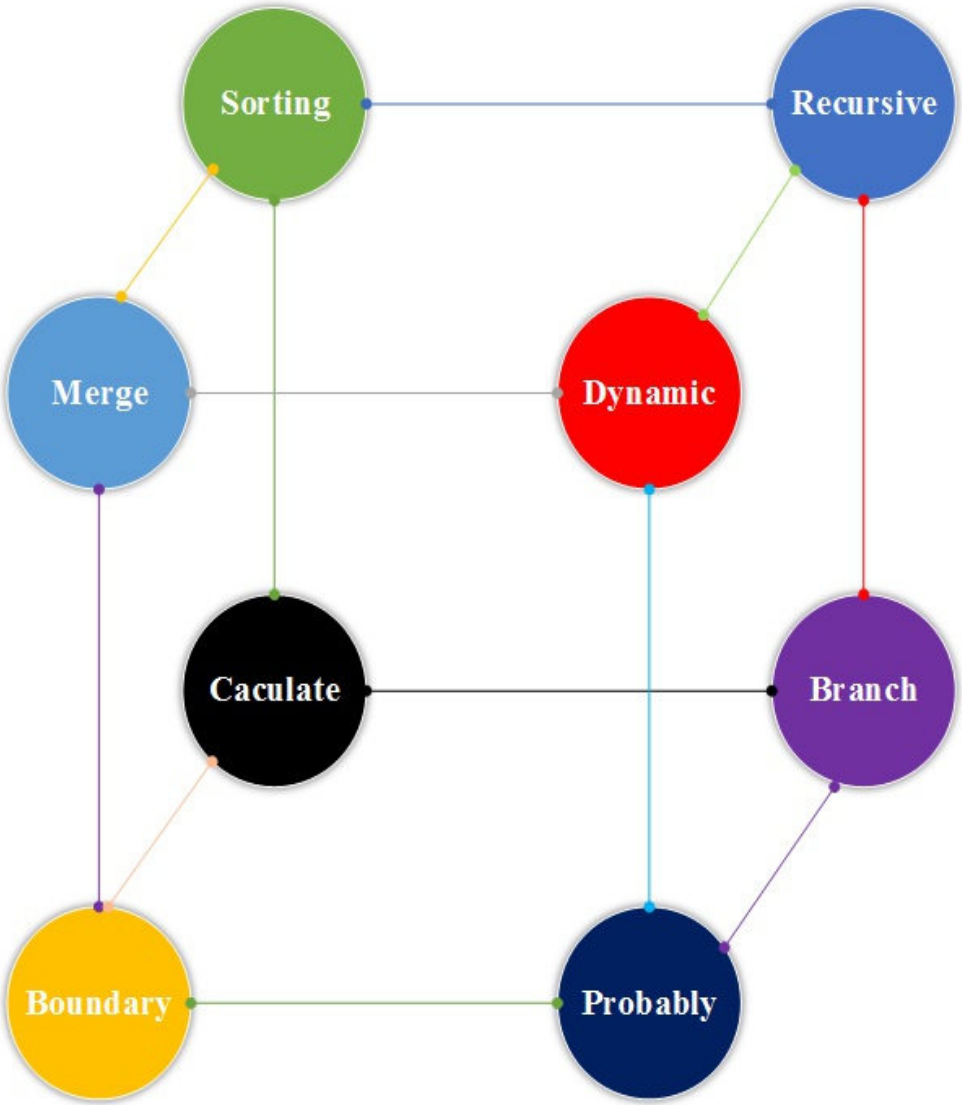
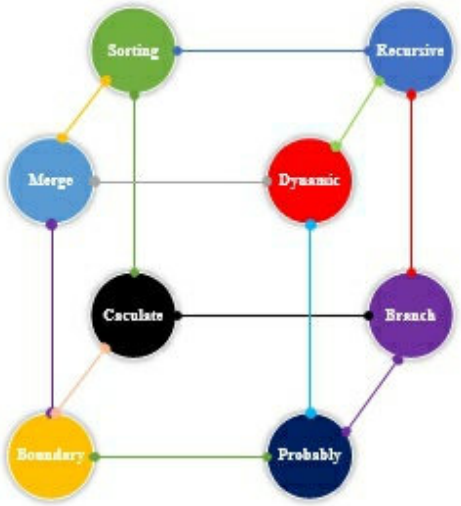
Algorithms Go



Algorithms Explain With Beautiful Pictures

Algorithms Go



YANG HU

Simple is the beginning of wisdom. From the essence of practice,this book to briefly explain the concept and vividly cultivate programming interest, you will learn it easy and fast.

http://en.verejava.com

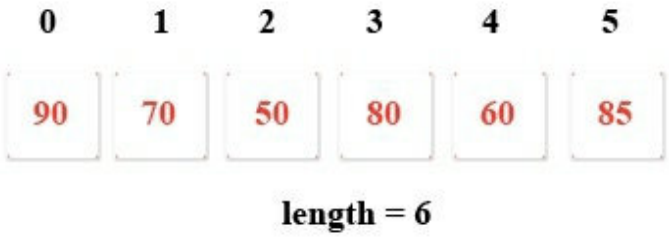
ISBN: 9798679735057

CONTENTS

Linear Table Definition

Linear Table:
Sequence of elements, is a one-dimensional array.

