第十章　進階資料型態

OOP with Ruby

本章內容

- 基本資料型態回顧
- 重新檢視陣列和雜湊 (Arrays and Hashes)
- 堆疊和佇列 (Stacks and Queues)
- 鏈結串列 (Linked Lists)
- 樹狀結構 (Trees)

基本資料型態回顧

- Numerical data
  - Integer class: Fixnum, Bignum
  - Floating point class: Float
- String data
  - Class: String
  - Use single or double quote
  - Escape sequence
  - Sub-string operations
- Range data
- Date time data
  - Class: Time

結構化資料型態回顧

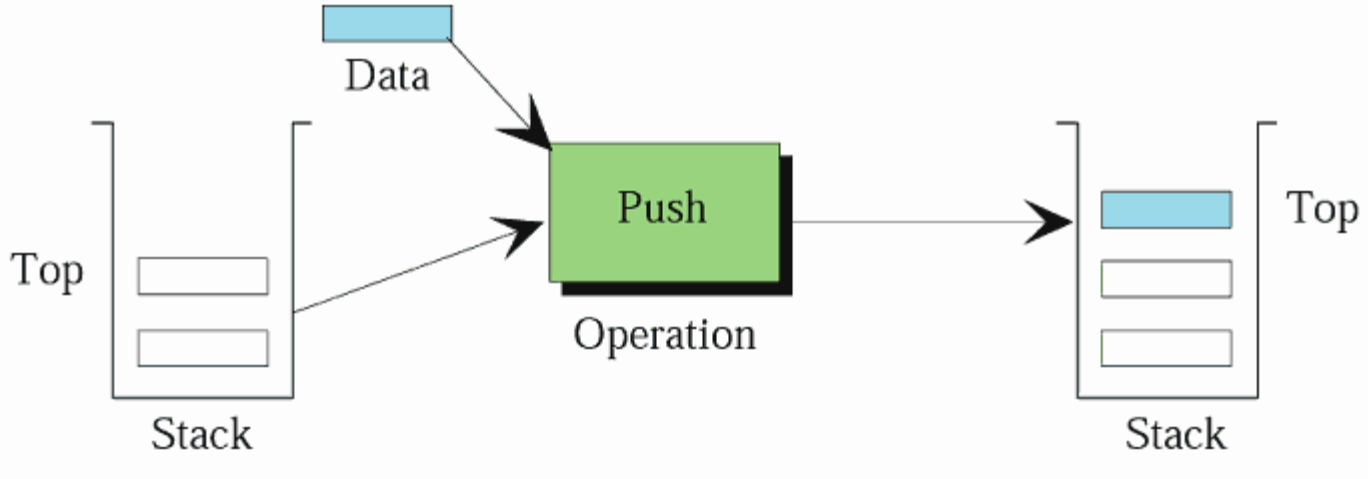
- Structured data types
  - Arrays
    - Indexed collections
    - Store collections of objects
    - Accessible using a integer-based key
  - Hashes
    - Indexed collections
    - Store collections of objects
    - Accessible using an object key

