Software Development (cs2500)

Lectures 48 & 49: The Collections Framework

M. R. C. van Dongen

February 7, 2014

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

For Wednesday

Acknowledgements

About this Document

Outline

- Study the `Java` Collection Framework.
- Finalise our in-depth study of linked lists.
- Explore the `Set` interface.
- Use the `Map` interface to implement lookup tables.

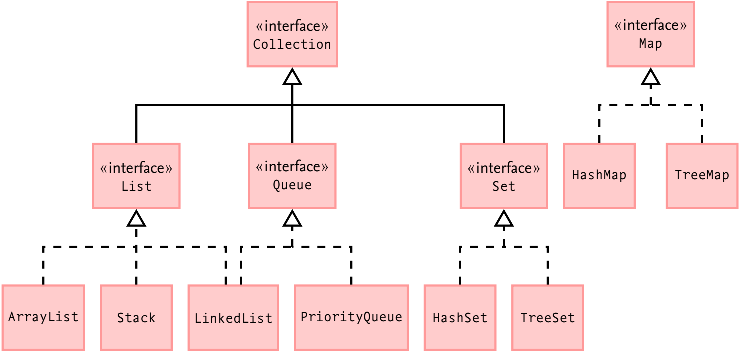
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Java Collections Framework

- Framework for storing items, querying, and traversing.
- All interfaces and classes are generic.
- At the top are the `Collection` and `Map` interfaces.
  - A `Collection<T>` is for storing/retrieving `T`s.
  - A `Map<K,V>` is for looking up value `V` using key `K`.
  - In some sense, a `Collection<T>` is a `Map<T,T>`.

Collections Hierarchy

All Interfaces/Classes are Generic



`Collection<T>` API

`size( )`	Determine the size of the collection.
`add( T item )`	Add a member to the collection.
`toString( )`	Compute `String` of what's in the collection.
`remove( T item )`	Tentatively remove `item` from collection. - Returns `true` if and only if successful.
`contains( T item )`	Detemine whether `item` is in the collection. - Returns `true` if and only `item` is in it.
`iterator( )`	Returns an `Iterator<T>`. - Defined in `Iterable<T>` interface. - Used to traverse the collection. - If a class implements `Iterable<T>`, you can use enhanced `for` notation.

Implementing `remove( )`

Java

```
public class MyList<T> {
    private Link<T> nodes;

    ...

    private static class Link<T> {
        private T head;
        private Link<T> tail;

        ...
    }
}
```

Implementing `remove( )` (Continued)

Worst-Case Performance Proportional to Length of List

Java

```java
public boolean remove( T item ) {
    final boolean result;

    if (nodes == null) {
        result = false;
    } else if (nodes.head.equals( item )) {
        result = true;
        nodes = nodes.tail;
    } else {
        Link<T> link = nodes;
        while ((link.tail != null) && (!link.tail.head.equals( item ))) {
            link = link.tail;
        }
        result = link.tail != null;
        if (result) {
            link.tail = link.tail.tail;
        }
    }

    return result;
}
```

`Iterator<T>` and `Iterable<T>`

`Iterator` Used to traverse collection.
Implementing classes must override:

- `hasNext( )` Determine whether there's another member.
- `next( )` Get the next member.
- `remove( )` Remove most recent `next( )` value.
  - Implementing this method is optional.

`Iterable` Implementing classes must override `iterator( )`:

- `public Iterator<T> iterator( )`
- Traverse `Iterable` classes with enhanced `for`:
  - ```
    for (T item : collection) {
       /* use item */
    }
    ```


Software Development
M. R. C. van Dongen
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document
    ```

Implementing `Iterable`

Java

```
  @Override
  public Iterator<T> iterator( ) {
      return new Iterator<T>( ) {
          private Link<T> current = nodes;

          @Override
          public boolean hasNext( ) {
              return current != null;
          }

          @Override
          public T next( ) {
              final T next = current.head;
              current = current.tail;
              return next;
          }

