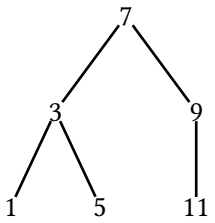
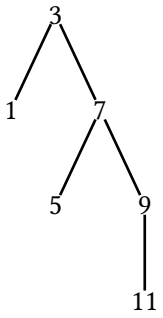
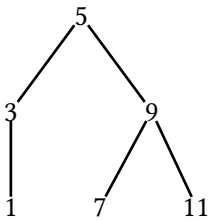
The two trees produce identical results for every tree. This is because they both recursively build a list starting from the left-most node of a tree (by finding the left-branch of the left-branch ...), then that node's parent, and then the right branch of that subtree, and then working back up to the right branch of the subtree of which the current subtree is the left branch, and so on. For both of the procedures, and all three trees in Figure 2.16, the list produced is always `(1 3 5 7 9 11)`.

The traversal is identical, but the methods differ with the first procedure using `append` to join each node whereas the second procedure makes use of a results parameter to adjoin to. Calling `append` results in a loss of efficiency because it traverses the entire first of two lists that it is given. In building the final list, the second procedure simply adjoins lists one node at a time with `cons`, so its steps grow as $\Theta(n)$ because there are `n` many $\Theta(1)$ calls to `cons`. The first procedure will perform just as many `cons`, but for each node it will also call `append`, which for a balanced tree will require a $\Theta(n)$ procedure for half of the number of nodes of that subtree. This means that doubling n (by adding a parent node to the current root, and assuming a new balanced tree) will result in one more `append` call that costs n steps. Therefore, the resulting growth in steps of the first procedure is $\Theta(n \log n)$. This is the order of growth for the `append` calls, as the `cons` calls ($\Theta(n)$) can be dropped as a constant for large values of n. Therefore, the two procedures do not have the same order of growth to convert a balanced tree with n elements to a list, and the first procedure grows more slowly.

Trees from Figure 2.16:








```
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1
                      (right-branch tree))))))
```

For the first tree, this procedure produces the list `(1 3 5 7 9 11)`.

```
(tree->list-1 '(7 (3 (1 () ()) (5 () ())) (9 () (11 () ()))))
```

For brevity, the leaf empty subtrees are omitted until/unless relevant.

```
(append (tree->list-1 '(3 1 5))
        (cons 7 (tree->list-1 '(9 () 11))))
(append (tree->list-1 '(3 1 5))
        (cons 7 (append (tree->list-1 '())
                        (cons 9 (tree->list-1 '(11 () ()))))))
(append (tree->list-1 '(3 1 5))
        (cons 7 (append '() (cons 9 (append (tree->list-1 '())
                                            (cons 11 (tree->list-1 '()))))))))
(append (tree->list-1 '(3 1 5))
        (cons 7 (append '() (cons 9 (append '() (cons 11 ()))))))
(append (tree->list-1 '(3 1 5)) '(7 9 11))
(append (append (tree->list-1 '(1 () ()))
                (cons 3 (tree->list-1 (5 () ()))))
        '(7 9 11))
(append (append (tree->list-1 '(1 () ()))
                (cons 3 (append (tree->list-1 '()) (cons 5 (tree->list-1 '())))))
        '(7 9 11))
(append (append (tree->list-1 '(1 () ()))
                (cons 3 (append (tree->list-1 '()) (cons 5 '()))))
        '(7 9 11)))
(append (append (tree->list-1 '(1 () ()))
                (cons 3 (append '() (cons 5 '()))))
        '(7 9 11))
(append (append (tree->list-1 '(1 () ())) '(3 5)) '(7 9 11))
(append (append (append (tree->list-1 '()) (cons 1 (tree->list-1 '())))
                '(3 5))
        '(7 9 11))
(append (append (append '() (cons 1 '())) '(3 5)) '(7 9 11))
(append (append '(1) '(3 5)) '(7 9 11))
(append '(1 3 5) '(7 9 11))
'(1 3 5 7 9 11)
```

For the second tree, this procedure produces the list `(1 3 5 7 9 11)`. For the third tree, this procedure produces the list `(1 3 5 7 9 11)`.

```
(define (tree->list2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list
                              (right-branch tree)
                              result-list)))))
  (copy-to-list tree '()))
```

For the first tree, this procedure produces the list `(1 3 5 7 9 11)`.

```
(tree->list2 '(7 (3 (1 () ()) (5 () ())) (9 () (11 () ()))))
```

For brevity, the leaf empty subtrees are omitted until/unless relevant.

```
(copy-to-list
  '(3 1 5) (cons 7 (copy-to-list (9 () 11) ())))
(copy-to-list
  '(3 1 5)
  (cons 7
        (copy-to-list '() (cons 9 (copy-to-list (11 () ()) '()))))))
(copy-to-list
  '(3 1 5)
  (cons 7 (copy-to-list '() (cons 9 (copy-to-list '() (cons 11 (copy-to-list '()
'()))))))))
(copy-to-list
  '(3 1 5)
  (cons 7 (copy-to-list '() (cons 9 (copy-to-list '() (cons 11 '()))))))
(copy-to-list
  '(3 1 5) (cons 7 (copy-to-list '() (cons 9 (cons 11 '())))))
(copy-to-list '(3 1 5) (cons 7 (cons 9 (cons 11 '()))))
(copy-to-list
  '(1 () ())
  (cons 3 (copy-to-list '(5 () ()) '(7 9 11))))
(copy-to-list
  '(1 () ())
  (cons 3 (copy-to-list '() (cons 5 (copy-to-list '() '(7 9 11))))))
(copy-to-list
  '(1 () ())
  (cons 3 (copy-to-list '() (cons 5 '(7 9 11)))))
(copy-to-list '(1 () ()) '(3 5 7 9 11))
(copy-to-list '() (cons 1 (copy-to-list '() '(3 5 7 9 11))))
'(1 3 5 7 9 11)
```

For the second tree, this procedure produces the list `(1 3 5 7 9 11)`. For the third tree, this procedure produces the list `(1 3 5 7 9 11)`.