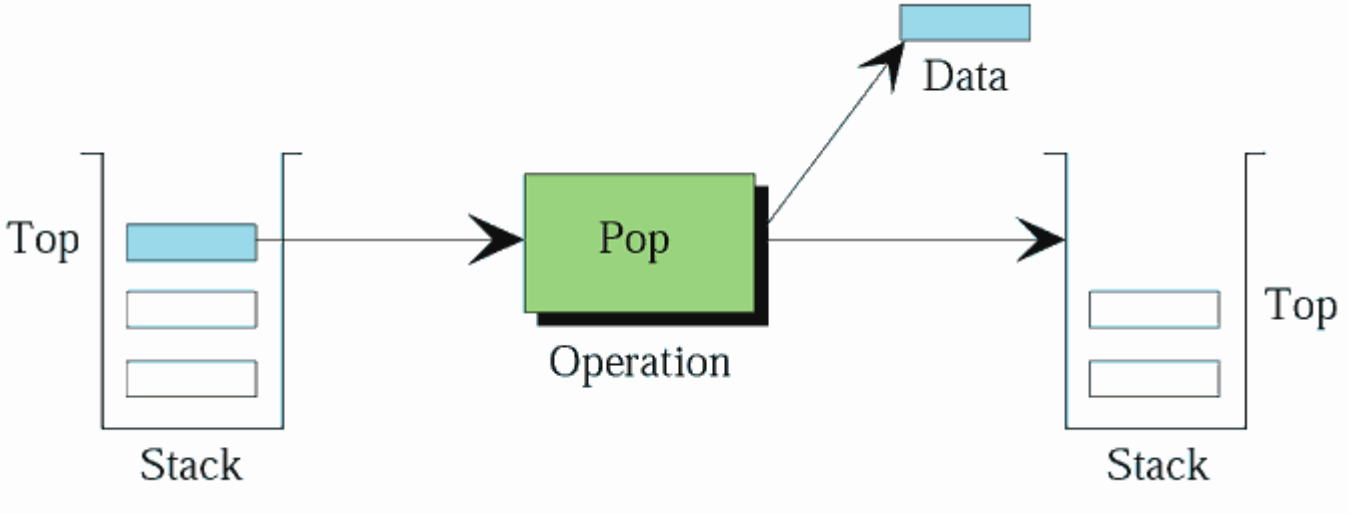
堆疊和佇列 (Stacks and Queues)

- Stacks and queues are the first entities we have discussed that are not strictly built in to Ruby
  - Ruby does not have *Stack* and *Queue* classes
- A stack is a last-in first-out (LIFO) data structure
  - Operations: push (add item), pop (remove item)
  - An array can implement a stack
- A queue is a first-in first-out (FIFO) data structure
  - Operations: enqueue (add item), dequeue (remove item)
  - Queues are useful in more real-time environments where entities are processed as they are presented to the system
  - They are useful in producer/consumer situations
    - A printer queue is a good example

堆疊

- Push and pop operations in a stack





with Ruby

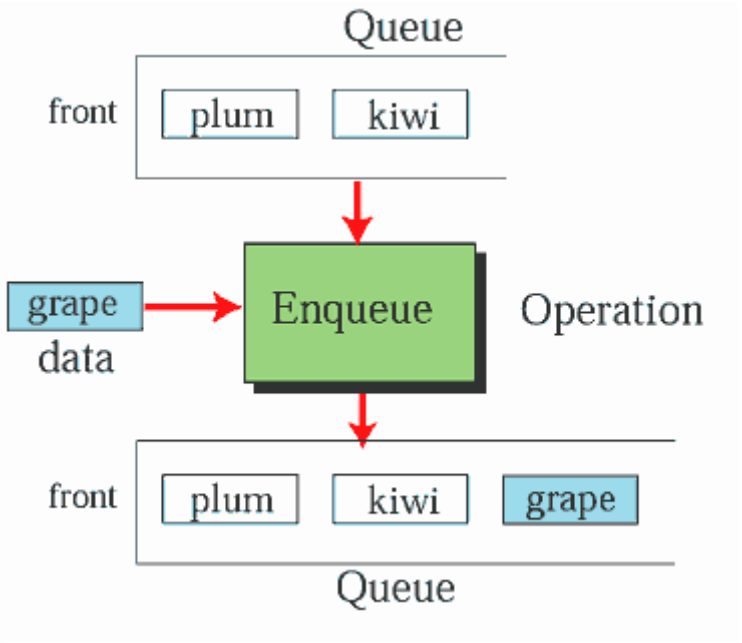
堆疊 (2)

