# Data Structures and Algorithms

## The Map Abstract Data Type

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

November 19, 2018

# Learning outcomes

After this lecture and the related practical students should...

- be able to implement an doubly linked list
- understand the use of polymorphism to implement a data structure that can store any type of data

# Table of Contents

# The Map Abstract Data Type

## Concept

- A map is an Abstract Data Type that stores key-value pairs known as entries
- Each entry must have a unique key
- This key is used to access the associated value
- Maps are often referred to as associative stores

# The Entry Abstract Data Type

## Concept

Before we can fully understand the Map abstract data type we need to look at the Entry abstract data type

- The Entry abstract data type models the idea of a key and an associated piece of data
- These two are linked together
- When an entry is stored, the key is used as an index
- Keys can be any type of object, but in our implementation we will use integers
- Maps support the retrieval of values associated with keys
- If no value is associated with a key, `null` is returned

# The Entry Abstract Data Type

## Specification

- Operation:
  - key(): This returns the key of the entry
  - value(): this returns the value stored in the entry

# The Entry Abstract Data Type

## Implementation

```java
public class Entry {
  private int key;
  private Object value;
  public Entry(int k, Object v) {
    key = k;
    value = v;
  }
  public int key() {
    return key;
  }
  public Object value() {
    return value;
  }
}
```

# The Map Abstract Data Type

## Specification

- get(k): If the map contains an entry `e`, with a key equal to `k`, then we return the value in `e`, otherwise return `null`
- put(k, v): If the map does not have an entry with key equal to `k`, then add a new entry with key `k` and value `v` to the map and return null, otherwise update the entry to associate `k` with `v` and return the old value
- remove(k): Remove the entry with key k and return the value, if no matching entry exists return `null`
- keys(): Return an iterator for the keys stored in the map
- values(): Return an iterator of the values stored in the map
- entries(): Return an iterator of the entries stored in the map

# The Map Abstract Data Type

## Interface

```java
public interface Map {
  public int size();
  public boolean isEmpty();
  public Object get(int k);
  public Object put(int k, Object v);
  public Object remove(int k);
  public Iterator entries();
}
```

# The Map Abstract Data Type

## Interface

```java
public interface Map {
  public int size();
  public boolean isEmpty();
  public Object get(int k);
  public Object put(int k, Object v);
  public Object remove(int k);
  public Iterator entries();
}
```

# The Map Abstract Data Type

Idea

- Whenever we perform a put() operation, an entry is added to the map containing the corresponding key and value
- Whenever we perform a get() operation, we return the value part of the entry whose key matches the argument
- Whenever we perform a put() operation, if an entry already exists with the same key, we replace it

| Operations | Map Contents |
|---|---|
| put("01234567", "David Lillis")<br>put("69234567", "Abey Campbell")<br>put("72234567", "Lina Xu")<br>get("01234567") = "David Lillis"<br>get("01234577") = null | {"72234567", "Lina Xu"}<br>{"69234567", "Abey Campbell"}<br>{"01234567", "David Lillis"} |

# The Map Abstract Data Type

Idea

- Whenever we perform a `put()` operation, an entry is added to the map containing the corresponding key and value
- Whenever we perform a `get()` operation, we return the value part of the entry whose key matches the argument
- Whenever we perform a `put()` operation, if an entry already exists with the same key, we replace it

| Operations | Map Contents |
|---|---|
| put("01234567", "David Lillis") | |
| put("69234567", "Abey Campbell") | |
| put("72234567", "Lina Xu") | {"72234567", "Lina Xu"} |
| get("01234567") = "David Lillis" | |
| get("01234577") = null | {"69234567", "Abey Campbell"} |
| put("01234567", "Sean Russell") | |
| | {"01234567", "Sean Russell"} |

# The Map Abstract Data Type

## Implementation Strategies

- The easiest way to implement a map is to view it as a list of entries
  - We can utilize the pre-existing List ADT
  - Our Map implementation creates and manipulates a List object
  - In order to manipulate entries, we must find them first, this is O(n)
- Array based implementation
  - An array of size N
  - Hashing function (shown as `h(k)` )used to map keys to integer values in the correct range
    - $0 \leq x < N$
    - Usually uses % N
  - We also need a collision handling strategy
  - This deals with the case where two keys have the same hash value