1. Define a one-dimensional array of student scores



TestOneArray.go

```go
package main

import "fmt"

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}

    var length = len(scores)
    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}
```

Result:

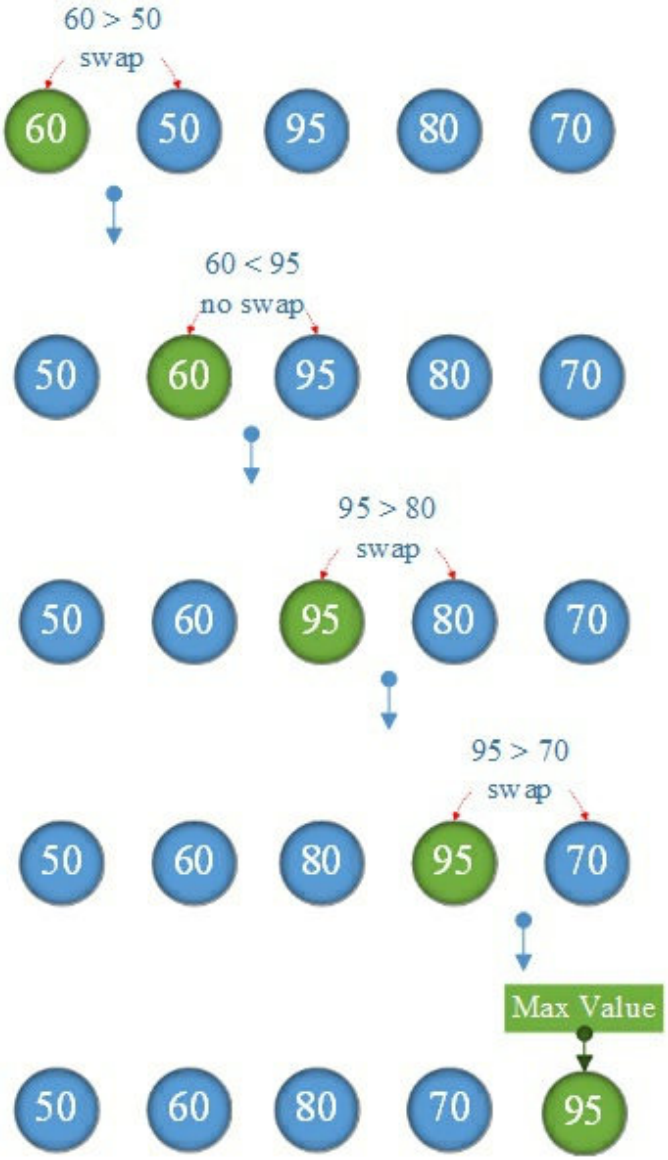
90,70,50,80,60,85,

Maximum Value

Maximum of Integer Sequences:

1. Algorithmic ideas

Compare arrays[i] with arrays[i + 1], if arrays[i] > arrays[i + 1] are exchanged. So continue until the last number, arrays[length - 1] is the maximum.



TestMaxValue.go

```
package main

import "fmt"

func max(arrays []int, length int) int {
    for i := 0; i < length-1; i++ {
        if arrays[i] > arrays[i+1] { // swap
        var temp = arrays[i]
        arrays[i] = arrays[i+1]
        arrays[i+1] = temp
        }
    }
    var maxValue = arrays[length-1]
    return maxValue
}

func main() {
    var scores = []int{60, 50, 95, 80, 70}
    var length = len(scores)
    var maxValue = max(scores, length)
    fmt.Printf("Max Value =  %d\n", maxValue)
}
```

Result:

Max Value = 95

Bubble Sorting Algorithm

Bubble Sorting Algorithm:
Compare arrays[j] with arrays[j + 1], if arrays[j] > arrays[j + 1] are exchanged.
Remaining elements repeat this process, until sorting is completed.

Sort the following numbers from small to large

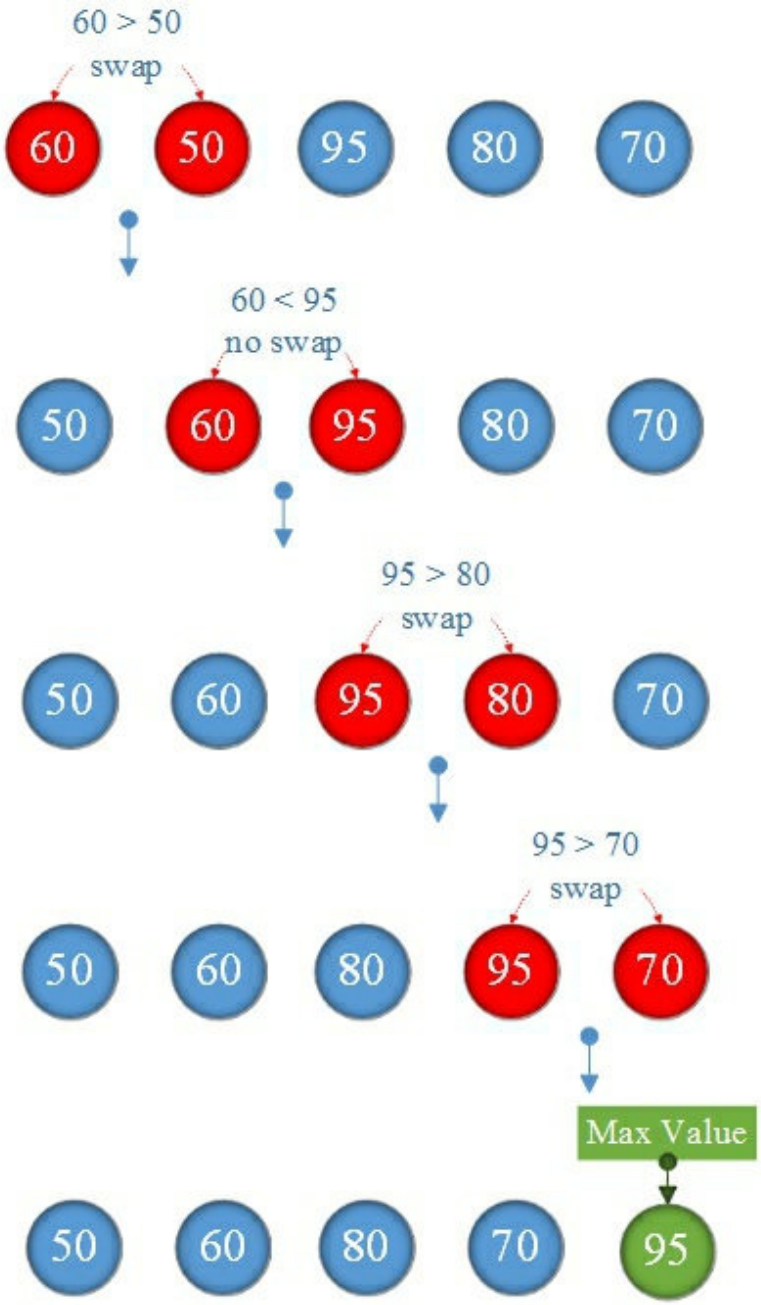
Explanation:


No sorting,

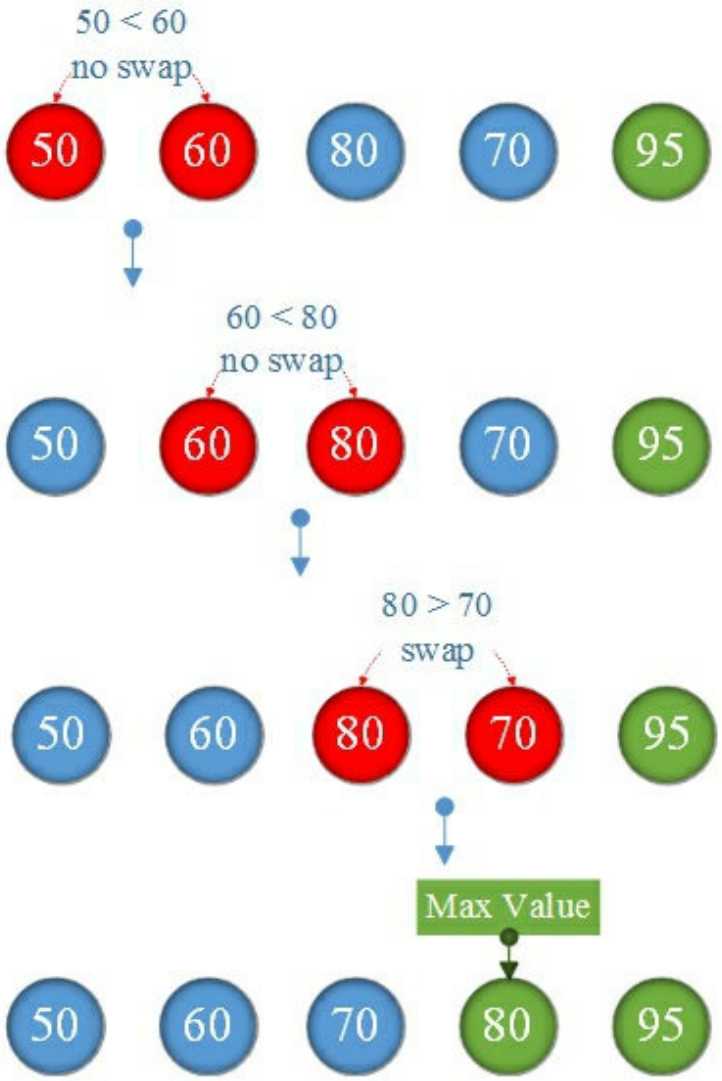

Comparing,


Already sorted

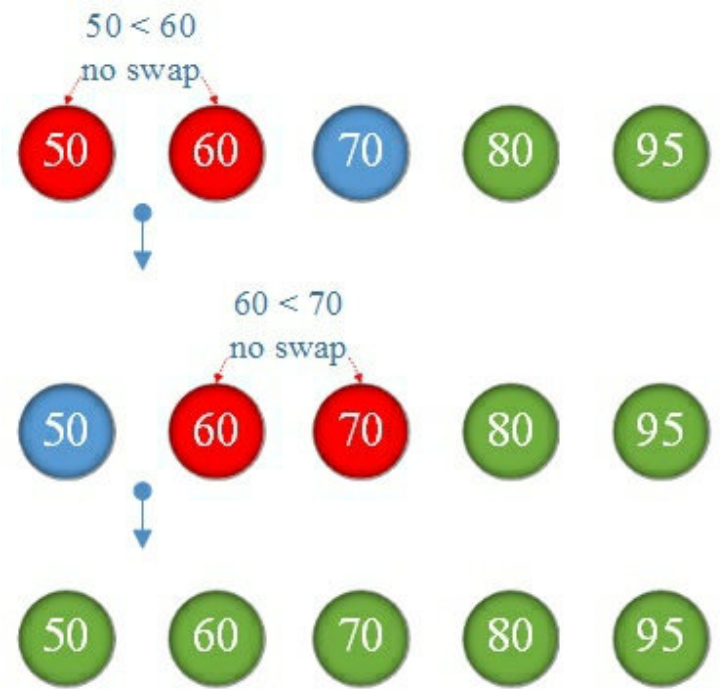
1. **First sorting:**



2. Second sorting:



3. Third sorting:



No swap so terminate sorting : we can get the sorting numbers from small to large

TestBubbleSort.go

```
package main

import "fmt"

func main() {
    // index starts from 0
    var scores = []int{90, 70, 50, 80, 60, 85}
    var length = len(scores)

    sort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}

func sort(arrays []int, length int) {
    for i := 0; i < length-1; i++ {
        for j := 0; j < length-i-1; j++ {
       if arrays[j] > arrays[j+1] { //swap
          var flag = arrays[j]
          arrays[j] = arrays[j+1]
          arrays[j+1] = flag
       }
        }
    }
}
```