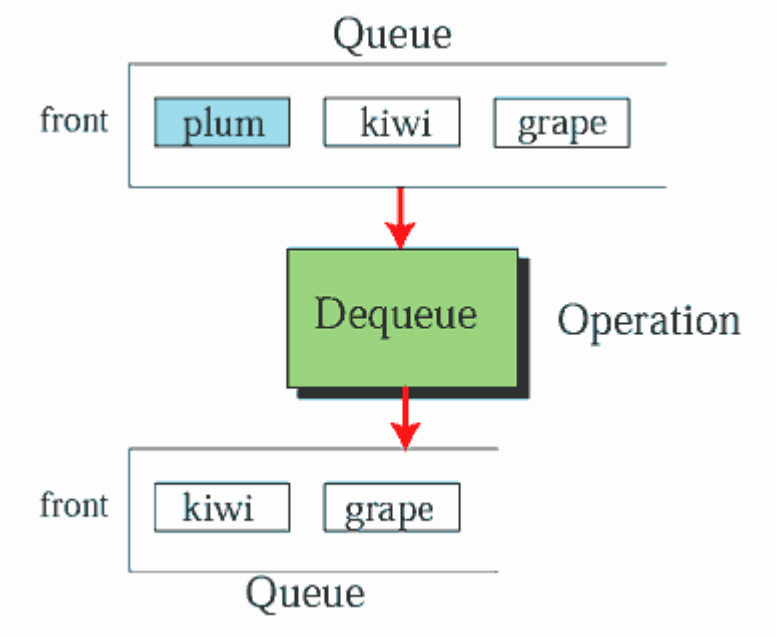
```
class Stack

  def initialize
    @store = []
  end

  def push(x)
    @store.push x
  end

  def pop
    @store.pop
  end

  def peek
    @store.last
  end

  def empty?
    @store.empty?
  end

end
```

- A stack sits at a higher level of abstraction than an array

佇列

- Enqueue and dequeue operations in a queue





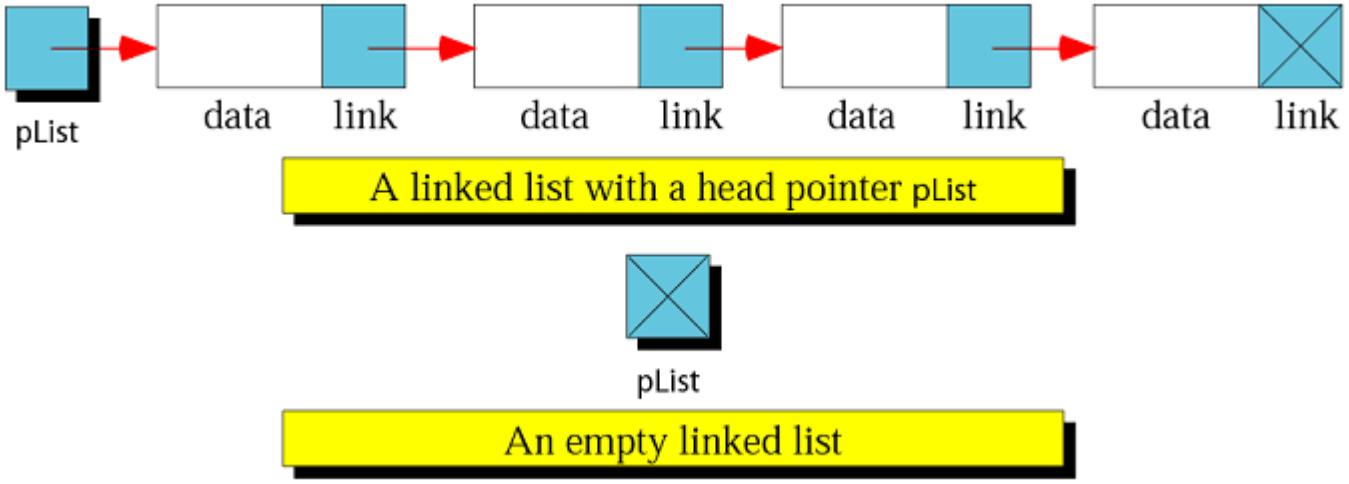
佇列 (2)

```
class Queue

  def initialize
    @store = []
  end

  def enqueue(x)
    @store << x
  end

  def dequeue
    @store.shift
  end

  def peek
    @store.first
  end
  def length
    @store.length
  end

  def empty?
    @store.empty?
  end

end
```

- The two basic queue operations are usually called enqueue and dequeue in the literature
  - The corresponding instance methods in the Array class are called shift and unshift, respectively

鏈結串列 (Linked Lists)

