```
package main

import(
        "fmt"
        "container/list"
        "math"
)

var l *list.List

func media(arreglo []float64) float64 {

    l := list.New()
    for i :=0; i<len(arreglo); i++{
        l.PushFront(arreglo[i])
    }

    suma := 0.0
    for e := l.Front(); e != nil; e = e.Next(){
        suma += e.Value.(float64)
    }

    return (suma/float64(len(arreglo)))

}

func desEst(arreglo []float64, media float64) float64 {

    l := list.New()
    for i :=0; i<len(arreglo); i++{
        l.PushFront(arreglo[i])
    }

    desv := 0.0
    for e := l.Front(); e != nil; e = e.Next(){
        desv += math.Pow((e.Value.(float64)-media), 2)
    }
    desv = math.Sqrt(desv/(float64(len(arreglo)-1)))
    return (desv)
}

func main(){

    columna1 := []float64{160,591,114,229,230,270,128,1657,624,1503}
    columna2 := []float64{15.0,69.9,6.5,22.4,28.4,65.9,19.4,198.7,38.8,138.2}
    fmt.Print("\nColumna1 --- La media es y desviación son: ", media(columna1), desEst(columna1,
media(columna1)))
    fmt.Print("\nColumna2 --- La media es y desviación son:", media(columna2), desEst(columna2,
media(columna2)))
```

```
}

package list

type Element struct {
    next, prev *Element
    list *List
    Value interface{}
}


func (e *Element) Next() *Element {
    if p := e.next; p != &e.list.root {
        return p
    }
    return nil
}

func (e *Element) Prev() *Element {
    if p := e.prev; p != &e.list.root {
        return p
        }
        return nil
}

type List struct {
    root Element // sentinel list element, only &root, root.prev, and root.next are used
    len  int     // current list length excluding (this) sentinel element
}


func (l *List) Init() *List {
    l.root.next = &l.root
    l.root.prev = &l.root
    l.len = 0
    return l
}

// New returns an initialized list.
func New() *List { return new(List).Init() }

// Len returns the number of elements of list l.
func (l *List) Len() int { return l.len }

// Front returns the first element of list l or nil
func (l *List) Front() *Element {
    if l.len == 0 {
        return nil
        }
        return l.root.next
}

// Back returns the last element of list l or nil.
func (l *List) Back() *Element {
    if l.len == 0 {
        return nil
        }
        return l.root.prev
}

// lazyInit lazily initializes a zero List value.
func (l *List) lazyInit() {
    if l.root.next == nil {
        l.Init()
        }
}

// insert inserts e after at, increments l.len, and returns e.
func (l *List) insert(e, at *Element) *Element {
    n := at.next
    at.next = e
    e.prev = at
    e.next = n
    n.prev = e
    e.list = l
    l.len++
    return e
}

// insertValue is a convenience wrapper for insert(&Element{Value: v}, at).
func (l *List) insertValue(v interface{}, at *Element) *Element {
    return l.insert(&Element{Value: v}, at)
}

// remove removes e from its list, decrements l.len, and returns e.
func (l *List) remove(e *Element) *Element {
    e.prev.next = e.next
    e.next.prev = e.prev
    e.next = nil // avoid memory leaks
    e.prev = nil // avoid memory leaks
    e.list = nil
    l.len--
    return e
}

// Remove removes e from l if e is an element of list l.
// It returns the element value e.Value.
func (l *List) Remove(e *Element) interface{} {
    if e.list == l {
        // if e.list == l, l must have been initialized when e was inserted
          // in l or l == nil (e is a zero Element) and l.remove will crash
            l.remove(e)
            }
            return e.Value
}

// Pushfront inserts a new element e with value v at the front of list l and returns e.
func (l *List) PushFront(v interface{}) *Element {
    l.lazyInit()
    return l.insertValue(v, &l.root)
}

// PushBack inserts a new element e with value v at the back of list l and returns e.
func (l *List) PushBack(v interface{}) *Element {
    l.lazyInit()
    return l.insertValue(v, l.root.prev)
}

// InsertBefore inserts a new element e with value v immediately before mark and returns e.
// If mark is not an element of l, the list is not modified.
func (l *List) InsertBefore(v interface{}, mark *Element) *Element {
    if mark.list != l {
        return nil
        }
        // see comment in List.Remove about initialization of l
        return l.insertValue(v, mark.prev)
}

// InsertAfter inserts a new element e with value v immediately after mark and returns e.
// If mark is not an element of l, the list is not modified.
func (l *List) InsertAfter(v interface{}, mark *Element) *Element {
    if mark.list != l {
        return nil
        }
        // see comment in List.Remove about initialization of l
        return l.insertValue(v, mark)
}

// MoveToFront moves element e to the front of list l.
// If e is not an element of l, the list is not modified.
func (l *List) MoveToFront(e *Element) {
    if e.list != l || l.root.next == e {
        return
        }
        // see comment in List.Remove about initialization of l
        l.insert(l.remove(e), &l.root)
}
```

```
func (l *List) MoveToBack(e *Element) {
    if e.list != l || l.root.prev == e {
        return
        }
        l.insert(l.remove(e), l.root.prev)
}

func (l *List) PushBackList(other *List) {
    l.lazyInit()
    for i, e := other.Len(), other.Front(); i > 0; i, e = i-1, e.Next() {
         l.insertValue(e.Value, l.root.prev)
         }
}

func (l *List) PushFrontList(other *List) {
    l.lazyInit()
    for i, e := other.Len(), other.Back(); i > 0; i, e = i-1, e.Prev() {
         l.insertValue(e.Value, &l.root)
         }
}
```