Result:

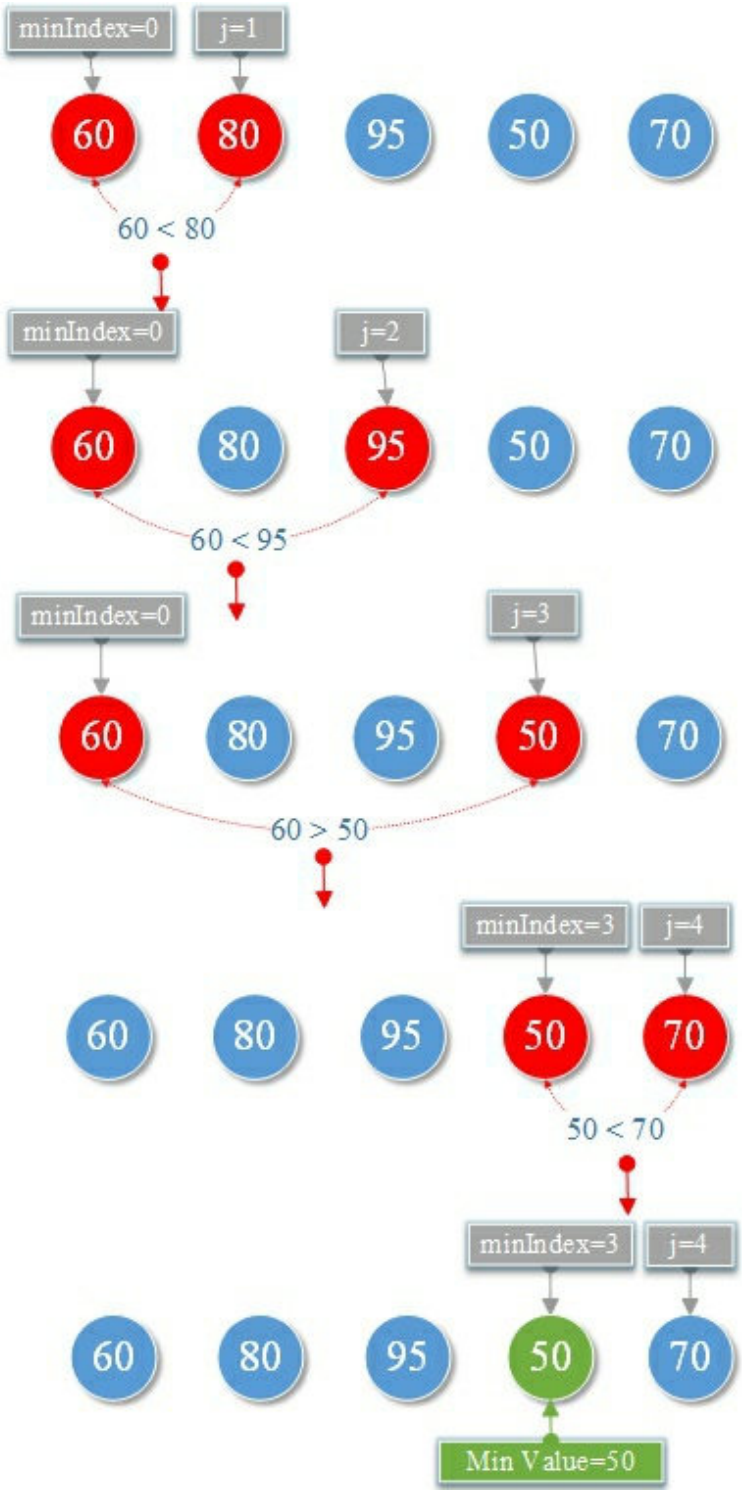
50,60,70,80,85,90,

Minimum Value

Search the Minimum of Integer Sequences:

1. Algorithmic ideas

Initial value minIndex=0, j=1 Compare arrays[minIndex] with arrays[j] if arrays[minIndex] > arrays[j] then minIndex=j, j++ else j++. continue until the last number, arrays[minIndex] is the Min Value.


minIndex=0
j=1
60
80
95
50
70
60 < 80
minIndex=0
j=2
60
80
95
50
70
60 < 95
minIndex=0
j=3
60
80
95
50
70
60 > 50
minIndex=3
j=4
60
80
95
50
70
50 < 70
minIndex=3
j=4
60
80
95
50
70
Min Value=50

TestMinValue.go

```
package main

import "fmt"

func min(arrays []int, length int) int {
    var minIndex = 0 // the index of the minimum
    for j := 1; j < length; j++ {
        if arrays[minIndex] > arrays[j] {
        minIndex = j
        }
    }
    return arrays[minIndex]
}

func main() {
    var scores = []int{60, 80, 95, 50, 70}
    var length = len(scores)
    var minValue = min(scores, length)
    fmt.Printf("Min Value =   %d\n", minValue)
}
```

Result:

Min Value = 50

Select Sorting Algorithm

Select Sorting Algorithm:
Sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

Sort the following numbers from small to large

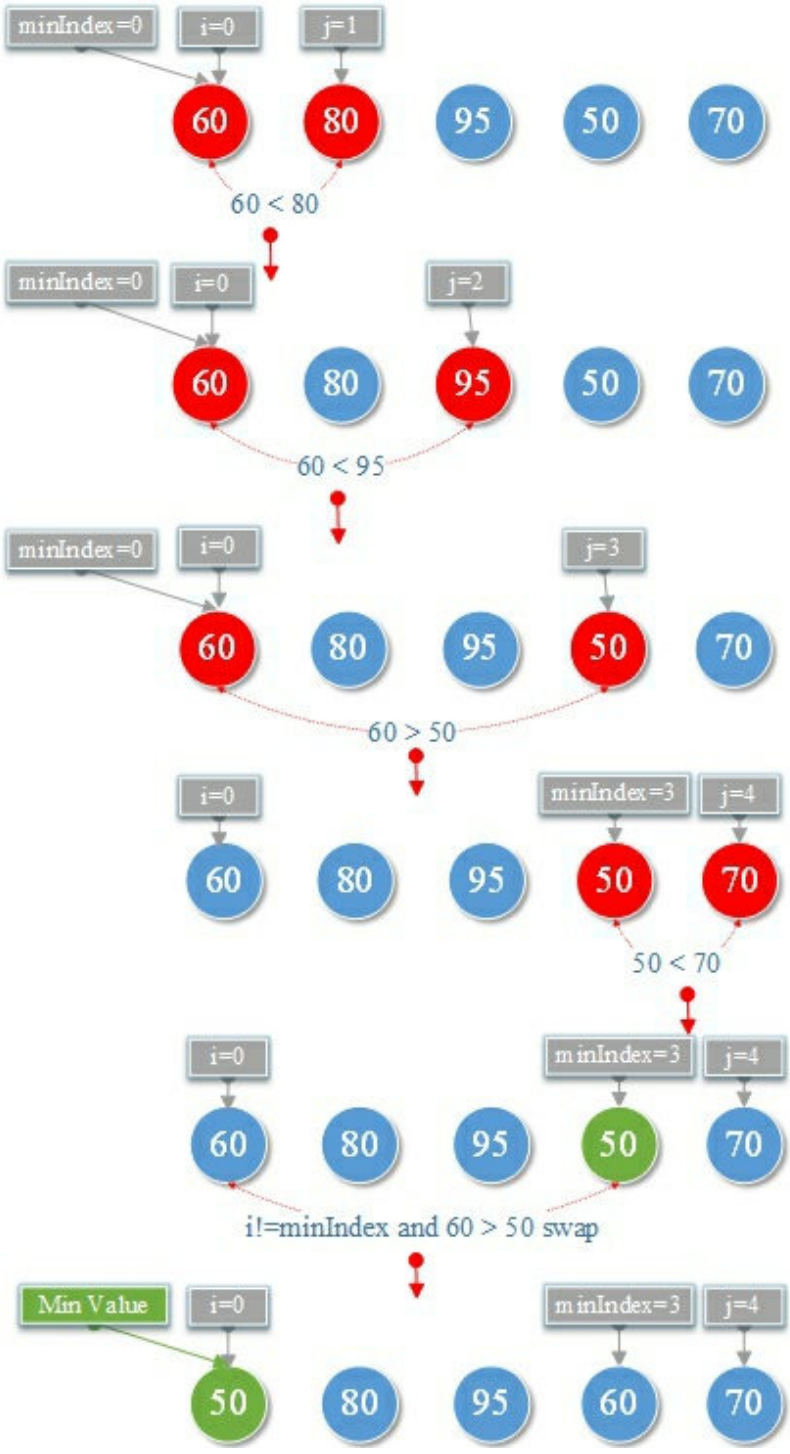
Explanation:


No sorting,

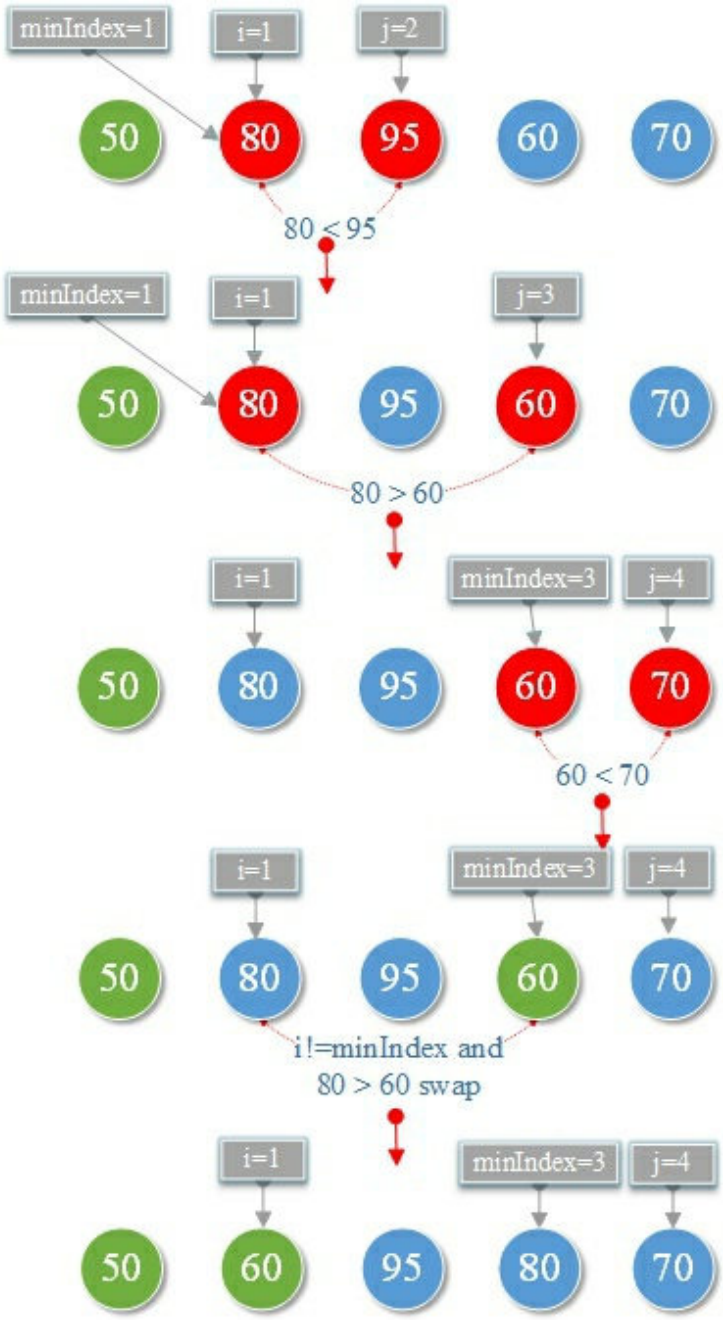

Comparing,


Already sorted.