          @Override
          public void remove( ) {
          }
      };
}
```


Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Singly Linked Lists

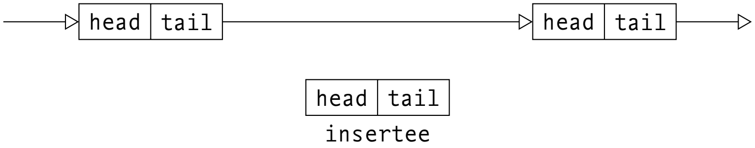
- So far we've implemented singly linked lists.
- At the top level a `Link` instance, `nodes`, represents the members.
- Inserting a new `Link` depends on two cases:

`nodes == null` No need to locate insertion position:

1. Create the new `Link`.
2. Assign `Link` to `nodes`.

`nodes != null` We must locate insertion position:

1. Create the new `Link`, `insertee`.
2. Find `Link`, `position`, to insert the `Link`.
3. Assign `position.tail` to `insertee.tail`.
4. Assign `insertee` to `position.tail`.



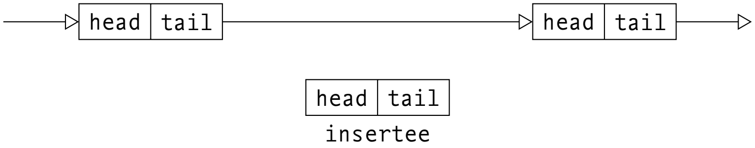
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Singly Linked Lists

- So far we've implemented singly linked lists.
- At the top level a `Link` instance, `nodes`, represents the members.
- Inserting a new `Link` depends on two cases:

  `nodes == null` No need to locate insertion position:
  1. Create the new `Link`.
  2. Assign `Link` to `nodes`.

  `nodes != null` We must locate insertion position:
  1. Create the new `Link`, `insertee`.
  2. Find `Link`, `position`, to insert the `Link`.
  3. Assign `position.tail` to `insertee.tail`.
  4. Assign `insertee` to `position.tail`.



Singly Linked Lists

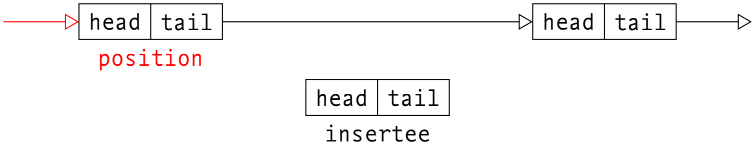
- So far we've implemented singly linked lists.
- At the top level a `Link` instance, `nodes`, represents the members.
- Inserting a new `Link` depends on two cases:

`nodes == null` No need to locate insertion position:

1. Create the new `Link`.
2. Assign `Link` to `nodes`.

`nodes != null` We must locate insertion position:

1. Create the new `Link`, `insertee`.
2. Find `Link`, `position`, to insert the `Link`.
3. Assign `position.tail` to `insertee.tail`.
4. Assign `insertee` to `position.tail`.



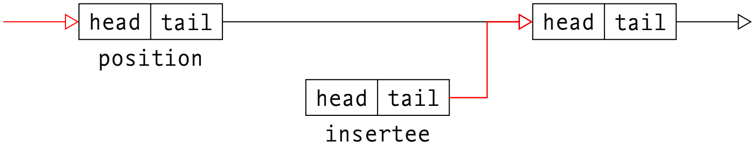
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Singly Linked Lists

- So far we've implemented singly linked lists.
- At the top level a `Link` instance, `nodes`, represents the members.
- Inserting a new `Link` depends on two cases:

  `nodes == null` No need to locate insertion position:
  1. Create the new `Link`.
  2. Assign `Link` to `nodes`.

  `nodes != null` We must locate insertion position:
  1. Create the new `Link`, `insertee`.
  2. Find `Link`, `position`, to insert the `Link`.
  3. Assign `position.tail` to `insertee.tail`.
  4. Assign `insertee` to `position.tail`.



Singly Linked Lists

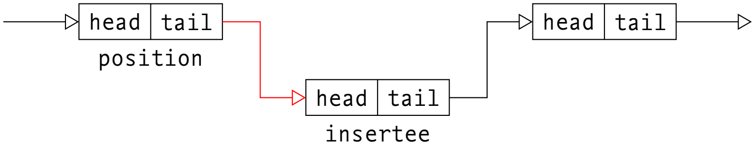
- So far we've implemented singly linked lists.
- At the top level a `Link` instance, `nodes`, represents the members.
- Inserting a new `Link` depends on two cases:

`nodes == null` No need to locate insertion position:

1. Create the new `Link`.
2. Assign `Link` to `nodes`.

`nodes != null` We must locate insertion position:

1. Create the new `Link`, `insertee`.
2. Find `Link`, `position`, to insert the `Link`.
3. Assign `position.tail` to `insertee.tail`.
4. Assign `insertee` to `position.tail`.



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Singly Linked Lists

Analysis

- Modern CPUs try to predict next instruction.
  - With branching, this prediction becomes difficult.
  - Inserting an item needs branching, so it's not ideal.
- Inserting at start of list is constant time operation.
  - Inserting at end requires linear time (in length of list).

Enters the Doubly Linked List

- A *doubly linked list* overcomes the singly linked list's problems.
- It requires no branching when inserting a new `Link`.
  - (Provided the insertion position is known.)
- Also supports constant time opration for appending items.
- Allows removal of `Link` in constant time.
  - (Provided a reference to `Link` is known.)
- Can be implemented with `addAfter( )` and `addBefore( )`.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

The Data Structure

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Java

```
public class DList<T> {
    private DLink<T> sentinel;

    public DList( ) {
        sentinel = new DLink( );
    }

    ...

    private static class DLink<T> {
        private T data;
        private DLink<T> prev;
        private DLink<T> next;

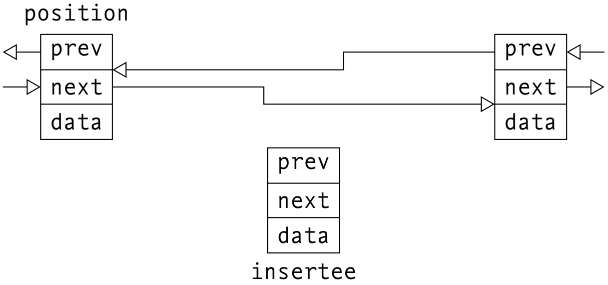
        private DLink( ) {
            prev = next = this;
        }
    }
}
```

The addAfter( ) Operation

AddBefore( ) is Similar

Java

```java
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```

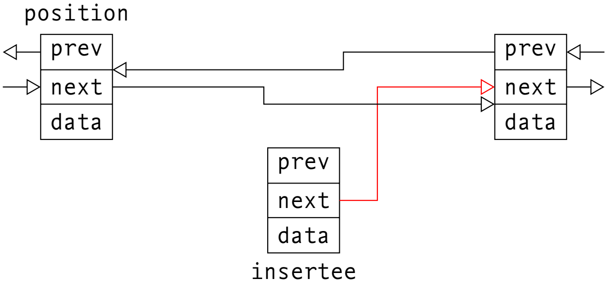


The addAfter( ) Operation

AddBefore( ) is Similar

Java

```
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```

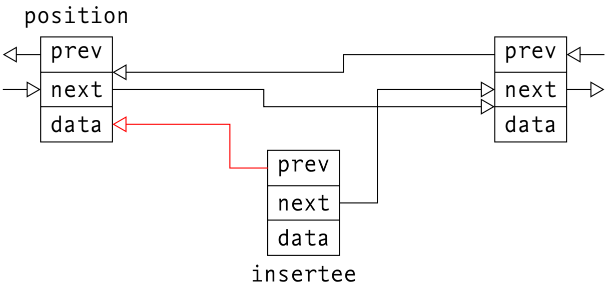


The `addAfter( )` Operation

`AddBefore( )` is Similar

Java