- A linked list is an ordered collection of data in which each element contains the location of the next element
  - The “location” is the reference in Ruby



鏈結串列 (2)

- Some convenient classes in Ruby behave like a linked list

樹狀結構 (Trees)

- A tree is a hierarchical data structure
  - Any item in a tree is a node
  - the first or topmost node is the root
  - A node may have descendants that are below it, and the immediate descendants are called children
  - Conversely, a node may also have a parent (only one) and ancestors
  - A node with no child nodes is called a leaf, otherwise an internal (intermediate) node
  - A subtree consists of a node and all its descendants
  - To travel through a tree (for example, to print it out) is called traversing the tree

二元樹 (Binary Trees)

- Binary tree: a tree which every internal node has at most two children
  - Each node needs an attribute of some kind for storing a piece of data
  - Each node also needs a pair of attributes for referring to the left and right subtrees under that node

```
class Tree

  attr_accessor :left
  attr_accessor :right
  attr_accessor :data

  def initialize(x=nil)
    @left = nil
    @right = nil
    @data = x
  end

  # other methods goes here

end
```

二元樹範例

- Breadth-first insertion and traversal in a tree

```
class Tree

attr_accessor :left
attr_accessor :right
attr_accessor :data

def initialize(x=nil)
  @left = nil
  @right = nil
  @data = x
end
```

- Test the tree

```
items = [1, 2, 3, 4, 5, 6, 7]

tree = Tree.new
items.each { |x| tree.insert(x)}

tree.traverse { |x| print "#{ x}  "}
print "\n"

# Prints "1 2 3 4 5 6 7 "
```

```
def insert(x)
  list = []
  if @data == nil
    @data = x
  elsif @left == nil
    @left = Tree.new(x)
  elsif @right == nil
    @right = Tree.new(x)
  else
    list << @left
    list << @right
    loop do
      node = list.shift
      if node.left == nil
        node.insert(x)
        break
      else
        list << node.left
      end
      if node.right == nil
        node.insert(x)
        break
      else
        list << node.right
      end
    end
  end
end

def traverse()
  list = []
  yield @data
  list << @left if @left != nil
  list << @right if @right != nil
  loop do
    break if list.empty?
    node = list.shift
    yield node.data
    list << node.left if node.left != nil
    list << node.right if node.right != nil
  end
end
```

二元樹範例 (2)

- Sorting using a binary tree

```
class Tree

  # Assumes definitions from
  # previous example...

  def insert(x)
    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new x
      else
        @left.insert x
      end
    else
      if @right == nil
        @right = Tree.new x
          else
            @right.insert x
          end
        end
      end
      def inorder()
        @left.inorder { |y| yield y}  if @left != nil
        yield @data
        @right.inorder { |y| yield y}  if @right != nil
      end

      def preorder()
        yield @data
        @left.preorder { |y| yield y}  if @left != nil
        @right.preorder { |y| yield y}  if @right != nil
      end

      def postorder()
        @left.postorder { |y| yield y}  if @left != nil
        @right.postorder { |y| yield y}  if @right != nil
        yield @data
      end

end
```

- Test the tree

```
items = [50, 20, 80, 10, 30, 70, 90, 5, 14,
         28, 41, 66, 75, 88, 96]

tree = Tree.new

items.each { |x| tree.insert(x)}

tree.inorder { |x| print x, " "}
print "\n"
tree.preorder { |x| print x, " "}
print "\n"
tree.postorder { |x| print x, " "}
print "\n"
```

其他樹狀結構說明

- A tree is a special case of a graph
  - It is a directed acyclic graph (DAG)
- A B-tree is a specialized form of multi-way tree
  - It is an improvement over a binary tree in that it is always balanced (that is, its depth is minimal)
- A red-black tree is a specialized form of binary tree in which each node has a color (red or black)
  - Each node has a pointer back to its parent (meaning that it is arguably not a tree at all because it isn't truly acyclic)
  - A red-black tree maintains its balance through rotations of its nodes. The nodes can be rearranged so that depth is minimized
- An AVL tree is a binary tree that uses slightly more sophisticated insertion and deletion algorithms to keep the tree balanced

本章回顧