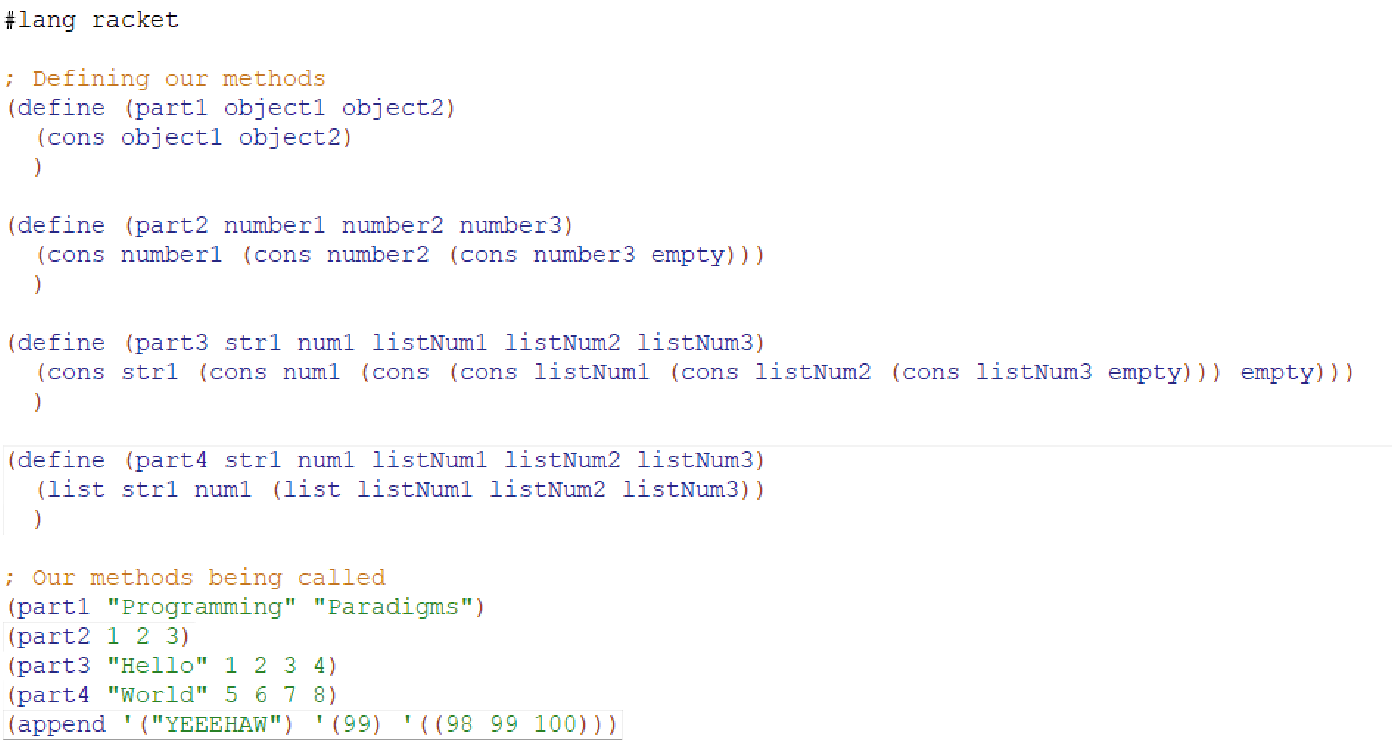
Irish Senthilkumar 16342613 CT331 Assignment 2

Github Link: https://github.com/16342613/CT331-Assignment-2

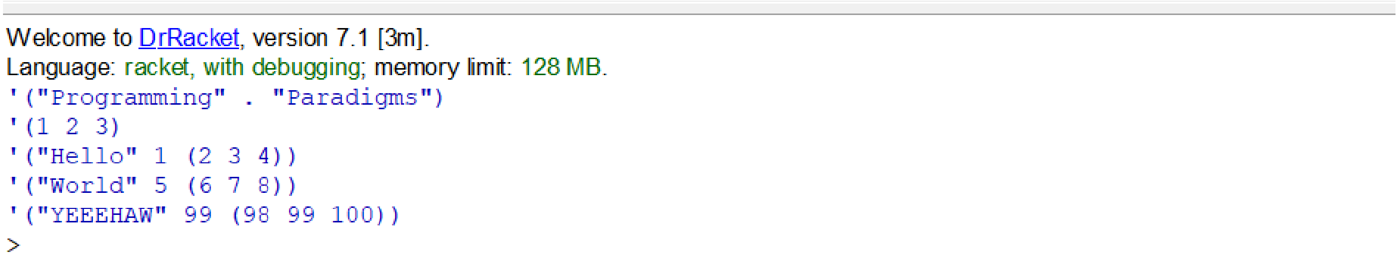
Question 1:

(a)

```
#lang racket

; Defining our methods
(define (part1 object1 object2)
  (cons object1 object2)
  )

(define (part2 number1 number2 number3)
  (cons number1 (cons number2 (cons number3 empty)))
  )

(define (part3 str1 num1 listNum1 listNum2 listNum3)
  (cons str1 (cons num1 (cons (cons listNum1 (cons listNum2 (cons listNum3 empty))) empty)))
  )

(define (part4 str1 num1 listNum1 listNum2 listNum3)
  (list str1 num1 (list listNum1 listNum2 listNum3))
  )

; Our methods being called
(part1 "Programming" "Paradigms")
(part2 1 2 3)
(part3 "Hello" 1 2 3 4)
(part4 "World" 5 6 7 8)