# Table of Contents

# List Based Map Implementation

- How the list based implementation will work, first we create a class called `ListMap`
- We create a list data structure that can be used to store entries
- Every time we add or remove an entry, it is added or removed from the list
- Methods control how entries are added and removed

**Variables**

- A reference to the list
- `private List list = new DLList();`

# List based Map Operations

- size()
  - Return the size that is stored in the list
- isEmpty()
  - Return the result of the expression list.isEmpty()
- entries()
  - We will come back to this later when we have studied iterators

# List based Map Operations

## Finding Entries

- Every time we perform and of the operations get, put or remove, first we must find the correct entry in the list
- Because we do not want to write the same code 3 times, we will write this as a private method
- `private Position find(int k){...}`
- This method will return the Position that the Entry is stored in, from this we can
  - Access the key and value of the entry
  - Remove the position from the list
  - Replace the position wilt a new entry

# List based Map Operations

## Finding Entries

- We are searching the list for a position containing an entry with the key k
- Get the position p which is first in the list
- While p is not the last position in the list...
  - Compare the key in the entry in p with k
  - If they match return p
  - If they don't replace p with the position after p
- Compare the key in the entry p with k
  - If they match return p
  - If they don't return null

# find(k)

```
Algorithm find(k):
Input:    A key, k
Output:   The position, p, that entry with key k is
    stored in

p ← list.first()
last ← list.last()
while p <> last do
  if p.element().key() = k then
    return p
  else
    p ← list.after(p)
if NOT list.isEmpty() AND p.element().key() = k then
  return p
else
  return null
```

# get(k)

- This method takes a key as a parameter and returns the value associated with that key or null
- First we have to find the position in the list containing the key k
- If this position is null, then the key is not in the list and we return null
- If the position is not null, we need to type cast the element to an Entry object and return the value

# get(k)

```
Algorithm get(k):
Input:     A key, k
Output:    The value, v, associated with k

p ← find(k)
if (p = null) then
  return null
return p.element().value()
```

# remove(k)

- This method takes a key as a parameter and removes the associated entry from the map
- The value contained in the entry is then returned
- First we have to find the position in the list containing the key k
- If this position is null, then the key is not in the list and we return null
- If the position is not null, we need to remove it from the list
- Finally we need to type cast the element to an Entry object and return the value

# remove(k)

```
Algorithm remove(k):
Input:     A key, k
Output:    The value, v, associated with k

p ← find(k)
if (p = null) then
  return null
list.remove(p)
return p.element().value()
```

# put(k, v)

- If the map has an entry with the key k, we replace it with the new value v and return the old value
- If the map has no value with the key k, we add a new entry and return null
- First we have to find the position, p, in the list containing the key k
- If this position is null
  - Create a new entry, e, containing the key k and value v
  - Add e to the list
  - return null
- If the position is not null
  - Create a new entry, e, containing the key k and value v
  - Add e to the list in the position after p
  - remove p from the list
  - return p.element().value()

# put(k, v)

```
Algorithm put(k, v):
Input:    A key, k and a value v associated
    with it
Output:   The value that was replaced or null

p ← find(k)
if (p = null) then
   create new entry e containing k and v
   Add e to the end of the list
   return null
else
   create new entry e containing k and v
   list.insertAfter(p, e)
   list.remove(p)
   return p.element().value()
```

# Performance

- The performance of get, put and remove depend on the find method
- The find method has the running time is O(n)

| Operation | Expected Running Time |
|---|---|
| size | O(1) |
| isEmpty | O(1) |
| get | O(n) |
| put | O(n) |
| remove | O(n) |

# Table of Contents

# The Map Abstract Data Type

## Implementation Strategies

- The easiest way to implement a map is to view it as a list of entries
  - We can utilize the pre-existing List ADT
  - Our Map implementation creates and manipulates a List object
  - In order to manipulate entries, we must find them first, this is O(n)
- Array based implementation
  - An array of size N
  - Hashing function (shown as `h(k)` )used to map keys to integer values in the correct range
    - $0 \leq x < N$
    - Usually uses % N
  - We also need a collision handling strategy
  - This deals with the case where two keys have the same hash value

