Let’s play with our example

```ruby
# Example 1

tree1 = BinaryTree.new(10)
tree1.insert(5)
tree1.insert(15)
tree1.insert(20)

tree1.traverse_in_order

# Example 2

tree2 = BinaryTree.new
input = %w{one two three four five six}

for w in input
  tree2.insert(w)
end

tree2.traverse_in_order
```
## % notation

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%q[]</td>
<td>Non-interpolated String (except for <code>\</code> and <code>\</code>)</td>
</tr>
<tr>
<td>%Q[]</td>
<td>Interpolated String (default)</td>
</tr>
<tr>
<td>%r[]</td>
<td>Interpolated Regexp (flags can appear after the closing delimiter)</td>
</tr>
<tr>
<td>%i[]</td>
<td>Non-interpolated Array of symbols, separated by whitespace (after Ruby 2.0)</td>
</tr>
<tr>
<td>%I[]</td>
<td>Interpolated Array of symbols, separated by whitespace (after Ruby 2.0)</td>
</tr>
<tr>
<td>%w[]</td>
<td>Non-interpolated Array of words, separated by whitespace</td>
</tr>
<tr>
<td>%W[]</td>
<td>Interpolated Array of words, separated by whitespace</td>
</tr>
<tr>
<td>%x[]</td>
<td>Interpolated shell command</td>
</tr>
</tbody>
</table>
Arrays

```ruby
x = Array.new  # same as x = []
x << 10        # same as x.push(10)

x[0] = 99
y = ["Alice", 23, 7.3]
x[1] = y[1] + y[-1]
```

From the end and downwards
Iterating through arrays

```ruby
# Initialize array a
a = [1, 2, 3, 4, 5]\n# Initialize index i
i = 0
# Iterate through the array while i is less than the length of a
while i < a.length
  puts a[i]
  i = i + 1
end
```

```ruby
# Alternatively, use each block to iterate through the array
a = [1, 2, 3, 4, 5]
a.each { |x| puts x }
```

**Code block**
Delimited by:
{ ... } or do ... end

**Parameter list**

**body**

**Code diagram**
More examples on code blocks

- `n.times` runs code block `n` times
- `n.upto(m)` runs code block for integers `n..m`
- `a.find` returns first element `x` of array such that the block returns true for `x`
- `a.collect` applies block to each element of array and returns new array (`a.collect!` modifies the original)
Calling code blocks

```ruby
def countx(x)
  for i in (1..x)
    puts i
    yield
  end
end

countx(4) { puts "foo" }
Let’s improve our binary tree class

class BinaryTree

  def traverse(mode = :in_order, &block)
    case mode
      when :in_order
        @left.traverse mode, &block unless @left.nil?
        yield @value
        @right.traverse mode, &block unless @right.nil?
      when :pre_order
        ...
      when :post_order
        ...
    end
  end
end
Let’s play again with our example

```ruby
tree1 = BinaryTree.new
[10, 20, 15, 5].each{|x| tree1.insert(x)}
tree1.traverse { |x| puts x }

tree2 = BinaryTree.new
%w{one two three four five six}.each{|w| tree2.insert(w)}
tree2.traverse(:in_order) { |x| print " #{x}" }; puts
tree2.traverse(:pre_order) { |x| print " #{x}" }; puts
tree2.traverse(:post_order) { |x| print " #{x}" }; puts
```
Hashes

person = Hash.new # same as person = {}

person["last_name"] = "Rodriguez"
person[:first_name] = "Alice"
order = {"item" => "Corn Flakes", "weight" => 18}
order = {:item => "Corn Flakes", :weight => 18}
order = {item: "Corn Flakes", weight: 18}
Regular Expressions

• Widely use tool for specifying (and implementing) pattern matching processors and even lexical analyzers

• RegExp are first-class citizens in Ruby

```
expr1 = Regexp.new('\(\d+\)')
expr2 = /\(\d+\)/
```

• Ruby’s syntax is mostly borrowed from Perl’s
A toy example