(append '("YEEEHAW") '(99) '((98 99 100))) ; PART 5
```



```
#lang racket

; Defining our methods
(define (part1 object1 object2)
  (cons object1 object2)
  )

(define (part2 number1 number2 number3)
  (cons number1 (cons number2 (cons number3 empty)))
  )

(define (part3 str1 num1 listNum1 listNum2 listNum3)
  (cons str1 (cons num1 (cons (cons listNum1 (cons listNum2 (cons listNum3 empty))) empty)))
  )

(define (part4 str1 num1 listNum1 listNum2 listNum3)
  (list str1 num1 (list listNum1 listNum2 listNum3))
  )

; Our methods being called
(part1 "Programming" "Paradigms")
(part2 1 2 3)
(part3 "Hello" 1 2 3 4)
(part4 "World" 5 6 7 8)
(append '("YEEEHAW") '(99) '((98 99 100)))
```



```
Welcome to DrRacket, version 7.1 [3m].
Language: racket, with debugging; memory limit: 128 MB.
'("Programming" . "Paradigms")
'(1 2 3)
'("Hello" 1 (2 3 4))
'("World" 5 (6 7 8))
'("YEEEHAW" 99 (98 99 100))
>
```

(b)

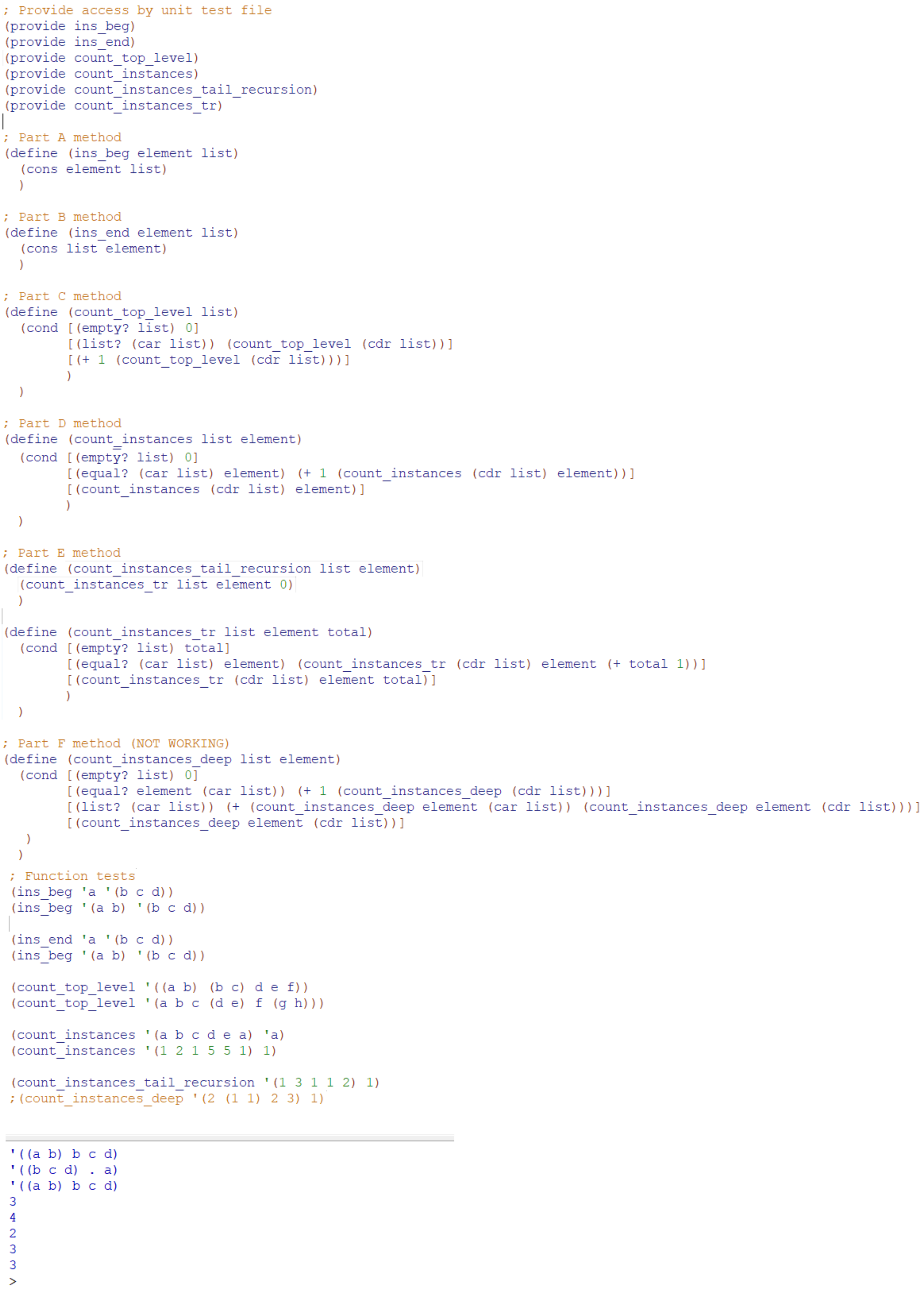
- For part 1, cons is used to make a con pair with the 2 input parameters.
- For part 2, a nested cons is used to create a list. We keep using nested cons until we get to the last element in the list, where we use a con pair of the last element and empty, since cons requires 2 input parameters.
- For part 3, we once again use nested cons, but here we use a double nested cons to get a nested list inside our list.
- For part 4, we use the list function to make the super-list, and another list function inside the super-list to make the sub-list.
- For part 5, we just append list items, whatever they may be, to the super-list.

