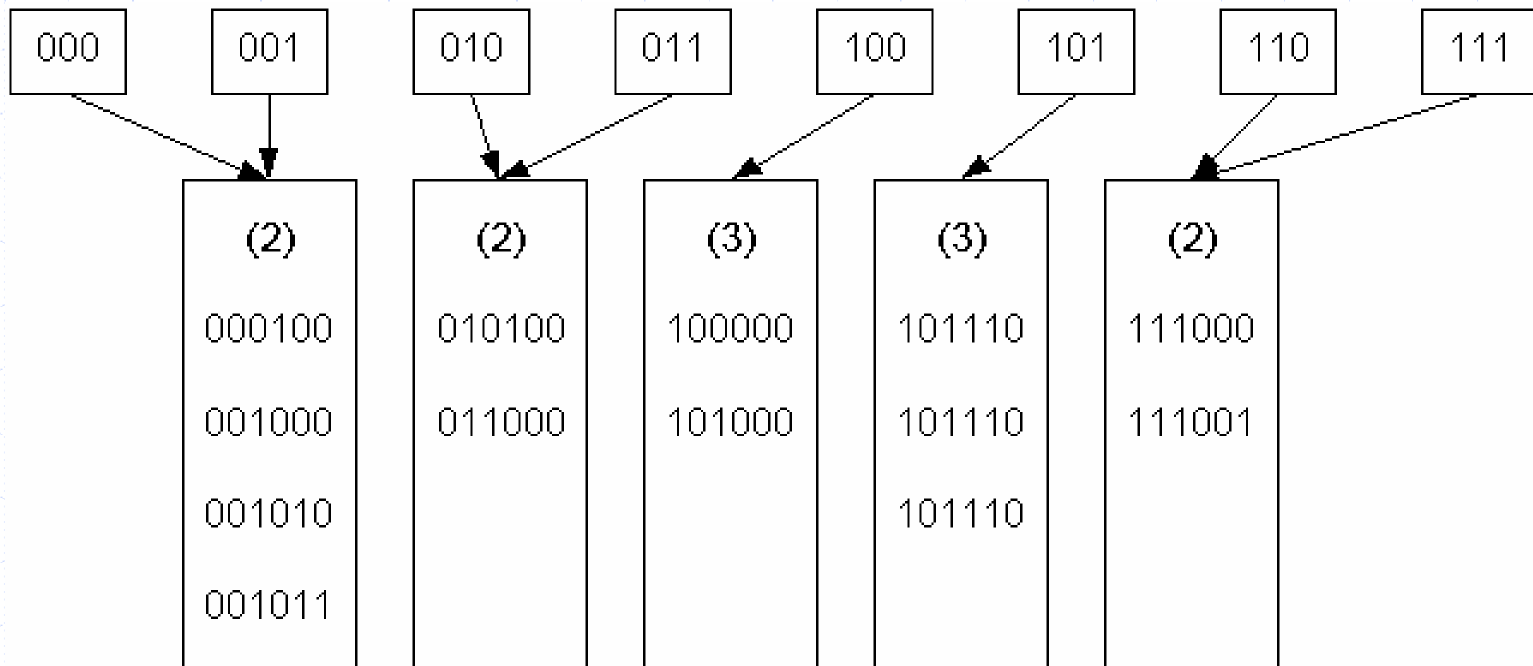
Extendible Hashing

- Use of idea in B-Trees
- Choose of M so large that B-Tree has a depth of 1
- Problem: Branching factor is too high, requires to much time to determine which leaf the data was in
- Time to perform this step is reduced