```
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```



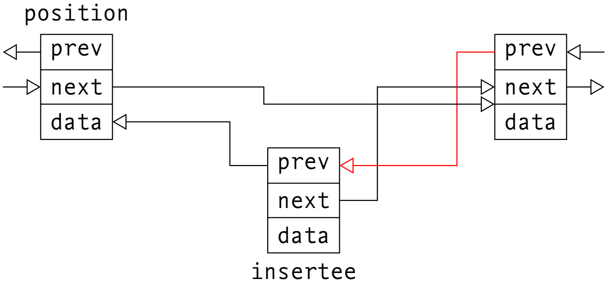
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

The `addAfter( )` Operation

`AddBefore( )` is Similar

Java

```
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```



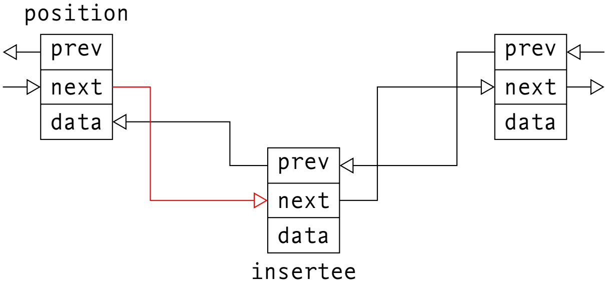
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

The `addAfter( )` Operation

`AddBefore( )` is Similar

Java

```
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```

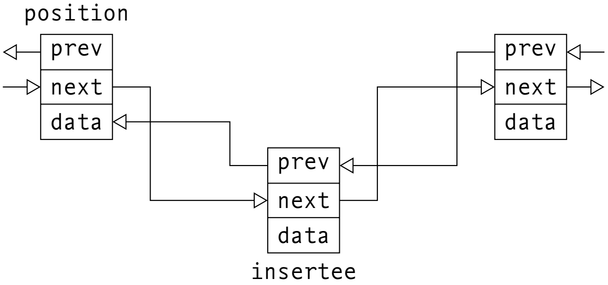


The addAfter( ) Operation

AddBefore( ) is Similar

Java

```
private static void addAfter( final DLink<T> position, final DLink<T> insertee ) {
    insertee.next = position.next;
    insertee.prev = position;
    position.next.prev = insertee;
    position.next = insertee;
}
```



Introduction

- The `Set` interface is for collections without repetitions.
- The order of traversal depends on the implementing class.

`HashSet` No “real” order.

  - Uses `item.hashCode( )` for `add( item )` and `remove( item )`.
  - Good average performance for `add( )` and `remove( )` if there are few `hashCode( )` collisions.

`TreeSet` Set with order.

  - Uses tree to represent the set.
  - Order determined by `compareTo( )`.
  - Usually poorer performance for `add( )` & `remove( )`.

Introduction to `HashSets`

- `HashSet` usually provides good average performance.
- Uses `hashCode( )` to partition members in to subcollections.
- Members with “similar” `hashCode( )` go to same subcollection.
  - Equality in subcollections is decided with `equals( )`.
  - Classes usually override `equals( )`.
- Good performance guaranteed if subcollections don’t “overflow.”
- Diversifying the `hashCode( )` values makes “overflow” less likely.

Contract

- Deciding deep equality is an expensive operation.
- The `HashSet` class uses `equals( )` to decide equality.
- If you override `equals( )` you must override `hashCode( )`.
- All `Java` classes make assumptions about `hashCode( )`:
  - The same object must always return the same `hashCode( )`.
  - Two deeply equal objects must have the same `hashCode( )`.
  - (Different objects may have the same `hashCode( )`.)
- With these assumptions, we can decide deep equality as follows:
  - If `a.hashCode( ) != b.hashCode( )` then `!a.equals( b );`
  - Else use `a.equals( b )`.
- This is usually very efficient, provided
  - `hashCode( )` is fast; and
  - `hashCode( )` depends on as many relevant attributes as possible.

Computing Hash Codes

- By default different objects have different `hashCode( )` values.
- This is usually not useful, so we must override it.
- Take as much information into account as possible.
- “Information” are the attributes that determine `equals( )`.
- This generally reduces collisions.

Computing Hash Codes (Continued)

Overall Method

1. Initialise `result` with some nonzero value;
2. For each relevant attribute:
   1. Compute an `int`, `intCode`, for the attribute;
   2. Combine the the current value of `result` and `intCode`:
      - `result = result * 31 + intCode`.
3. Return `result`.

Computing Hash Codes (Continued)

Computing `intCode` of Attribute, `attr`

- If attribute is `boolean` return `attr ? 1 : 0;`
- If it's `byte`, `char`, or `short`, return `(int)attr;`
- If it's an `int` return `attr;`
- If it's a `long` return `attr ^ (attr >>> 32);`
- If it's a `float`, return `Float.floatToIntBits( attr );`
- If it's a `double`, return `Double.doubleToIntBits( attr );`
- If it's `null` return 0;
- If it's an array:
  - If all values are relevant, return `Arrays.hashCode( attr )`.
  - Recursively compute a hash code for the relevant values.
- Otherwise, return `attr.hashCode( )`.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Example

Java

```
public class Person {
    final static int INITIAL_HASH_CODE = 17;
    final static int HASH_MULTIPLIER = 31;
    private final String name;
    private final boolean isMale;
    private final int age;

    ...

    @Override
    public boolean equals( Object that ) {
        final Person p = (Person)that;
        return (name.equals( p.name )) && (age == p.age);
    }

    @Override
    public int hashCode( ) {
        int result = INITIAL_HASH_CODE;
        result = result * HASH_MULTIPLIER + age;
        result = result * HASH_MULTIPLIER + (isMale ? 1 : 0);
        result = result * HASH_MULTIPLIER + name.hashCode( );
        return result;
    }
}
```

Using the Person Class

Java

```
// Always use an interface variable to reference the set.
final Set<Person> set = new HashSet<Person>( );
set.add( new Person( "Joe", true, 20 ) );
set.add( new Person( "Jane", false, 20 ) );
...
```