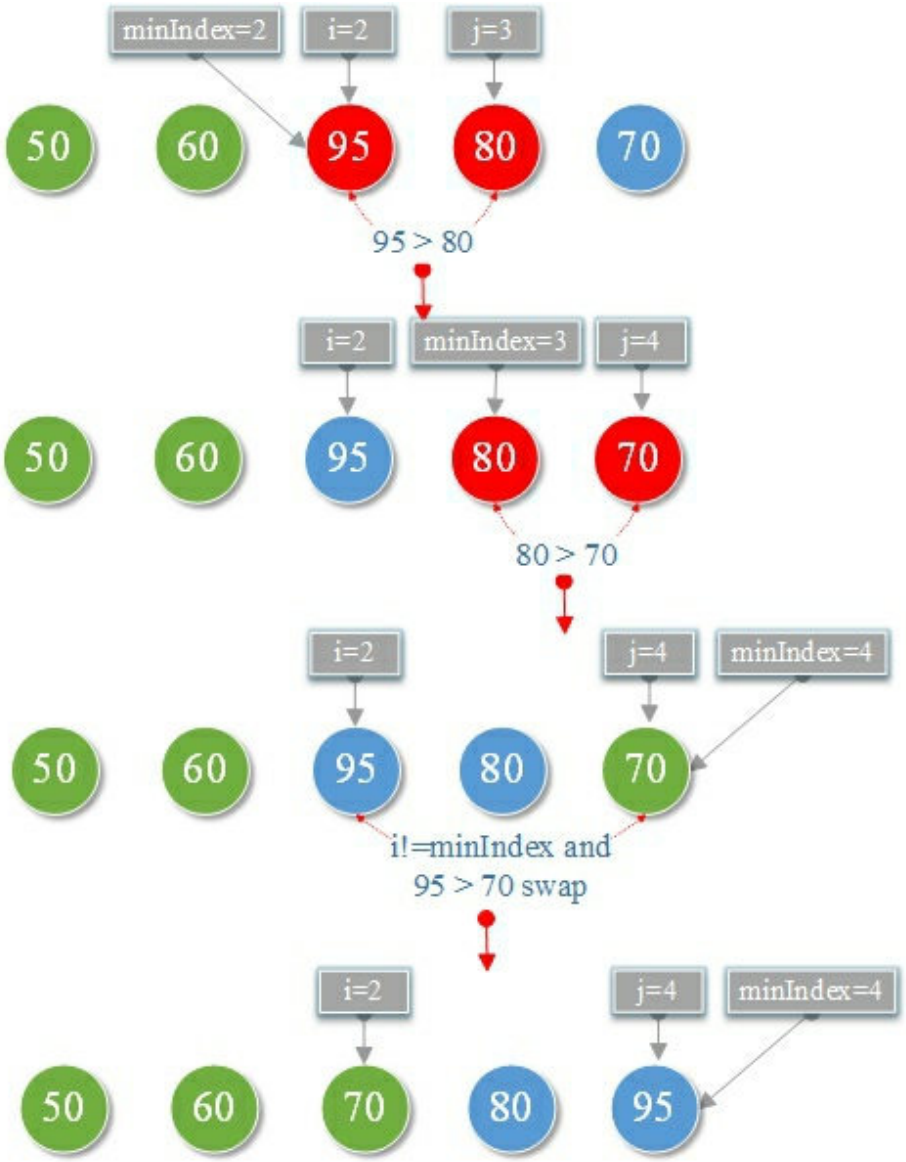
1. **First sorting:**



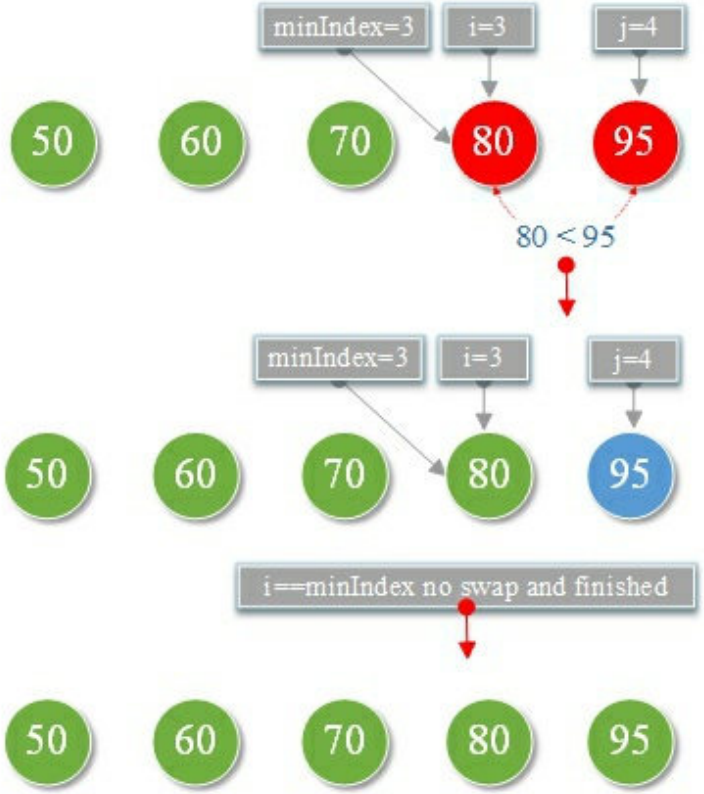
2. Second sorting:



3. Third sorting:



4. Forth sorting:



we can get the sorting numbers from small to large

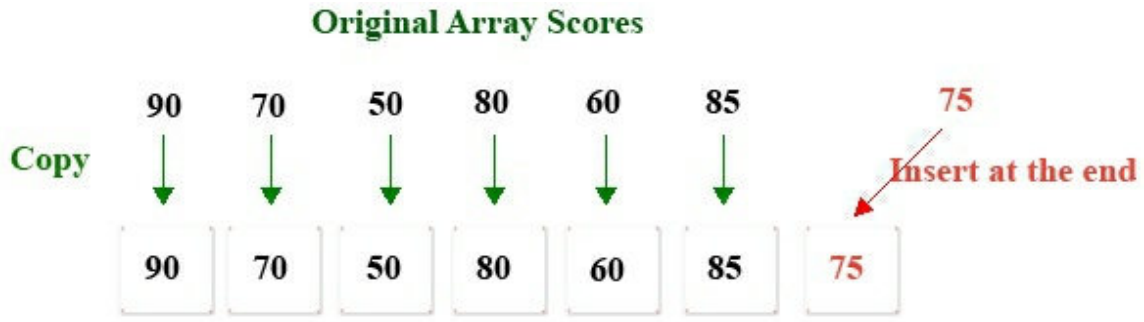
TestSelectSort.go

```
package main
import "fmt"

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}
    var length = len(scores)

    sort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}

func sort(arrays []int, length int) {
    var minIndex int // Save the index of the selected minimum

    for i := 0; i < length-1; i++ {
        minIndex = i
        //Save the minimum value of each loop as the first element
        var minValue = arrays[minIndex]
        for j := i; j < length-1; j++ {
            if minValue > arrays[j+1] {
                minValue = arrays[j+1]
                minIndex = j + 1
            }
        }

        //if minIndex changed, current minimum is exchanged with
minIndex
        if i != minIndex {
            var temp = arrays[i]
            arrays[i] = arrays[minIndex]
            arrays[minIndex] = temp
        }
    }
}
```

Result:
50,60,70,80,85,90,

Linear Table Append

1. Add a score 75 to the end of the one-dimensional array scores.



Analysis:

1. First create a temporary array(tempArray) larger than the original scores array length
2. Copy each value of the scores to tempArray
3. Assign 75 to the last index position of tempArray
4. Finally assign the tempArray pointer reference to the original scores;

TestOneArrayAppend.go

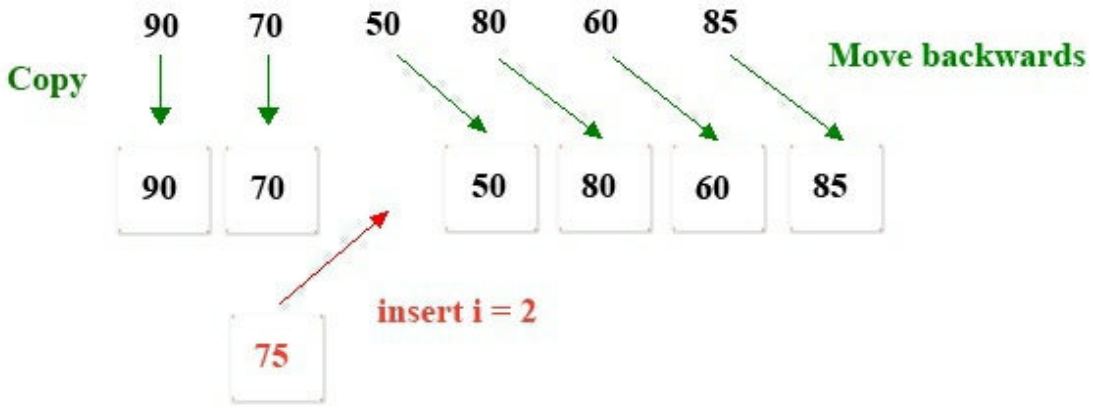
```
package main

import "fmt"

func append(array []int, value int) []int {
    var length = len(array)
    var tempArray = make([]int, length+1) //create a new array

    for i := 0; i < length; i++ {
        tempArray[i] = array[i]
    }
    tempArray[length] = value
    return tempArray
}

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}

    scores = append(scores, 75)

    var length = len(scores)
    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}
```

Result:

90,70,50,80,60,85,75,

Linear Table Insert

1. Insert a student's score anywhere in the one-dimensional array scores.



Analysis:

1. First create a temporary array tempArray larger than the original scores array length
2. Copy each value of the previous value of the scores array from the beginning to the insertion position to tempArray
3. Move the scores array from the insertion position to each value of the last element and move it back to tempArray
4. Then insert the score 75 to the index of the tempArray.
5. Finally assign the tempArray pointer reference to the scores;