Question 2:

```
#lang racket

; Provide access by unit test file
(provide ins_beg)
(provide ins_end)
(provide count_top_level)
(provide count_instances)
(provide count_instances_tail_recursion)
(provide count_instances_tr)
```

```
; Part A method
(define (ins_beg element list)
  (cons element list)
  )

; Part B method
(define (ins_end element list)
  (cons list element)
  )

; Part C method
(define (count_top_level list)
  (cond [(empty? list) 0]
        [(list? (car list)) (count_top_level (cdr list))]
        [(+ 1 (count_top_level (cdr list)))]
        )
  )

; Part D method
(define (count_instances list element)
  (cond [(empty? list) 0]
        [(equal? (car list) element) (+ 1 (count_instances (cdr list) element))]
        [(count_instances (cdr list) element)]
        )
  )

; Part E method
(define (count_instances_tail_recursion list element)
  (count_instances_tr list element 0)
  )

(define (count_instances_tr list element total)
  (cond [(empty? list) total]
        [(equal? (car list) element) (count_instances_tr (cdr list) element (+ total 1))]
        [(count_instances_tr (cdr list) element total)]
        )
  )

; Part F method (NOT WORKING)
(define (count_instances_deep list element)
  (cond [(empty? list) 0]
        [(equal? element (car list)) (+ 1 (count_instances_deep (cdr list)))]
        [(list? (car list)) (+ (count_instances_deep element (car list)) (count_instances_deep element (cdr list)))]
        [(count_instances_deep element (cdr list))]
   )
  )

; Function tests
(ins_beg 'a '(b c d))
(ins_beg '(a b) '(b c d))

(ins_end 'a '(b c d))
(ins_beg '(a b) '(b c d))
```

```
(count_top_level '((a b) (b c) d e f))
(count_top_level '(a b c (d e) f (g h)))

(count_instances '(a b c d e a) 'a)
(count_instances '(1 2 1 5 5 1) 1)

(count_instances_tail_recursion '(1 3 1 1 2) 1)
;(count_instances_deep '(2 (1 1) 2 3) 1)
```



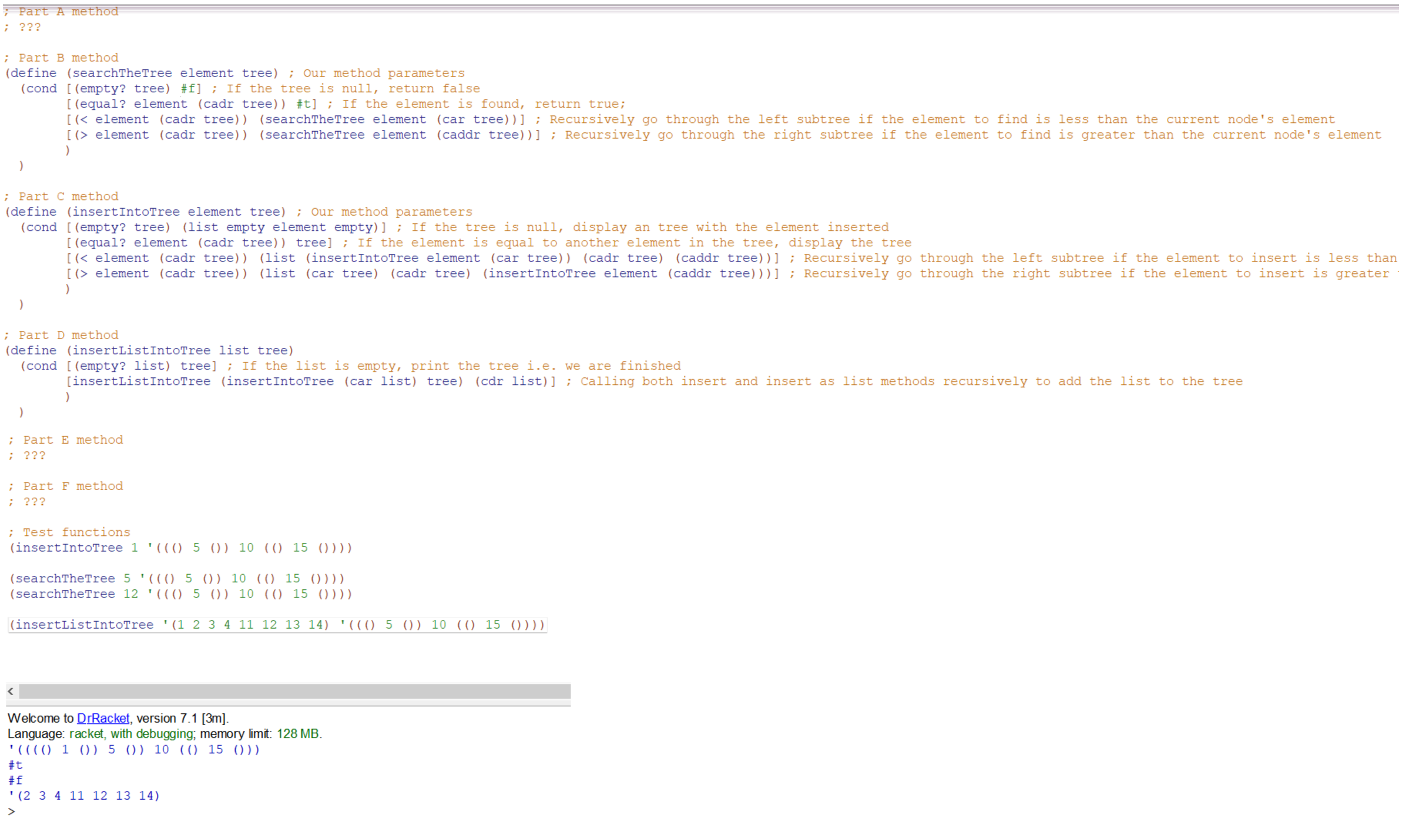
```
; Provide access by unit test file