# Collision Handling Strategies

- There are two main types of collision handling strategies
  - Separate chaining
  - Open Addressing
- First we will study separate chaining

# Table of Contents

# Separate Chaining

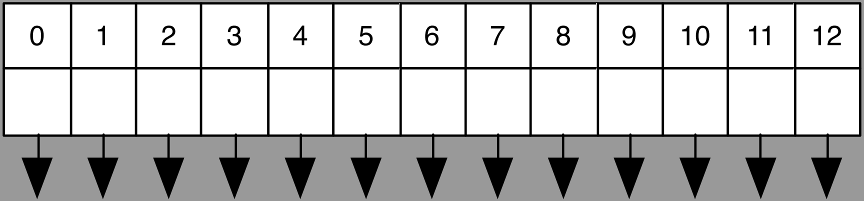
- In this basic strategy, entries with the same hash value are chained together
- This is usually implemented as an array of lists
- There will be one list for every index in the array
- Collisions are solved by adding a new entry to the end of the correct list

## Example

- We will use an array of size 13 to store elements in our hashmap
- The has map uses the following has function `h(x) = x mod 13`
- We will insert entries with the following keys
  `{ 18, 44,41, 22, 59, 32, 31, 73 }`
- In this example we will only show keys, to make it easier to understand

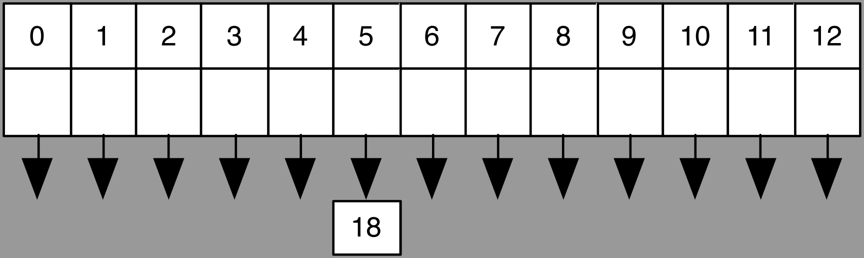
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $18 \bmod 13 = 5$, $44 \bmod 13 = 5$, $41 \bmod 13 = 2$, $22 \bmod 13 = 9$, $59 \bmod 13 = 7$, $32 \bmod 13 = 6$, $31 \bmod 13 = 5$, $73 \bmod 13 = 8$



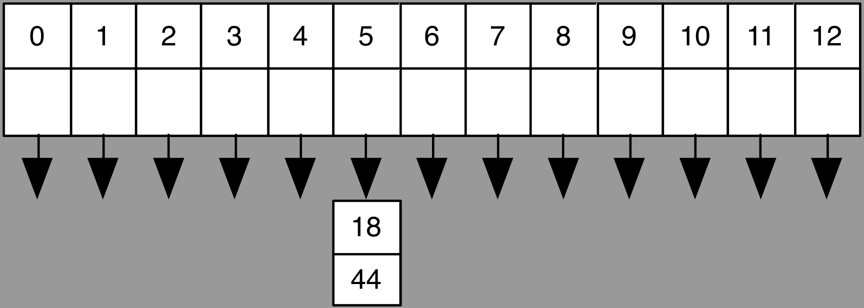
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 18 mod 13 = 5, 44 mod 13 = 5, 41 mod 13 = 2, 22 mod 13 = 9, 59 mod 13 = 7, 32 mod 13 = 6, 31 mod 13 = 5, 73 mod 13 = 8



