Assignment 2 – Cathal Lawlor – 21325456

Question 1

```
#lang racket
(display "\nAssigment 2\nQuestion 1\n\n")

;;A cons pair of two numbers
(cons 1 2)

;; A list of 3 numbers, using only the cons function.
(cons 2 (cons 4 (cons 7 empty)))

;;A list containing a string, a number and a nested list of three numbers, using only the
;;cons function.
(cons "This is a string!"
      (cons 5
            (cons (cons 1 (cons 2 (cons 3 empty))) empty)
      )
)

;;A list containing a string, a number and a nested list of three numbers, using only the
;;list function
(list "This is a string!" 5 (list 1 2 3))
;;A list containing a string, a number and a nested list of three numbers, using only the
;;append function.
(append '("This is a string!") '(5) '((1 2 3)))
```

```
Assigment 2
Question 1

A cons pair of two numbers.
'(1 . 2)

A list of 3 numbers, using only the cons function.
'(2 4 7)

;A list containing a string, a number and a nested list of three numbers.
'("This is a string!" 5 (1 2 3))

A list containing a string, a number and a nested list of three numbers - list
'("This is a string!" 5 (1 2 3))

A list containing a string, a number and a nested list of three numbers - append.
'("This is a string!" 5 (1 2 3))
```

1. The cons function creates a cons pair, which is not always a 'list' as we see here, due to the two elements being delimited by a '.' as it is only a cons pair, and not a 'list' which is actually a cons pair, where the second element is actually a 'list' or empty.
2. We create a list using cons, of three numbers, where firstly we create a list with just one element, having 3 and empty in the innermost cons. From there we make a two-element list by making a cons pair, with 2 and the earlier created one element list. We again create a three-element list by making a cons pair, with 1 and the two-element list we have.
3. We use the cons to make a one-element list containing 3, and then pair that list with 2 and 1. Then we use cons to pair that three-element list to empty, generating a 'list', where we pair that sub-list (1,2,3) with 5. From there finally we use cons to pair a string to that list (5, (1,2,3)) containing the number and the sub-list.
4. We use the list function to put 3 elements in a list, being a string ("This is a string"), a number (5), and then a sub-list (1,2,3), created by running (list 1 2 3) inside the function.
5. First, we create three lists using '() of the string, the number and then a list of three numbers. Then, we use append which can only accepts lists as an argument, to append the three lists together.

The difference between cons, list, and append is that cons creates a new pair, list creates a new list, and append concatenates lists. cons is used to build up pairs and lists, while list is used to create new lists from scratch. append is used to combine existing lists into a single list.

Question 2

```
#lang racket
(provide ins_beg)
(provide ins_end)
(provide cout_top_level)
(provide count_instances)
(provide tail_count_instances)
(provide tail_count_instances_helper)
(provide count_instances_deep)

;; ins_beg: Scheme function that takes an element and a list and returns a new list with
the element added to the beginning of the original list.
(define (ins_beg el lst)
  (cons el lst)) ;;creating new list containing the new element at the start using cons

;; ins_end: Scheme function that takes an element and a list and returns a new list with
the element added to the end of the original list.
(define (ins_end el lst)
  (append lst (cons el empty))) ;;Assuming lst is a list, and appending a new list
(created using cons) with el to the end of lst

;; cout_top_level: Scheme function that takes a list and returns the number of top-level
items in the list (i.e., items that are not in a sublist).
(define (cout_top_level lst)
  (if (empty? lst) ;; empty? is a built-in function to 0 when true - if the list is empty.
      0
      (+ 1 (cout_top_level (cdr lst))) ;;if not empty, return 1 + counting the rest of the
list - recursively
  )
)

;;D. Write a non-tail recursive function called count_instances that counts the number
;;of times an item occurs in a list of items (you may assume all items are atomic).
(define (count_instances item lst)
  (cond
    [(null? lst) 0] ;; Base case - list is empty / finished counting
    [(equal? (car lst) item) (+ 1 (count_instances item (cdr lst)))] ;;If the first
element in list matches return incremented by 1 + recursively count the remainder of the
list
    [else (count_instances(cdr lst))] ;;Otherwise, continue with the rest of the list.
  )
)


;; E. Write a tail-recursive version of part D solution called count_instances_tr
(define (tail_count_instances item lst cnt)
  (cond
    [(null? lst) cnt] ;; Base case - list is empty, finished counting, return count

    ;; if first element is equals to the item
    [(eq? (car lst) item)
      ;; increment 1 + recurse remainder of the list
      (tail_count_instances item (cdr lst) (+ cnt 1))
```

```
    ]

  ;; else - first element isn't equal to the item
  [else
    ;; recursively count the remainder of the list - count is the same
    (tail_count_instances item (cdr lst) cnt)
  ]
  )
)

;; Tail recursive function helper for count of a given item in a list - assumption all
items are atomic
(define (tail_count_instances_helper item lst)
  ;;Calling the function with starter count - 0
  (tail_count_instances item lst 0)
)


;; F. Write a Scheme function named count_instances_deep that counts the number of
;; times an item occurs in a list of items; note that the list may contain sub-lists and
you
;; should also count occurrences in those lists.
(define (count_instances_deep item lst)
  (cond
    [(null? lst)  ;; Base case - list is empty / finished counting
      0
    ]

    ;; If the first element in the list is a list, recurse through the list, and then
recurse through the remainder of the main list
    ;; returning the sum of the two results
    [(pair? (car lst))
      (+ (count_instances_deep item (car lst)) (count_instances_deep item (cdr lst)))
    ]

    ;; If the first element in list matches return incremented by 1 + recursively count
the remainder of the list
    [(equal? (car lst) item)
      (+ 1 (count_instances_deep item (cdr lst)))
    ]

    [else (count_instances_deep item (cdr lst))] ;;Otherwise, continue with the rest of
the list.
  )
)
```