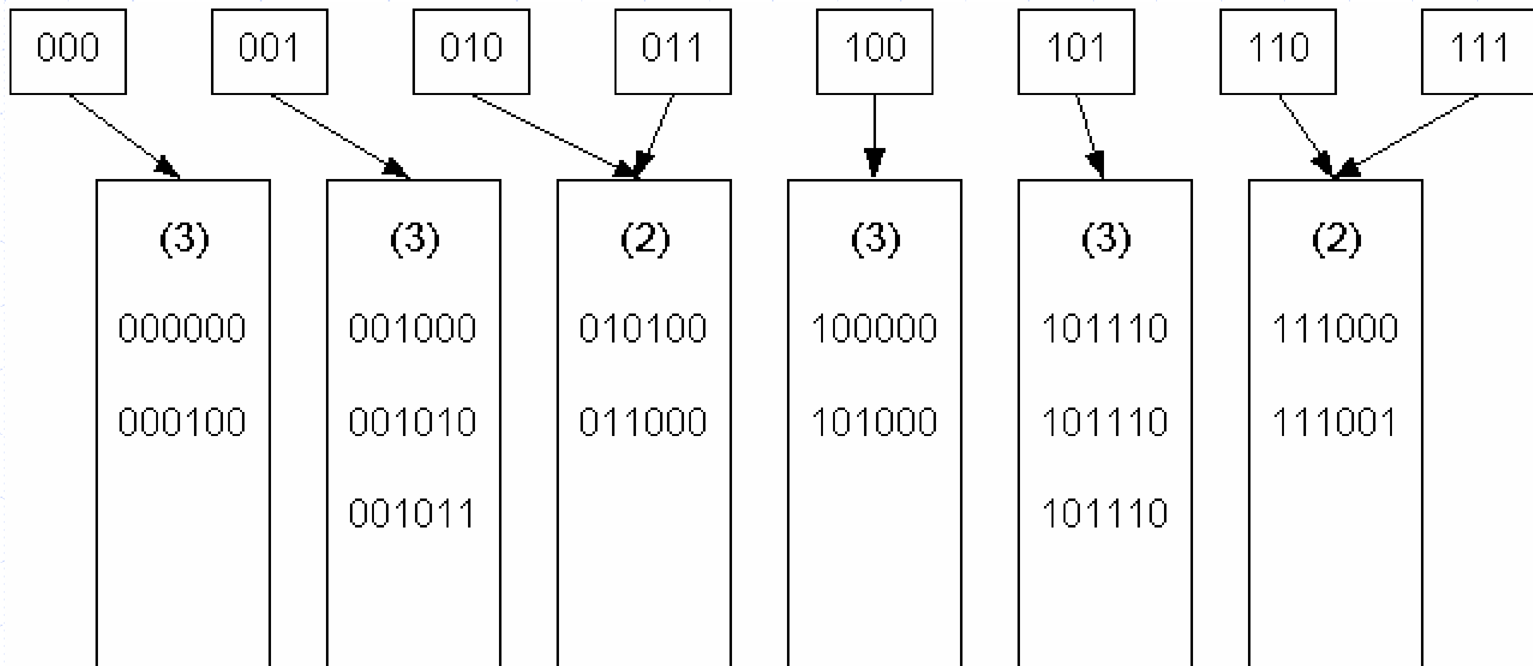
Extendible Hashing



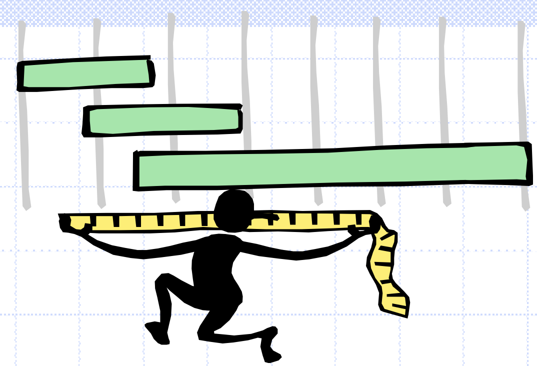
Extendible Hashing



Extendible Hashing



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 ...
```

HWs

Problem 2 in the book:

When rehashing, we choose a table size that is roughly twice as large and prime. In our case, the appropriate new table size is 19, with hash function $h(x) = x \pmod{19}$.

(a) Scanning down the separate chaining hash table, the new locations are 4371 in list 1, 1323 in list 12, 6173 in list 17, 4344 in list 12, 4199 in list 0, 9679 in list 8, and 1989 in list 13.

(b) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 14 because both 12 and 13 are already occupied, and 4199 in bucket 0.

(c) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 16 because both 12 and 13 are already occupied, and 4199 in bucket 0.

(d) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 15 because 12 is already occupied, and 4199 in bucket 0.

- Problems in CHP5: 1, 4, 5, 11, 16

Hash Functions (*)

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

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

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

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

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes (*)

- 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

    $$p_0(z) = a_{n-1}$$

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

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

Compression Functions (*)

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

    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Map Methods with Separate Chaining used for Collisions (*)

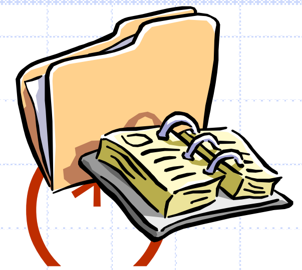
- Delegate operations to a list-based map at each cell:

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 $t \neq$ **null then** {k was found}
  $n = n - 1$
return t

Search with Linear Probing (*)



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

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

Updates with Linear Probing(*)

- To handle insertions and deletions, we introduce a special object, called $AVAILABLE$, 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