```go
package main

import     (
    "fmt"
    "errors"
    "os"
)

func Pop (pstack *[]rune) (rune, error)   {
    if Empty(*pstack)  {
        return 0, errors.New("")
    }  else  {
        c:= (*pstack)[len(*pstack)-1]
        *pstack = (*pstack)[:len(*pstack)-1]
        return c, nil
    }
}

func Push (pstack *[]rune, r rune) {
    *pstack = append(*pstack, r)
}

func Empty(stack []rune) bool  {
    return len(stack) == 0
}

func check(brackets string) bool  {
    var stack []rune
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            Push(&stack, b)
        case ')':
            if top, err:= Pop(&stack); err != nil || top != '('  {
                return false
            }
        case ']':
            if top, err:= Pop(&stack); err != nil || top != '['  {
                return false
            }
        case '}':
            if top, err:= Pop(&stack); err != nil || top != '{'  {
                return false
            }
        default:
            return false
        }
    }
    return Empty(stack)
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

Brackets_01a.go

```
package main

import     (
    "fmt"
    "errors"
    "os"
)

type  (
    node struct {
        bracket rune
        next *node
    }
    stack *node
)

func Pop (ps *stack) (rune, error)   {
    if Empty(*ps)  {
        return 0, errors.New("")
    }  else  {
        c:= (*(*ps)).bracket
        *ps = (*(*ps)).next
        return c, nil
    }
}

func Push (ps *stack, r rune) {
     p:= new(node)
     (*p).bracket = r
     (*p).next = *ps
     *ps = p
}

func Empty(s stack) bool  {
    return s == nil
}

func check(brackets string) bool  {
    var s stack
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            Push(&s, b)
        case ')':
            if top, err:= Pop(&s); err != nil || top != '('  { return false }
        case ']':
            if top, err:= Pop(&s); err != nil || top != '['  { return false }
        case '}':
            if top, err:= Pop(&s); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return Empty(s)
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

Brackets_01b.go

```go
package main

import     (
    "fmt"
    "errors"
    "os"
)

type stack []rune

func (ps *stack) Pop () (rune, error)   {
    if (*ps).Empty()  {
        return 0, errors.New("Pop from empty stack")
    }  else  {
        c:= (*ps)[len(*ps)-1]
        *ps = (*ps)[:len(*ps)-1]
        return c, nil
    }
}

func (ps *stack) Push (r rune) {
    *ps = append(*ps, r)
}

func (s stack) Empty() bool  {
    return len(s) == 0
}

func check(brackets string) bool  {
    var s stack
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            s.Push(b)
        case ')':
            if top, err:= s.Pop(); err != nil || top != '('  { return false }
        case ']':
            if top, err:= s.Pop(); err != nil || top != '['  { return false }
        case '}':
            if top, err:= s.Pop(); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return s.Empty()
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

Brackets_02a.go

```go
package main

import  (
    "fmt"
    "errors"
    "os"
)

type lmnt struct {
    bracket rune
    next *lmnt
}

type stack struct {
    head *lmnt
}

func (s *stack) Push(bracket rune) {
    s.head = &lmnt{bracket, s.head}
}

func (s *stack) Pop() (rune, error) {
    if s.head == nil {
        return 0, errors.New("List is empty")
    }
    c:= s.head.bracket
    s.head = s.head.next
    return c, nil
}

func (s stack) Empty() bool  {
    return s.head == nil
}

func NewStack() stack {
    return stack{nil}
}

func check(brackets string) bool  {
    s := NewStack()
    for _, b := range []rune(brackets)  {
        switch b  {
        case '(', '[', '{':
            s.Push(b)
        case ')':
            if top, err:= s.Pop(); err != nil || top != '('  { return false }
        case ']':
            if top, err:= s.Pop(); err != nil || top != '['  { return false }
        case '}':
            if top, err:= s.Pop(); err != nil || top != '{'  { return false }
        default:
            return false
        }
    }
    return s.Empty()
}

func main() {
    f, err := os.Open("brackets.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var line string
    for  {
        _, err := fmt.Fscanf(f, "%s\n", &line)
        if err !=nil  { break }
        fmt.Println(line, check(line))
    }
}
```

Brackets_02b.go

```go
package main

import     (
    "fmt"
    "errors"
    "os"
    "math"
)

type queue []int

func NewQueue() queue  {
     return make([]int, 0, 0)
}

func (q *queue) Add (n int) {
    *q = append(*q, n)
}

func (q queue) Max () int {
    res := math.MinInt64
    for _, x := range q  {
           if x > res  { res = x }
     }
     return res
}


func (q *queue) Remove () error   {
    if (*q).Empty()  {
        return errors.New("Attempt to remove from empty queue")
    }  else  {
        *q = (*q)[1:]
        return nil
    }
}


func (q queue) Empty() bool  {
    return len(q) == 0
}

func main() {
    f, err := os.Open("numbers.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    q:= NewQueue()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }
        q.Add(x)
    }
    for {
        fmt.Println(q.Max())
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }
        q.Add(x)
        q.Remove()
    }
}
```

Numbers_01.go

```
package main

import      (
    "fmt"
    "errors"
    "os"
    "math"
)

type lmnt struct {
    n int
    next *lmnt
}

type queue struct {
    head *lmnt
    tail *lmnt
}

func NewQueue() queue  {
    return queue{nil, nil}
}

func (q *queue) Add (n int) {
    if (*q).Empty()  {
        (*q).tail = &lmnt{n, nil}
        (*q).head = (*q).tail
    } else {
        (*(*q).tail).next = &lmnt{n, nil}
        (*q).tail = (*(*q).tail).next
    }
}

func (q *queue) Remove () error   {
    if (*q).Empty()  {
        return errors.New("Attempt to remove from empty queue")
    }  else  {
        (*q).head = (*(*q).head).next
        return nil
    }
}

func (q queue) Max () int {
    res := math.MinInt64
    runner:= q.head
    for runner != nil  {
        if (*runner).n > res  { res = (*runner).n }
        runner = (*runner).next
    }
    return res
}

func (q queue) Empty() bool  {
    return q.head == nil
}

func main() {
    f, err := os.Open("numbers.dat")
    if err != nil { return }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    q:= NewQueue()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }
        q.Add(x)
    }
    for {
        fmt.Println(q.Max())
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }
        q.Add(x)
        q.Remove()
    }
}
```

Numbers_02.go