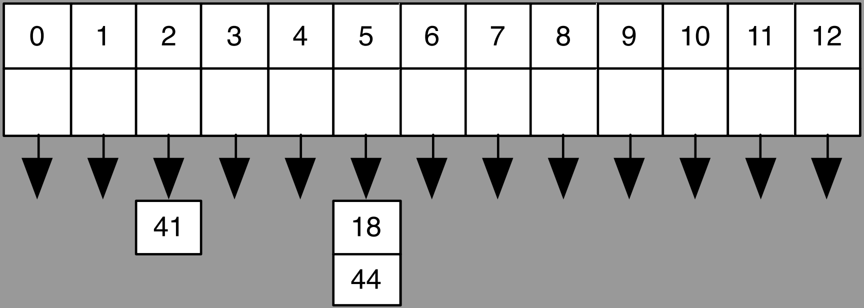
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 18 mod 13 = 5, 44 mod 13 = 5, 41 mod 13 = 2, 22 mod 13 = 9, 59 mod 13 = 7, 32 mod 13 = 6, 31 mod 13 = 5, 73 mod 13 = 8



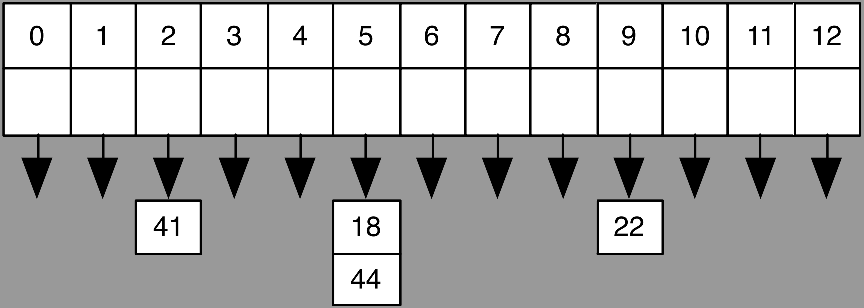
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 18 mod 13 = 5, 44 mod 13 = 5, 41 mod 13 = 2, 22 mod 13 = 9, 59 mod 13 = 7, 32 mod 13 = 6, 31 mod 13 = 5, 73 mod 13 = 8



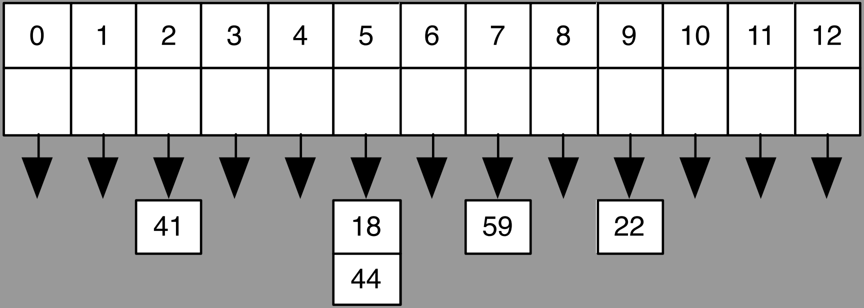
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 18 mod 13 = 5, 44 mod 13 = 5, 41 mod 13 = 2, 22 mod 13 = 9, 59 mod 13 = 7, 32 mod 13 = 6, 31 mod 13 = 5, 73 mod 13 = 8



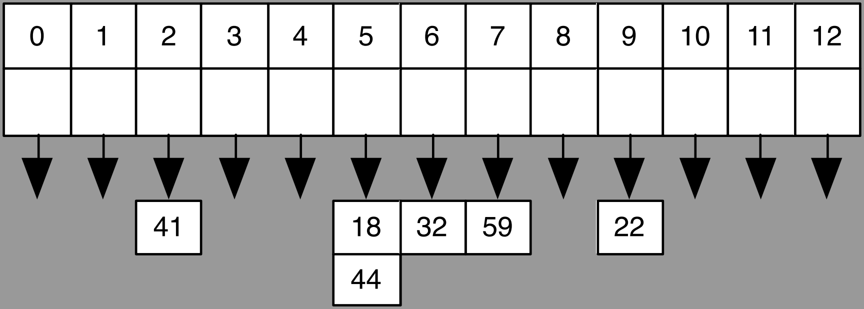
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $18 \bmod 13 = 5$, $44 \bmod 13 = 5$, $41 \bmod 13 = 2$, $22 \bmod 13 = 9$, $59 \bmod 13 = 7$, $32 \bmod 13 = 6$, $31 \bmod 13 = 5$, $73 \bmod 13 = 8$