When Computations get Expensive

Don't Try This at Home

```
public class Casher {
    ...

    public Casher( ... ) {
        // initialise attribute values.
    }

    @Override
    public int hashCode( ) {
        return motherOfAllHashCodeComputations( );
    }
}
```


Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

To Avoid Recomputing, you can Cache the Value

Java

```
public class Casher {
    private static final hashCode;
    ...

    public Casher( ... ) {
        // first initialise attribute values.
        ...
        hashCode = motherOfAllHashCodeComputations( );
    }

    @Override
    public int hashCode( ) {
        return hashCode( );
    }
}
```

Lazy Computation

Code is Correct for Single-Threaded Classes

- Initialising many hash code values may take too long.
- When too many hash codes are initialised at the same time, this may cause unacceptable delays.
- You can postpone the expensive initialisation.
  - Makes sense if you don't need the hash code values.
- This is called *lazy initialisation*.
  - Intelligent classes such as `String` use lazy initialisation.

Java

```
public class Lazy {
    private Integer hashCode;

    ...

    @Override
    public int hashCode( ) {
        if (hashCode == null) {
            hashCode = motherOfAllInitialHashCodeComputations( );
        }
        return hashCode;
    }
}
```


Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

Maps

- A `Map` is like a mathematical function.
- It maps a *key* to a *value*.
- The `Map` interface is generic: `Map<K,V>`.
- A given key can only have no more than one value.
- Some keys have no values: their values are `null`.
- There are several `Map` implementations:
  - `HashMap`.
  - `TreeMap`.

Instance Methods

`put( key, value )`	Add `value` for `key`. - Returns current value if present; - Otherwise `null`.
`get( key )`	Get value of `key`; `null` if not present.
`keySet( )`	Get a set representation of the keys.

Example

Java

```
final Map<Color,String> map = HashMap<Color,String>( );
map.put( Color.RED, "Bad" );
map.put( Color.Green, "Good" );
map.put( Color.Blue, "Cold" );
```


Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

For Wednesday

Acknowledgements

About this Document

Introduction

- The `Queue` class implements a first-in-first-out (FIFO) collection.
- The `PriorityQueue` class implements a *priority queue*.
    - The elements in the collection are ordered.
    - Order depends on `compareTo( )`.
    - Least significant instances have higher *priority*.
    - The `remove( )` method removes instance with highest priority.

Example

Java

```java
public class MyProcess implements Comparable<MyProcess> {
    private int priority;
    private final Task task;

    public static void main( String[] args ) {
        final PriorityQueue<MyProcess> tasks = new PriorityQueue( );
        tasks.add( ... );
        ...
        while (!tasks.isEmpty( )) {
            final Task task = queue.remove( );
            task.run( );
        }
    }

    ...
    @Override
    public int compareTo( MyProcess that ) {
        return (this.priority < that.priority) ? -1 :
               (this.priority > that.priority) ?  1 : 0;
    }
}
```

Outline

- Hashing maps keys to positions in arrays.
- *Perfect hashing* maps different keys to different positions.
- Perfect hashing is rarely possible:
  - In general *collisions* will occur.
  - Should they occur, these collisions should be *resolved*.
- We shall study two classes of collision resolution strategies.
  - Open addressing: here we look for other free cells.
  - Separate chaining: stores colliding keys in a special data structure.

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Implementing a `Table` Class

- Arrays are the basic data structure for many applications.
- However, storing and retrieving by index isn't always ideal.
- E.g., let's assume we want to implement a `Table` (`Map`) class.
- Each item in the `Table` has a *key*.
- Different items have different keys.
- The `Table` supports the following methods.

`Table( int size ):`	Create an *empty* `Table` object.
`isFull( ):`	Determine if the `Table` is full.
`add( Object o ):`	Add a new item to the `Table`.
`remove( Object o ):`	Remove existing item from the `Table`.
`contains( Object o ):`	Decide if the `Table` contains `o`.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Implementing a `Table` Class

- Arrays are the basic data structure for many applications.
- However, storing and retrieving by index isn't always ideal.
- E.g., let's assume we want to implement a `Table` (`Map`) class.
- Each item in the `Table` has a *key*.
- Different items have different keys.
- The `Table` supports the following methods.

`Table( int size ):`	Create an *empty* `Table` object.
`isFull( ):`	Determine if the `Table` is full.
`add( Object o ):`	Add a new item to the `Table`.
`remove( Object o ):`	Remove existing item from the `Table`.
`contains( Object o ):`	Decide if the `Table` contains `o`.

- All operations should be fast.
- We shall use an array to store each object in the `Table`.
- We use the keys to determine the object's positions.
- How do we map the keys to index positions in the array?

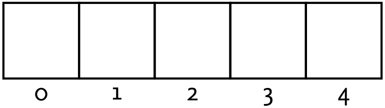
Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

First Attempt: Use hashCode( )

Don't Try This at Home

```
public class Table {
    private final Object[] members;

    public Table( int size ) {
        members = new Object[ size ];
    }

    private int hashFunction( Object o ) {
        return Math.abs( o.hashCode( ) ) % members.length;
    }

    public void add( Object o ) {
        members[ hashFunction( o ) ] = o;
    }

    public void remove( Object o ) {
        members[ hashFunction( o ) ] = null;
    }

    public boolean contains( Object o ) {
        return (o != null)
                  && (o == members[ hashFunction( o ) ]);
    }
}
```

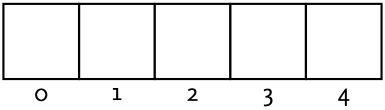
Let's See



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let's See

Add Object with HashCode 123



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let’s See

Add Object with HashCode 123

0	1	2	3	4
			123	

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let's See

Add Object with HashCode 20

			123	
0	1	2	3	4

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let's See

Add Object with HashCode 20

20			123	
0	1	2	3	4

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let’s See

Add Object with HashCode 666

20			123	
0	1	2	3	4

Let’s See

Add Object with HashCode 666

20	666		123	
0	1	2	3	4

Let's See

Add Object with HashCode 33

20	666		123	
0	1	2	3	4

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let's See

Add Object with HashCode 33

0	1	2	3	4
20	666		123 33	

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Let’s See

Add Object with HashCode 33

20	666		123 33	
0	1	2	3	4

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Let’s See

You Caaan’t Do That

0	1	2	3	4
20	666		123 33	

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Hash Functions

Definition (Hash Function)

- A *hash function* is a function that maps its argument to an index position of an array.