```ruby
def code_matcher(expr, str)
  if expr =~ str
    puts "Country code #\{expr.match(str)\}"
  else
    puts 'No country code provided'
  end
end

phone1 = '(372) 58 442-456'
phone2 = '58 442-456'

expr = /\d+/#

code_matcher(expr, phone1)
code_matcher(expr, phone2)
```

- `=~`:
  - nil (cf. false) when no match
  - index of the first occur. on the string, otherwise

- `match`:
  - nil when no match
  - Instance of MatchData, otherwise
# Character classes

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[Rr]uby/</td>
<td>Matches “Ruby” or “ruby”</td>
</tr>
<tr>
<td>/Rub(y</td>
<td>en)/</td>
</tr>
<tr>
<td>/[aeiou]/</td>
<td>Matches a single lowercase vowel</td>
</tr>
<tr>
<td>/[0-9]/</td>
<td>Matches a single digit</td>
</tr>
<tr>
<td>/[a-zA-Z]/</td>
<td>Matches a single alphabetic character</td>
</tr>
<tr>
<td>/[^aeiou]/</td>
<td>Matches anything other than a lowercase vowel</td>
</tr>
<tr>
<td>/[^0-9]/</td>
<td>Matches anything other than a digit</td>
</tr>
</tbody>
</table>
## Character classes (continuation)

<table>
<thead>
<tr>
<th>Abreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>A digit character, as [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>Any character except a digit, as [^0-9]</td>
</tr>
<tr>
<td>\s</td>
<td>Any whitespace, as [ \t\r\n\f]</td>
</tr>
<tr>
<td>\S</td>
<td>Any character except a whitespace</td>
</tr>
<tr>
<td>\w</td>
<td>Any word character, as [a-zA-Z0-9_]</td>
</tr>
<tr>
<td>\W</td>
<td>Any character except a work character, as [^a-zA-Z0-9_]</td>
</tr>
<tr>
<td>.</td>
<td>Any character except a whitespace</td>
</tr>
</tbody>
</table>

### POSIX character classes

`[:alpha:], [:alnum:], [:blank:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], [:xdigit:]`
# Repetition and anchoring

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ruby?/</td>
<td>Matches “rub” or “ruby”</td>
</tr>
<tr>
<td>/ruby*/</td>
<td>Matches “rub” followed by 0 or more “y”s</td>
</tr>
<tr>
<td>/ruby+/</td>
<td>Matches “rub” followed by 1 or more “y”s</td>
</tr>
<tr>
<td>/\d{3}/</td>
<td>Matches exactly three digits</td>
</tr>
<tr>
<td>/\d{3,}/</td>
<td>Matches 3 or more digits</td>
</tr>
<tr>
<td>/\d{3,5}/</td>
<td>Matches 3, 4 or 5 digits</td>
</tr>
<tr>
<td>/^Ruby/</td>
<td>Matches “rub” or “ruby”</td>
</tr>
<tr>
<td>/Ruby$/</td>
<td>Matches “rub” followed by 0 or more “y”s</td>
</tr>
<tr>
<td>/\ARuby/</td>
<td>Matches “rub” followed by 1 or more “y”s</td>
</tr>
<tr>
<td>/Ruby\Z/</td>
<td>Matches exactly three digits</td>
</tr>
<tr>
<td>/\bRuby\b/</td>
<td>Matches “Ruby” if and only if found as single word</td>
</tr>
</tbody>
</table>
Matching groups

```
exp1 = /\d+\D+(\d+)/
exp2 = /\d+/!

m = exp1.match("_ 123 abc 345 x 67 end")
puts "#{\$1}, #{\$2}"
m.captures

exp2.match("_ 123 abc 345 x 67 end")
puts "#{\$1}, #{\$2}"

m = "_ 123 abc 345 x 67 end".scan(/(\d+)/)

m.each { |x| puts x }```

Matching and replacing

```ruby
str1 = "he said 'hello' to me"
str2 = str1.dup

puts str1.gsub(/he/, 'He')
puts str2.gsub(/\bhe\b/, 'He')
```