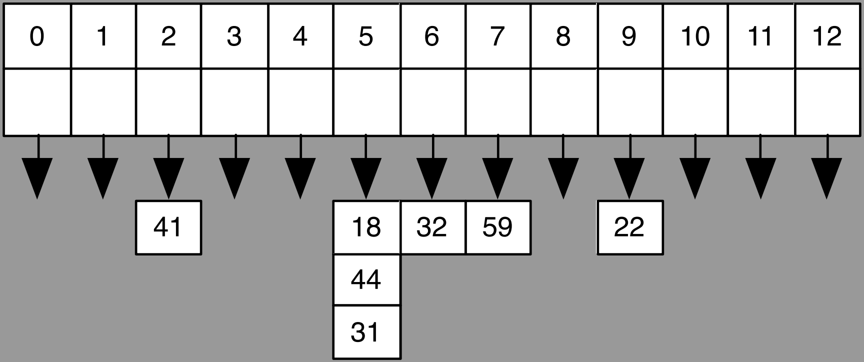
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $18 \bmod 13 = 5$, $44 \bmod 13 = 5$, $41 \bmod 13 = 2$, $22 \bmod 13 = 9$, $59 \bmod 13 = 7$, $32 \bmod 13 = 6$, $31 \bmod 13 = 5$, $73 \bmod 13 = 8$



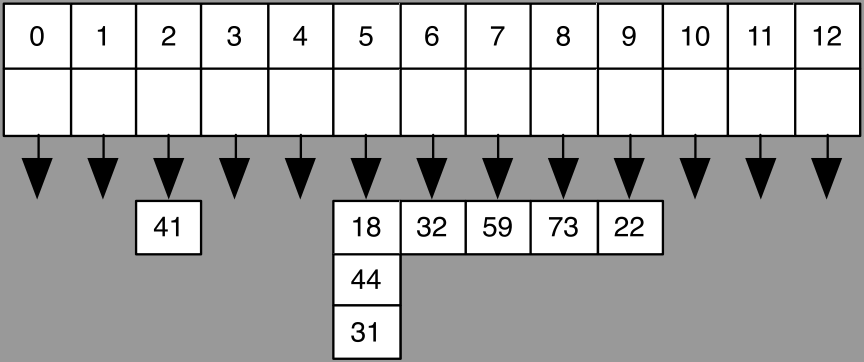
# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $18 \bmod 13 = 5$, $44 \bmod 13 = 5$, $41 \bmod 13 = 2$, $22 \bmod 13 = 9$, $59 \bmod 13 = 7$, $32 \bmod 13 = 6$, $31 \bmod 13 = 5$, $73 \bmod 13 = 8$



# Separate Chaining Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 18 mod 13 = 5, 44 mod 13 = 5, 41 mod 13 = 2, 22 mod 13 = 9, 59 mod 13 = 7, 32 mod 13 = 6, 31 mod 13 = 5, 73 mod 13 = 8



# Finding a Value

- Just like the list base implementation, we need to find the correct position in the list for each operation
- However, in this implementation we only need to search one of the lists in the array
- The list with the matching hashcode

# find(l, k)

- This operation takes two parameters
  - A list `l` we want to search
  - The key `k` we want to find the entry for
- Get the position p which is first in the list l
- Until p is equal to the last position in l
  - Compare the key in the entry in p with k
  - If they match return p
  - If they don't replace p with the position after p
- Compare the key in the entry p with k
  - If they match return p
  - If they don't return null

# find(l, v)

```
Algorithm find(l, k):
Input:    A key, k and a list l to be searched
Output:   The position, p, that entry with key k is
    stored in

p ← l.first()
last ← l.last()
while p <> last do
  if p.element().key() = k then
    return p
  else
    p ← l.after(p)
if NOT l.isEmpty() AND p.element().key() = k then
  return p
else
  return null
```

# get(k)

- Get the value associated with the key k and return it
  - Use the hash function to find the correct index x in the array
  - Find the position p in the list in index x in the array with the key k
  - If p is null, return p
  - If p is not null, return the value stored inside the entry in p

# get(k)

```
Algorithm get(k):
Input:      A key, k
Output:     The value, v, associated with k

x ← hashCode(k)
p ← find(lists[x], k)
if (p = null) then
  return null
return p.element().value()
```