Java

```
public class IntTable {
    final int[] members;

    private int hashFunction( int key ) {
        return Math.abs( key ) % members.length;
    }
    ...
}
```


Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Ideal Hash Function Properties

- Good hash functions should be correct:
  - The result should be an allowed index.
- They should be fast.
- They should minimise collisions.
- Random keys should result in random index positions.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

More Complications

- Let's assume we have three objects.
- The hash codes of the objects are 0, 1, and 5.
- Let's also assume we have two `Table` objects.
- `Table` 1 has a capacity of 3: its hash function is $h_3(c) = c \,\%\, 3$.
- `Table` 2 has a capacity of 4: its hash function is $h_4(c) = c \,\%\, 4$.
- With the first hash function we get no collisions:
  - $h_3(0) = 0$, $h_3(1) = 1$, and $h_3(5) = 2$.
- With the second hash function we get a collision:
  - $h_4(0) = 0$, $h_4(1) = 1$, and $h_4(5) = 1$.
- So, how do we fix this problem?

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Different Hash Functions

- Possible solution:
  - Let each `Table` object *compute* its own hash function.
- They start with a random hash function.
- They use it until a collision arises.
- When the collision arises:
  - The `Table` computes a new random hash function.
  - The new hash function should resolve the colission.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

A Family of 2-Universal Hash Functions

- Assume the hash codes are in the domain $U = \{0, \ldots, m-1\}$.
- Furthermore, assume our array has length $n \leq m$.
- Let $p \geq m$ be a random prime.
- Finally, let $0 < a < p$ and $0 \leq b \leq p$ be random integers.
- Consider the following hash function:

$$h_{a,b}(x) = ((ax + b) \,\%\, p) \,\%\, n \,.$$

- You can prove that if x and y are random members from U, then

$$\mathbb{P}\left(h_{a,b}(x) = h_{a,b}(y)\right) \leq 1/n \,. \tag{1}$$

- Mitzenmacher, and Upfal [2005, Lemma 13.6] prove (1).

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Improving our Hash Functions

Java

```java
public class IntTable {
    final Random rand;
    int[] array;
    int a, b, m, p;

    public IntTable( int size, int m ) {
        array = new int[ size ];
        rand  = new Random( );
        this.m = m;
        computeNewHashFunctionConstants( );
    }

    private void computeNewHashFunctionConstants( ) {
        p = randomPrime( m );
        a = 1 + rand.nextInt( p - 2 );
        b = rand.nextInt( p + 1 );
    }

    // ASSUMPTION 0 <= x < m.
    private int hashFunction( int x ) {
        final int n = array.length;
        return ((a * x + b) % p) % n;
    }
    ...
}
```


Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Collision Resolution

- A *perfect hash function* maps each key to a unique index.
- Non-perfect hash function may result in collisions.
  - They cannot be avoided.
- If a collision occurs when adding a key,
  - We must *resolve the collision*.
- There are two techniques:

  Open addressing: Use a different, free index.

  Buckets: Allow multiple keys per index.

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Running Example (Start)

Letters Left: B_2, J_{10}, S_{19}

0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

`hashCode( )`

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Running Example (Start)

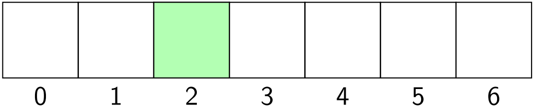
Letters Left: B_2, J_{10}, S_{19}

0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Running Example (Start)

Letters Left: B_2, J_{10}, S_{19}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Running Example (Start)

Letters Left: B_2, J_{10}, S_{19}

		B_2				
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Running Example (Start)

Letters Left: J_{10}, S_{19}

0	1	2	3	4	5	6
		B_2				

Running Example (Start)

Letters Left: J_{10}, S_{19}

0	1	2	3	4	5	6
		B_2				

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Running Example (Start)

Letters Left: J_{10}, S_{19}

0	1	2	3	4	5	6
		B_2	J_{10}			

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Running Example (Start)

Letters Left: S_{19}

		B_2	J_{10}			
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Running Example (Start)

Letters Left: S_{19}

		B_2	J_{10}			
0	1	2	3	4	5	6

Running Example (Start)

Letters Left: S_{19}

		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Running Example (Start)

Letters Left:

		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Open Addressing with Linear Probing

- Let $n > 2$ be a prime and let's assume we have an n-sized array that isn't full.
- Furthermore, let's assume we want to insert a key, k.
- However, this time a collision occurs.
- To resolve the collision we need to find a free cell.
- We start at $h(k)$, next visit $(h(k) - 1) \% n$, visit $(h(k) - 2) \% n$, ...
- Eventually, we should find some free cell.
- This collision resolution policy is called *linear probing*.

Example

Letters Left: N_{14}, X_{24}, W_{23}

0	1	2	3	4	5	6
		B_2	J_{10}		S_{19}	

Example

Letters Left: N_{14}, X_{24}, W_{23}

		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: N_{14}, X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Example

Letters Left: N_{14}, X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Example

Letters Left: X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Example

Letters Left: X_{24}, W_{23}

N_{14}		B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Example

Letters Left: X_{24}, W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_2	J_{10}		S_{19}	
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: W_{23}

N_{14}	X_{24}	B_{2}	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left:

N_{14}	X_{24}	B_2	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Probe Sequences

- Linear probing visits a sequence of occupied and free indices.
- The sequence of occupied cells is called the *probe sequence*.
- The key, k, and the hash function determine the sequence's first index: $h(k)$.
- The remaining indices are determined by the collision resolution policy.
- The ith next index is $(h(k) - i \times p(k)) \,\%\, n$,
  - Where $p(\cdot)$ is the *probe decrement function*.
- For linear probing the probe decrement function is given by $p(k) = 1$.