Question 2 continued.

Part A

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

Part B

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

Part C

```
(define my-list '(1 2 3 2 4 2 5))
(display (cout_top_level my-list))
7
```

Part D

```
> (define my-list '(1 2 3 2 4 2 5))
(define item-to-count 2)

(display (count_instances item-to-count my-list))
3
>
```

Part E

```
> (define my-list '(1 2 3 2 4 2 5))
(define item-to-count 2)

(display (count_instances_tr_helper item-to-count my-list))
3
```

Part F

```
> (define my-list '(1 (2 (1 3)) 4 (1 (5 (1 6))) 7))

(define item-to-count 1)

(display (count_instances_deep item-to-count my-list))

4
```

Question 3

Part A

```
> (define bst '(5 (3 (2 () ()) (4 () ())) (8 (7 () ()) (9 () ()))))
(display_in_order bst)
2
3
4
5
7
8
9
```

Part B

```
> ;; Define a binary search tree
(define bst '(5 (3 (2 () ()) (4 () ())) (8 (7 () ()) (9 () ()))))

(display "Testing tree_search:\n")

;; Test cases
(display "Is 4 present in the tree? ")
(if (tree_search 4 bst) (display "#t") (display "#f"))
(newline)

(display "Is 6 present in the tree? ")
(if (tree_search 6 bst) (display "#t") (display "#f"))
(newline)

(display "Is 7 present in the tree? ")
(if (tree_search 7 bst) (display "#t") (display "#f"))
(newline)
Testing tree_search:
Is 4 present in the tree? #t
Is 6 present in the tree? #f
Is 7 present in the tree? #t
```

Part C

```
> (define empty-tree '()) ; Define an empty tree

;; Insert items into the tree
(define tree1 (tree_insert 5 empty-tree))
(define tree2 (tree_insert 3 tree1))
(define tree3 (tree_insert 8 tree2))
(define tree4 (tree_insert 2 tree3))
(define tree5 (tree_insert 4 tree4))
(define tree6 (tree_insert 7 tree5))
(define tree7 (tree_insert 9 tree6))

;; Define a function to display the binary search tree in sorted order
(define (display_tree_in_order bst)
  (cond
    [(null? bst) '()]
    [else
     (append (display_tree_in_order (cadr bst))
             (list (car bst))
             (display_tree_in_order (caddr bst)))]))

(display "Original binary search tree (empty):\n")
(display (display_tree_in_order empty-tree))
(newline)

(display "Binary search tree after inserting elements:\n")
(for-each (lambda (item) (display item) (newline)) (display_tree_in_order tree7))

Original binary search tree (empty):
()
Binary search tree after inserting elements:
2
3
4
5
7
8
9
```

Part D

```
> (display_in_order (tree_insert_list '(5 3 7 1 4 6 8) '()))
1
3
4
5
6
7
8
>
```