# remove(k)

- Get the entry associated with the key k from the map and return the associated value
  - Use the hash function to find the correct index x in the array
  - Find the position p in the list in index x in the array with the key k
  - If p is null, return p
  - If p is not null
    - Remove p from the list in index x
    - Return the value stored inside the entry in p

# remove(k)

```
Algorithm remove(k):
Input:    A key, k
Output:   The value, v, associated with k

x ← hashCode(k)
p ← find(lists[x], k)
if (p = null) then
  return null
lists[x].remove(p)
return p.element().value()
```

# put(k, v)

- Update the map by adding a new entry with key k and value v, or update an existing entry by replacing the value with v
  - Use the hash function to find the correct index x in the array
  - Find the position p in the list in index x in the array with the key k
  - If p is null,
    - ⋆ Create a new entry, e, containing key k and value v
    - ⋆ Add e to the list in index x
    - ⋆ return null
  - If p is not null
    - ⋆ Create a new entry, e, containing key k and value v
    - ⋆ Add e to the list in index x, in the position after p
    - ⋆ Remove p from the list in index x
    - ⋆ Return the value stored inside the entry in p

# put(k,v)

```
Algorithm put(k, v):
Input:     A key, k and the value v associated
   with it
Output:    The value that was replaced or null

x ← hashCode(k)
p ← find(lists[x], k)
if (p = null) then
  create new entry e containing k and v
  lists[x].insertLast(e)
  return null
else
  create new entry e containing k and v
  lists[x].insertAfter(p, e)
  lists[x].remove(p)
  return p.element().value()
```

# Table of Contents

# Performance

- The performance of get, put and remove depend on the number of collisions
- In the best case, no collisions happen and the running time is O(1)
- In the worst case, every key has the same hash value and the running time is O(n)
- Normally, hash map performance is measured as expected running time
- In practice we try to achieve this by choosing a good hash function

| Operation | Expected Running Time |
|---|---|
| size | O(1) |
| isEmpty | O(1) |
| get | O(1) |
| put | O(1) |
| remove | O(1) |

# Hash Functions

- Hash functions convert keys to integer hash values in the range 0 to N - 1
  - Where N is the size of the array being used
- Any type of object can be a key
- To handle this hash functions must perform two basic mappings
  - Hash code map: assigns an integer value to each key
  - Compression map: converts an integer to an integer in the correct range

# Hash Functions

## The Division Method

- The previous example used a compression map known as the division method (% N)
- This is because our keys were already integers
- N should be a prime number
  - If it is not prime there will be more collisions
- We need to be careful of patterns in hash codes that form a pattern like $p * N + 1$
- Here many keys will map to q and there will be many collisions

# Hash Functions

## The MAD Method

- A better compression map is the Multiply Add and Divide method (MAD)
- The method takes the hash code and
  - Multiplies it by a constant value, known as the scale factor
  - Adds a second constant value, known as the shift
  - Returns the remainder when this value is divided by N

### MAD

For a given hash code, i, this method takes the form
$(a * i + b)\%N$
$a\%N$ should not equal 0

# Hash Code Maps

## Primitive Data Types

- Hash code map: assigns an integer value to each key
- There are several different types for primitive data types
- Integer cast: re-interpret the bits as an integer value
  - For example for a double, d, use (int) d
- Component sum: break the bits into integer sized blocks, cast each block as an integer and sum the values
  - For example for a long, l, $(int)(l >> 32) + (int)l$
- Polynomial sum: same as component sum, but multiply each term by a constant polynomial coefficient
  - For example for a sequence $S = c_0, c_1, c_2, ..., c_{n-1}$

$$h(s) = \sum_{i=0}^{n-1} c_i * p^i = c_0 + c_1 * p + c_2 * p^2 + ... + c_n - 1 * p^{n-1}$$

# Hash Code Maps

## Object Data Types

- For Objects, use the memory address or adapt one of the above based on the instance variables
- Has proven to be a simple but effective solution
- For Strings, a simple solution would be to use component sum
  - Strings are a sequence of characters represented by integers