Removing an Item:

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}	B_2	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Removing an Item: Remove B_2

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}	B_2	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Removing an Item: Remove B_2

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}		J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Removing an Item: Locate X_{24}

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}		J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Removing an Item: Locate X_{24}

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}		J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Removing an Item: Locate X_{24}

- When removing a value we shouldn't break a probe sequence.
- E.g. let's assume we remove B_2 by making Cell 2 empty.
- If make it empty we'll have problems locating X_{24}:
  - A free cell in the sequence should indicate that B_2 isn't in the table.

N_{14}	X_{24}		J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Removing an Item:

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	B_2	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Removing an Item: Remove B_2

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	B_2	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Removing an Item: Remove B_2

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	$\perp$	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Removing an Item: Locate X_{24}

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	$\perp$	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Removing an Item: Locate X_{24}

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	$\perp$	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Removing an Item: Locate X_{24}

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	$\perp$	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Removing an Item: Locate X_{24}

- Instead of making B_2's cell available, we mark it $\perp$.
  - When looking for items we treat $\perp$ as a used cell.
  - When looking for free cells we treat $\perp$ as a free cell.

N_{14}	X_{24}	$\perp$	J_{10}		S_{19}	W_{23}
0	1	2	3	4	5	6

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Linear Probing Revisited

- Linear probing is not a good collision resolution policy.
- It tends to lead to clusters, which in their turn lead to bigger clusters, ...
- With large clusters you get long probe sequences.
- The larger the clusters get the faster they grow.
- In general it is better to have *different* probe decrements for different keys.
- This is called *double hashing*.

Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Double Hashing

- With double hashing we have two hash functions.
  - $h_1(k)$ should be random index.
  - $h_2(k)$ should be a random probe decrement as a function of k.
- We use the two to compute the probe sequence (modulo n):

$$h_1(k), h_1(k) - h_2(k), h_1(k) - 2h_2(k), h_1(k) - 3h_2(k), \ldots .$$

- Square hopping should still work with $h_2(\cdot)$.
  - Therefore $h_2(k)$ should be *relative prime* to n.
- Usually, n is a power of two.
- If n is a power of two ands $h_2(k)$ is odd,
  - Then $h_2(k)$ and n are relative prime [Cormen et al. 2001].

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

About this Document

Limitations of Open Addressing

- Frequent additions cause clustering.
- Frequent additions will eventually fill the table.
- Both problems may be overcome by resizing the table.
- `Java` has many classes based on hashing.
- Many re-size themselves so as to ensure good performance.

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

For Wednesday

Acknowledgements

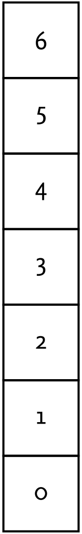
About this Document

Separate Chaining

- Another collision resolution strategy is *separate chaining*.
- It changes the structure of the hash table.
- Table locations now store *multiple* values.
- Each non-empty table index now has a *bucket*.
- Initially all buckets are empty.
- When a key is mapped to a location,
  - We add it to the location's bucket.
- A possible implementation for the bucket is a linked list.
- Separate chaining provides a simple way to resolve collisions.
- However, it requires more memory than open addressing.

Example

Letters Left: B_2,J_{10}, S_{19},N_{14},X_{24},W_{23}



Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

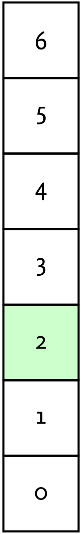
For Wednesday

Acknowledgements

About this Document

Example

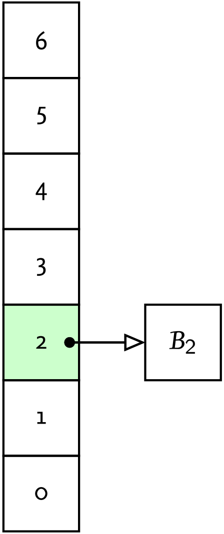
Letters Left: B_2, J_{10}, S_{19}, N_{14}, X_{24}, W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

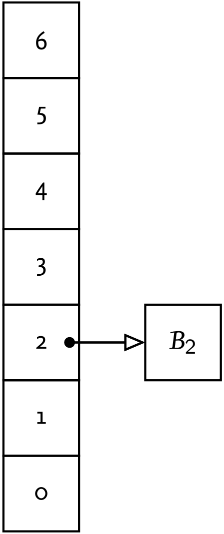
Example

Letters Left: B_2,J_{10}, S_{19},N_{14},X_{24},W_{23}



Example

Letters Left: B_2,J_{10}, S_{19},N_{14},X_{24},W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