Part E

```
> (tree_sort '(5 3 7 1 4 6 8))
1
3
4
5
6
7
8
>
```

Part F

```
> (display "\nsort-ascending\n")
(display_in_order (tree_sort_ho_list '(5 3 7 1 4 6 8) '() sort-ascending))

(display "\n")
(display "\nsort-descending\n")
(display_in_order (tree_sort_ho_list '(5 3 7 1 4 6 8) '() sort-descending))

(display "\n")
(display "\nsort-ascending-last-digit\n")
(display_in_order (tree_sort_ho_list '(123 45 71 12 49 1046 18) '() sort-ascending-last-digit))

sort-ascending
1
3
4
5
6
7
8


sort-descending
8
7
6
5
4
3
1


sort-ascending-last-digit
71
12
123
45
1046
18
49
```

```racket
#lang racket


(provide display_in_order)
(provide tree_search)
(provide tree_insert)
(provide tree_insert_list)
(provide tree_sort)
(provide tree_sort_ho)
(provide tree_sort_ho_list)
(provide sort-ascending)
(provide sort-descending)
(provide sort-ascending-last-digit)




;; A. Display in sorted order the contents of a binary search tree

;; Once the binary search tree has been created, its elements can be retrieved in-order
by:
  ;; • Recursively traversing the left subtree of the root node.
  ;; • Accessing the node itself
  ;; • Then recursively traversing the right subtree of the node

(define (display_in_order bst)
  (cond
  ;; If current tree is null, print out a string that's empty (nothing)
  [(null? bst) (display "")]

  ;; If the binary tree has nodes - continue traversing
  [else
    ;; Traverse and display the contents of the left sub-tree of current node
    (display_in_order (cadr bst))

    ;; display current node
    (display (car bst))
    (newline) ;; new line for formatting

    ;; Traverse and display the contents of the right sub-tree of current node
    (display_in_order (caddr bst)) ;; AKA (cadr (cdr bst))) - access the right sub-tree
which is the third element in the list
  ])
)

;; B. Return #t or #f if a given item is present or absent in a tree or not. The function
;; should take the item and a list representing a tree.
(define (tree_search item bst)
  (cond
    ;; If current tree is null, return false
    [(null? bst) #f]

    ;; If the item is found, return true
    [(equal? item (car bst)) #t]
```

```
    [else
      (or
        (tree_search item (cadr bst)) ;; search left sub-tree
        (tree_search item (caddr bst)) ;; search right sub-tree
        ;; Could have checked if item is less than or greater than the current node for
more efficiency
        ;; but the improved efficiency is not worth the extra code / complexity
      )
    ]
  )
)


;; C. Insert an item correctly into a list representing a binary search tree. Your
function
;; should take an item and a tree as inputs.
(define (tree_insert item bst)
  (cond
    ;; If the tree is empty, set current node to 5, return
    [(null? bst)
      (list item '() '())
    ]

    ;; If item is less than the current node, insert into left sub-tree
    [(< item (car bst))
      ;; create a new tree, keeping the same root node, not changing the right sub-tree,
      ;; but a left sub-tree that has the item inserted
      (list (car bst) (tree_insert item (cadr bst)) (caddr bst))
    ]

    ;; If item is greater than the current node, insert into right sub-tree
    [(> item (car bst))
      ;; create a new tree, keeping the same root node, not changing the left sub-tree,
      ;; but a right sub-tree that has the item inserted
      (list (car bst) (cadr bst) (tree_insert item (caddr bst)))
    ]

    ;; Else, the item is equal to the current node, return the tree (doing nothing)
    [else
      bst
    ]
  )
)


;; D. Take a list of items and insert them into a binary search tree.
(define (tree_insert_list lst bst)
  (if (null? lst) ;; if the list is empty, return the tree as is
      bst

      ;; otherwise, recurse through with the remainder of the list & binary tree created
from the first element into bst
      (tree_insert_list (cdr lst) (tree_insert (car lst) bst))
```

```
  )
)

;; E. Implement a tree-sort algorithm. Your function should take a list of items and
display them in sorted order.
(define (tree_sort lst)
  ;; insert the list into a binary search tree structure to sort it and then displaying
the contents in order
  (display_in_order (tree_insert_list lst '()))
)

;; F. Implement a higher order version of the tree-sort function that takes a list and a
;; function that determines the sorted order. For example, write a version that sorts
;; the list in ascending, descending and ascending based on last digit.
(define (tree_sort_ho item bst compare-func)
  (cond
    ;; if there are no elements in the list, create an empty tree
    [(null? bst)
      (list item '() '())
    ]

    ;; if the item is to go before the current node, insett it to the left hand side of
the bst
    [(compare-func item (car bst))
      ;; create new bst with same root node, same right-hand side, but a left hand side
that has the item inserted
      (list (car bst) (tree_sort_ho item (cadr bst) compare-func) (caddr bst))
    ]

    ;; if the item is to go after the current node, insert it to the right hand side of
the bst
    [(compare-func (car bst) item)
      ;; create new bst with same root node, same left hand side, but a right hand side
that has the item inserted
      (list (car bst) (cadr bst) (tree_sort_ho item (caddr bst) compare-func))
    ]

    ;; otherwise, the item is equal to the current node, return the tree as is
    [else
      bst
    ]
  )
)

;; Function to sort a list using a higher order function
(define (tree_sort_ho_list lst bst compare-func)
  (if (null? lst) ;; if the list is empty, return the tree as is
    bst

    ;; else, recurse through with the remainder of the list & binary tree created from the
first element into bst
    (tree_sort_ho_list (cdr lst) (tree_sort_ho (car lst) bst compare-func) compare-func)

  )
```

```
)

;; sorts the list in ascending order - states if items were supplied in correct order
(define (sort-ascending item1 item2)
  (if (< item1 item2)
    #t
    #f
  )
)

;; sorts the list in descending order - states if items were supplied in correct order
(define (sort-descending item1 item2)
  (if (> item1 item2)
    #t
    #f
  )
)

;; sorts the list in ascending order based on the last digit - e.g. if item1 = item2, then
return false
(define (sort-ascending-last-digit item1 item2)
  (if (< (modulo item1 10) (modulo item2 10))
    #t
    #f
  )
)
```