### Component Sum Example

$$h(\text{“}dog\text{''}) = (int)\text{‘}d' + (int)\text{‘}o' + (int)\text{‘}g' = 100 + 111 + 103 = 314$$
$$h(\text{“}god\text{''}) = (int)\text{‘}g' + (int)\text{‘}o' + (int)\text{‘}d' = 103 + 111 + 100 = 314$$

- A better solution is to use polynomial sum

### Polynomial Sum Example (p = 3)

$$h(\text{“}dog\text{''}) = 103 + 111 * 3 + 100 * 9 = 1,336$$
$$h(\text{“}god\text{''}) = 100 + 111 * 3 + 103 * 9 = 1,360$$

# Table of Contents

# Collision Handling Strategies

- There are two main types of collision handling strategies
  - Separate chaining
  - Open Addressing
- Now we will study open addressing

# Separate Chaining

- Separate chaining uses an array of lists
- When a collision happens, the new entry is placed at the back of the list
- This offers infinite capacity
- However there are some drawbacks:
- It uses another data structure
- In practice the number of collisions increases as the number of entries increases

# Open Addressing

- Open addressing does not require any other data structures
- It has finite capacity, but we can support rehashing to extend the capacity
- We will study the linear probing form of open addressing

# Linear Probing

- For linear probing method of collision handling, we create an array of entries
  - Sometimes called a hash table
- The hash value h(k) can then be used as an index in this array
- If there is already an entry in this index a collision occurs
- We can resolve the collision by placing the entry in the next (circularly) available array index
- This is done by probing, consecutive positions in the array
- e.g. h(k)+1, h(k)+2,...

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$:

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

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $18 \bmod 13 = 5$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 18 | | | | | | | |

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $44 \bmod 13 = 5$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 18 | 44 | | | | | | |

Here there is a collision, so 44 must go in the next available index (6)

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $41 \bmod 13 = 2$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | | | | | | |

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $22 \bmod 13 = 9$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | | | 22 | | | |

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: h(x) = x mod 13: 59 mod 13 = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | 59 | | 22 | | | |

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $32 \bmod 13 = 6$

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

Here there is a collision, so 32 must go in the next available index (8)

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $31 \bmod 13 = 5$

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

Here there is a collision, so 31 must go in the next available index (10)

# Linear Probing Example

- Insert entries with these keys: 18, 44, 41, 22, 59, 32, 31, 73
- Hash Function: $h(x) = x \bmod 13$: $73 \bmod 13 = 8$

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

Here there is a collision, so 73 must go in the next available index (11)

# Getting Data

- The system of finding the next available index works when putting data in, but what about getting data out?
- If we want to get the value associated with the key 31, how do we do it?
  - First we check the correct index
  - If there is not entry there, the value is not in the hashmap
  - If there is an entry there, but the key does not match, we check the next index
  - We keep performing the same steps until we find the correct key, an empty space or come back around where we started

# Getting Data

## Finding 31

- $31 \% 13 = 5$
- Start Searching at 5

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

Key does not match 31, try next index (6)

# Getting Data

Finding 31

- $31 \ \% \ 13 = 5$
- Start Searching at 5

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

Key does not match 31, try next index (7)

# Getting Data

## Finding 31

- $31 \ \% \ 13 = 5$
- Start Searching at 5

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

Key does not match 31, try next index (8)

# Getting Data

## Finding 31

- 31 % 13 = 5
- Start Searching at 5

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

Key does not match 31, try next index (9)

# Getting Data

## Finding 31

- $31 \% 13 = 5$
- Start Searching at 5

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

Key does not match 31, try next index (10)

# Getting Data

## Finding 31

- 31 % 13 = 5
- Start Searching at 5

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

Key matches 31, return associated value

# Getting Data

Finding 35

- $35 \% 13 = 9$
- Start Searching at 9

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

Key does not match 35, try next index (10)

# Getting Data

## Finding 35

- $35 \% 13 = 9$
- Start Searching at 9

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

Key does not match 35, try next index (11)

# Getting Data

## Finding 35

- 35 % 13 = 9
- Start Searching at 9

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

Key does not match 35, try next index (12)