(provide ins_beg)
(provide ins_end)
(provide count_top_level)
(provide count_instances)
(provide count_instances_tail_recursion)
(provide count_instances_tr)

; Part A method
(define (ins_beg element list)
  (cons element list)
  )

; Part B method
(define (ins_end element list)
  (cons list element)
  )

; Part C method
(define (count_top_level list)
  (cond [(empty? list) 0]
        [(list? (car list)) (count_top_level (cdr list))]
        [(+ 1 (count_top_level (cdr list)))]
        )
  )

; Part D method
(define (count_instances list element)
  (cond [(empty? list) 0]
        [(equal? (car list) element) (+ 1 (count_instances (cdr list) element))]
        [(count_instances (cdr list) element)]
        )
  )

; Part E method
(define (count_instances_tail_recursion list element)
  (count_instances_tr list element 0)
  )

(define (count_instances_tr list element total)
  (cond [(empty? list) total]
        [(equal? (car list) element) (count_instances_tr (cdr list) element (+ total 1))]
        [(count_instances_tr (cdr list) element total)]
        )
  )

; Part F method (NOT WORKING)
(define (count_instances_deep list element)
  (cond [(empty? list) 0]
        [(equal? element (car list)) (+ 1 (count_instances_deep (cdr list)))]
        [(list? (car list)) (+ (count_instances_deep element (car list)) (count_instances_deep element (cdr list)))]
        [(count_instances_deep element (cdr list))]
   )
  )
 ; Function tests
 (ins_beg 'a '(b c d))
 (ins_beg '(a b) '(b c d))

 (ins_end 'a '(b c d))
 (ins_beg '(a b) '(b c d))

 (count_top_level '((a b) (b c) d e f))
 (count_top_level '(a b c (d e) f (g h)))

 (count_instances '(a b c d e a) 'a)
 (count_instances '(1 2 1 5 5 1) 1)

 (count_instances_tail_recursion '(1 3 1 1 2) 1)
 ;(count_instances_deep '(2 (1 1) 2 3) 1)
```

```
'((a b) b c d)
'((b c d) . a)
'((a b) b c d)
3
4
2
3
3
>
```

Question 3:

#lang racket

```
; Structure is (Left, Element, Right)

; Part A method
; ???