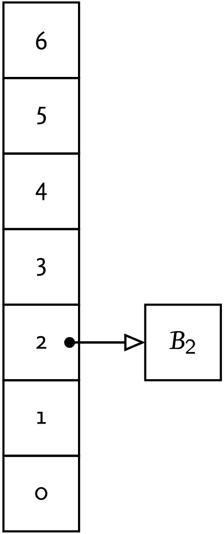
For Wednesday

Acknowledgements

About this Document

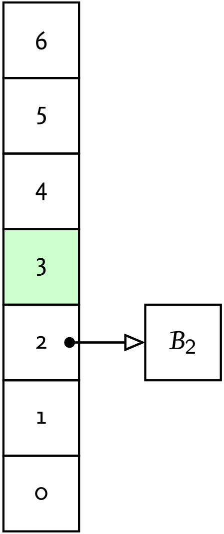
Example

Letters Left: J_{10}, S_{19},N_{14},X_{24},W_{23}



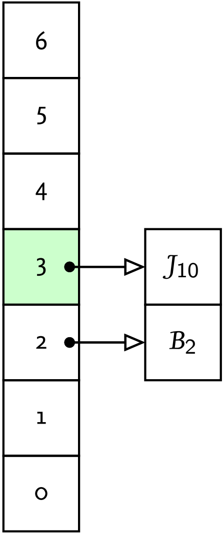
Example

Letters Left: J_{10}, S_{19},N_{14},X_{24},W_{23}



Example

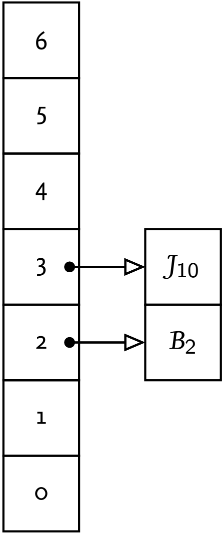
Letters Left: J_{10}, S_{19}, N_{14}, X_{24}, W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: J_{10}, S_{19},N_{14},X_{24},W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

`hashCode( )`

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

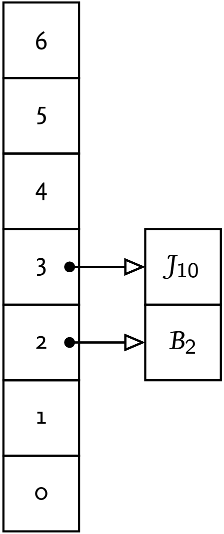
For Wednesday

Acknowledgements

About this Document

Example

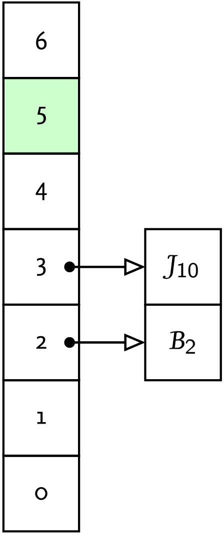
Letters Left: $S_{19}, N_{14}, X_{24}, W_{23}$



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

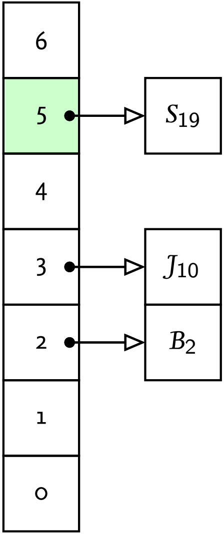
Letters Left: S_{19},N_{14},X_{24},W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: S_{19},N_{14},X_{24},W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

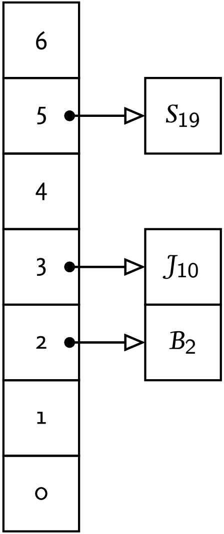
For Wednesday

Acknowledgements

About this Document

Example

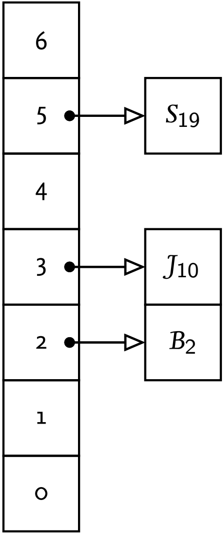
Letters Left: $S_{19}, N_{14}, X_{24}, W_{23}$



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: N_{14}, X_{24}, W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

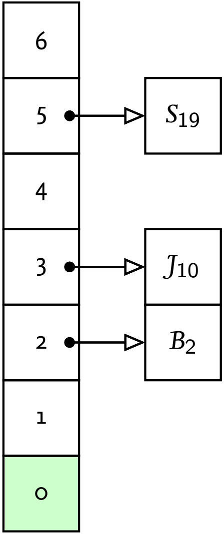
For Wednesday

Acknowledgements

About this Document

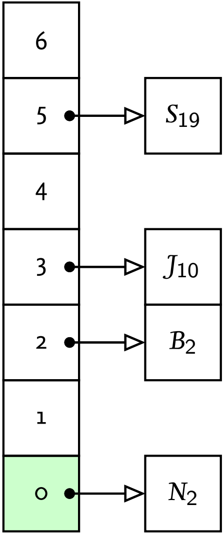
Example

Letters Left: N_{14},X_{24},W_{23}



Example

Letters Left: N_{14}, X_{24}, W_{23}



Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

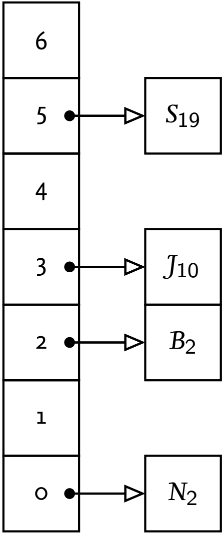
For Wednesday

Acknowledgements

About this Document

Example

Letters Left: N_{14}, X_{24}, W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

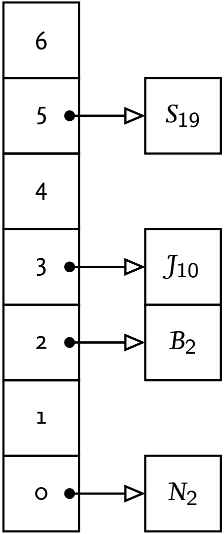
For Wednesday

Acknowledgements

About this Document

Example

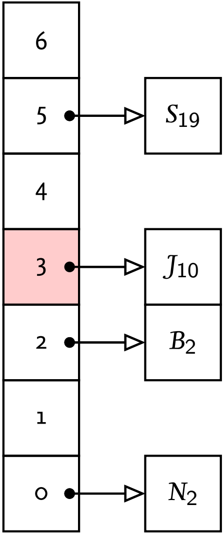
Letters Left: X_{24}, W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