# Getting Data

## Finding 35

- $35 \ \% \ 13 = 9$
- Start Searching at 9

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

Index is empty, 35 is not here, return null

# Retrieval with Linear Probing

- First we use the hash function to get the correct index x in the array
- We probe consecutive locations until one of the following occurs
  - An item with key k is found
  - An empty cell is found
  - N indexes have been unsuccessfully probed

# get(k)

```
Algorithm get(k):
Input:     A key, k
Output:    The value, v, associated with k

i ← hashFunction(k)
p ← 0
repeat
  c ← A[i]
  if c = null then return null
  else if c.key() = k
    return c.value()
  else
    i ← (i + 1) mod N
  p ← p + 1
until p = N
return null
```

# Removal of Entries

- One problem we still have is how to remove entries
- Search is the key operation
- The current search algorithm terminates when a ‘gap’ is found
- If we simply remove entries, they will be replaced by ‘gaps’
- These ‘gaps’ would cause the search algorithm to stop
- To solve this a special object called AVAILABLE is used
  - Removed entries are replaced by the AVAILABLE token
  - A modified search algorithm could check whether each probe detects a valid entry of the token

# Remove 32

- When we remove 32, we replace it with the AVAILABLE token

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

# Remove 32

- When we remove 32, we replace it with the AVAILABLE token

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

# Get 31

- 31 % 13 = 5
- We start at index 5

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

Key does not match 31, try next index (6)

# Get 31

- 31 % 13 = 5
- We start at index 5

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

Key does not match 31, try next index (7)

# Get 31

- $31 \% 13 = 5$
- We start at index 5

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

Key does not match 31, try next index (8)

# Get 31

- 31 % 13 = 5
- We start at index 5

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

AVAILABLE token found, try next index (9)

# Get 31

- $31 \% 13 = 5$
- We start at index 5

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

Key does not match 31, try next index (10)

# Get 31

- $31 \ \% \ 13 = 5$
- We start at index 5

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

Key matches 31, return associated value

# put(k, v)

```
Algorithm put(k, v):
Input:    A key, k and a value v associated
   with it
Output:   The value that was replaced or null

h ←  hashFunction(k)
p ←  0
available ←  -1
while p < N do
  e ←  A[h]
  if e = null then
    if available > -1 then
      A[available] ←  new Entry(k, v)
      size ← size + 1
    else
      A[h] ←  new Entry(k, v)
```

# put(k, v)

```
      size ← size + 1

    return null
  if e = AVAILABLE AND available == -1 then
    available ← h
  else if e.key() = k then
    temp ← e.value()
    A[h] = new Entry(k, v)
    return temp

  h ← (h + 1) mod N
  p ← p + 1

if available > -1 AND A[available] ==
   AVAILABLE then
    A[available] ← new Entry(k, v)
    size ← size + 1
```

# remove(k)

```
Algorithm remove(k):
Input:    A key, k
Output:   The value, v, associated with k

h ← hashFunction(k)
p ← 0

while p < N do
  e ← A[h]
  if e = null then return null
  if e.key() = k then
    temp ← e.value()
    A[h] ← AVAILABLE
    size ← size − 1
    return temp

  h ← (h + 1) mod N
  p ← p + 1
return null
```

# Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take O(n) time
- This occurs when all the keys inserted into the map collide
- The load factor $a = n/N$ also affects the performance of a hash table
- Assuming hash values are like random numbers, the expected number of probes for (open addressing) insertion is: $1/(1 - a)$
- The expected running time of all the map ADT operations in a hash table is O(1)
- In practice, hashing is very fast provided the load factor is not close to 100%

# Rehashing

- Rehashing is the process of expanding the capacity of a hash table
- It's a lot like an extendible array (I.e. ArrayList)
- Rehashing is performed when the load factor moves above a certain threshold.
- We rehash by
  - Creating a new array ($> 2N$ in size)
  - Specifying a new compression map (e.g. update the division method to work with the new size)
  - Inserting each entry into the new array.
- Given insertion is O(1), rehashing is an O(N) operation:

# Further Information and Review

If you wish to review the materials covered in this lecture or get further information, read the following sections in Data Structures and Algorithms textbook.

- 9.1 - Maps
- 9.2 - Hash Tables