; Part B method
(define (searchTheTree element tree) ; Our method parameters
  (cond [(empty? tree) #f] ; If the tree is null, return false
       [(equal? element (cadr tree)) #t] ; If the element is found, return true;
       [(< element (cadr tree)) (searchTheTree element (car tree))] ; Recursively go through the left subtree if the element to find is less than the current node's element
       [(> element (cadr tree)) (searchTheTree element (caddr tree))] ; Recursively go through the right subtree if the element to find is greater than the current node's element
       )
  )

; Part C method
(define (insertIntoTree element tree) ; Our method parameters
  (cond [(empty? tree) (list empty element empty)] ; If the tree is null, display an tree with the element inserted
       [(equal? element (cadr tree)) tree] ; If the element is equal to another element in the tree, display the tree
       [(< element (cadr tree)) (list (insertIntoTree element (car tree)) (cadr tree) (caddr tree))] ; Recursively go through the left subtree if the element to insert is less than the current node's element
       [(> element (cadr tree)) (list (car tree) (cadr tree) (insertIntoTree element (caddr tree)))] ; Recursively go through the right subtree if the element to insert is greater than the current node's element
       )
  )

; Part D method
(define (insertListIntoTree list tree)
  (cond [(empty? list) tree] ; If the list is empty, print the tree i.e. we are finished
       [insertListIntoTree (insertIntoTree (car list) tree) (cdr list)] ; Calling both insert and insert as list methods recursively to add the list to the tree
       )
  )

; Part E method
; ???

; Part F method
; ???

; Test functions
;(insertIntoTree 1 '((() 5 ()) 10 (() 15 ())))

;(searchTheTree 5 '((() 5 ()) 10 (() 15 ())))
;(searchTheTree 12 '((() 5 ()) 10 (() 15 ())))

;(insertListIntoTree '(1 2 3 4 11 12 13 14) '((() 5 ()) 10 (() 15 ())))
```



```
; Part A method
; ???

; Part B method
(define (searchTheTree element tree) ; Our method parameters
  (cond [(empty? tree) #f] ; If the tree is null, return false
        [(equal? element (cadr tree)) #t] ; If the element is found, return true;
        [(< element (cadr tree)) (searchTheTree element (car tree))] ; Recursively go through the left subtree if the element to find is less than the current node's element
        [(> element (cadr tree)) (searchTheTree element (caddr tree))] ; Recursively go through the right subtree if the element to find is greater than the current node's element
        )
  )

; Part C method
(define (insertIntoTree element tree) ; Our method parameters
  (cond [(empty? tree) (list empty element empty)] ; If the tree is null, display an tree with the element inserted
        [(equal? element (cadr tree)) tree] ; If the element is equal to another element in the tree, display the tree
        [(< element (cadr tree)) (list (insertIntoTree element (car tree)) (cadr tree) (caddr tree))] ; Recursively go through the left subtree if the element to insert is less than
        [(> element (cadr tree)) (list (car tree) (cadr tree) (insertIntoTree element (caddr tree)))] ; Recursively go through the right subtree if the element to insert is greater
        )
  )

; Part D method
(define (insertListIntoTree list tree)
  (cond [(empty? list) tree] ; If the list is empty, print the tree i.e. we are finished
        [insertListIntoTree (insertIntoTree (car list) tree) (cdr list)] ; Calling both insert and insert as list methods recursively to add the list to the tree
        )
  )

 ; Part E method
 ; ???

 ; Part F method
 ; ???

 ; Test functions
 (insertIntoTree 1 '((() 5 ()) 10 (() 15 ())))

 (searchTheTree 5 '((() 5 ()) 10 (() 15 ())))
 (searchTheTree 12 '((() 5 ()) 10 (() 15 ())))

 (insertListIntoTree '(1 2 3 4 11 12 13 14) '((() 5 ()) 10 (() 15 ())))
```

```
Welcome to DrRacket, version 7.1 [3m].
Language: racket, with debugging; memory limit: 128 MB.
'(((() 1 ()) 5 ()) 10 (() 15 ()))
#t
#f
'(2 3 4 11 12 13 14)
>
```