Example

Letters Left: X_{24}, W_{23}



Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

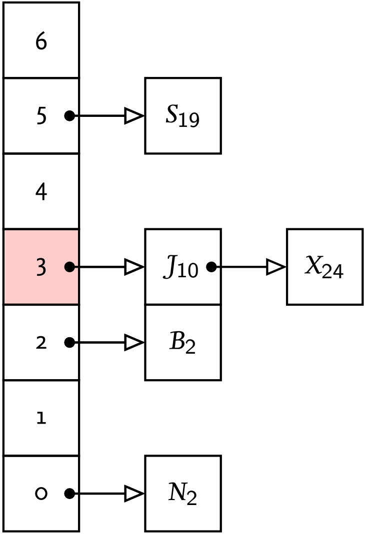
For Wednesday

Acknowledgements

About this Document

Example

Letters Left: X_{24}, W_{23}



Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

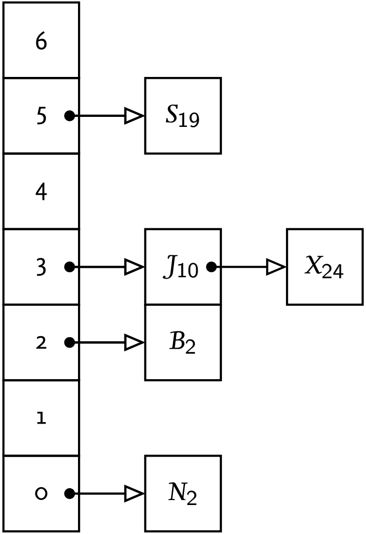
For Wednesday

Acknowledgements

About this Document

Example

Letters Left: X_{24}, W_{23}



Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

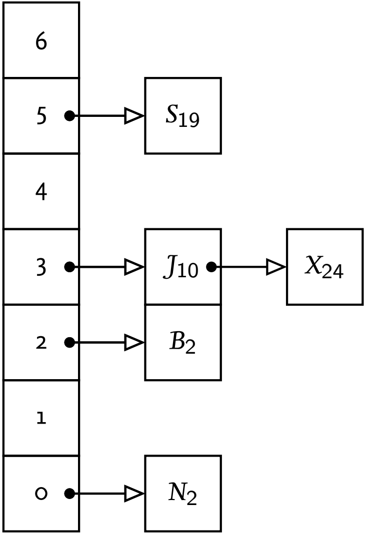
For Wednesday

Acknowledgements

About this Document

Example

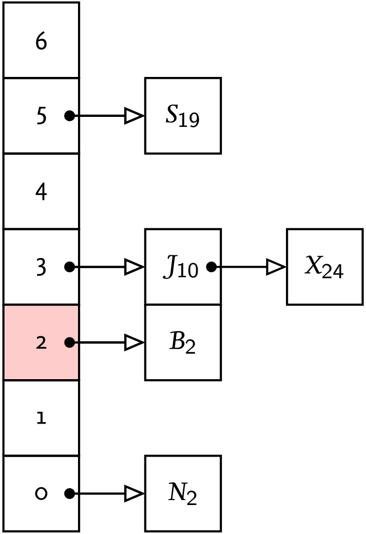
Letters Left: W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

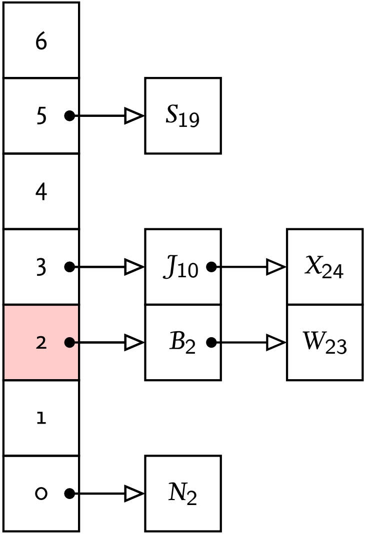
Example

Letters Left: W_{23}



Example

Letters Left: W_{23}



Software Development

M. R. C. van Dongen

Outline

Collections

Linked Lists

Sets

Maps

Queues

Hashing

hashCode( )

Random Hash Functions

Collision Resolution

Running Example

Linear Probing

Double Hashing

Limitation of Open Addressing

Separate Chaining

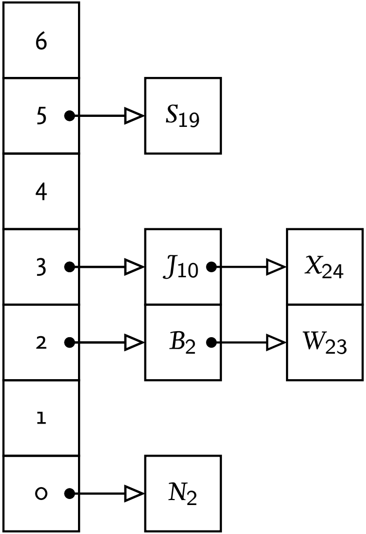
For Wednesday

Acknowledgements

About this Document

Example

Letters Left: W_{23}



Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
hashCode( )
Random Hash Functions
Collision Resolution
Running Example
Linear Probing
Double Hashing
Limitation of Open Addressing
Separate Chaining
For Wednesday
Acknowledgements
About this Document

For Wednesday

- Study [Horstmann 2013, Chapter 14.1–14.4].
- Read [Horstmann 2013, Chapter 15.4].

Acknowledgements

- Parts of this lecture correspond to [Horstmann 2013, Chapter 14.1–14.4].
- Overriding `hashCode( )` is based on [Bloch 2008, Item 9].
- The part about random hashing is based on [Mitzenmacher, and Upfal 2005].
- The running linear probing example is based on [Standish 1994, Chapter 11].

Outline
Collections
Linked Lists
Sets
Maps
Queues
Hashing
For Wednesday
Acknowledgements
About this Document

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.