TestOneArrayInsert.go

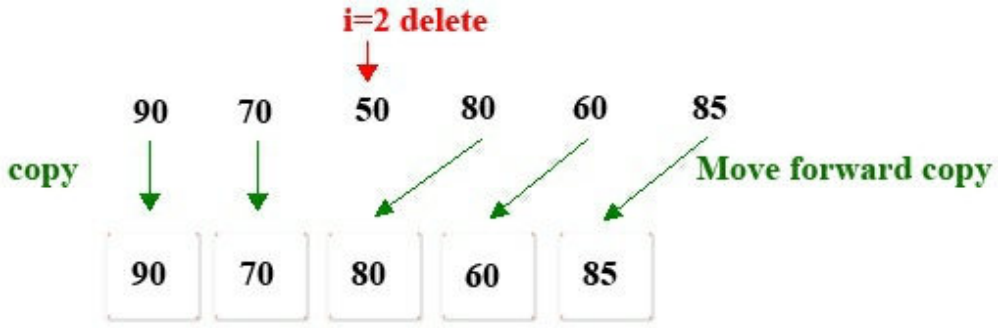
```
package main

import "fmt"

func insert(array []int, length int, tempArray []int, score int, insertIndex
int) {
    for i := 0; i < length; i++ {
        if i < insertIndex {
            tempArray[i] = array[i]
        } else {
            tempArray[i+1] = array[i]
        }
    }
    tempArray[insertIndex] = score
}

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}
    var length = len(scores)
    var tempArray = make([]int, length+1)

    insert(scores, length, tempArray, 75, 2) //Insert 75 into the index = 2

    scores = tempArray

    for i := 0; i < length+1; i++ {
        fmt.Printf("%d,", scores[i])
    }
}
```

Result:

90,70,75,50,80,60,85,

Linear Table Delete

1. **Delete the value of the index=2 from scores array**



Analysis:

1. Create a temporary array tempArray that length smaller than scores by 1.
2. Copy the data in front of i=2 to the front of tempArray
3. Copy the array after i=2 to the end of tempArray
4. Assign the tempArray pointer reference to the scores
5. Printout scores

TestOneArrayDelete.go

```
package main

import "fmt"

func remove(array []int, index int) []int {
    var length = len(array)
    var tempArray = make([]int, length-1) // create a new array
    for i := 0; i < length; i++ {
        if i < index { // Copy data in front of index to the front of tempArray
            tempArray[i] = array[i]
        }
        if i > index { // Copy the array after index to the end of tempArray
            tempArray[i-1] = array[i]
        }
    }
    return tempArray
}

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}
    fmt.Printf("Please enter the index to be deleted: \n")
    var index int
    fmt.Scan(&index)

    scores = remove(scores, index)

    var length = len(scores)
    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}
```

Result:

Please enter the index to be deleted:
2

90,70,80,60,85,

Insert Sorting Algorithm

Insert Sorting Algorithm:
Take an unsorted new element in the array, compare it with the already sorted element before, if the element is smaller than the sorted element, insert new element to the right position.

Sort the following numbers from small to large

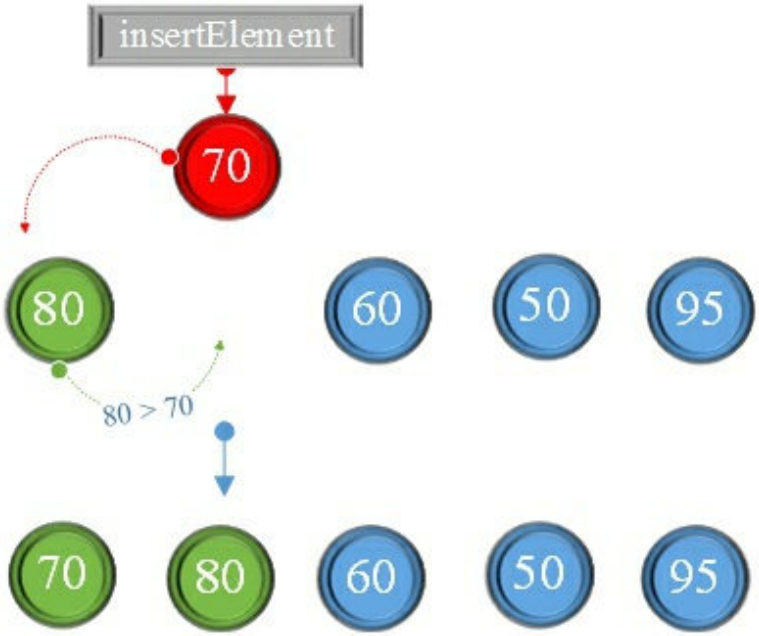
Explanation:


No sorting,

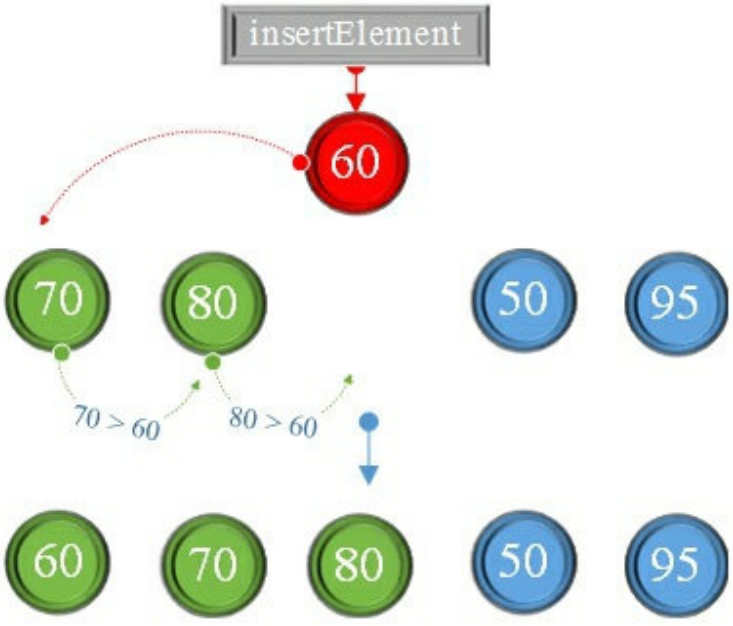

Inserting,


Already sorted

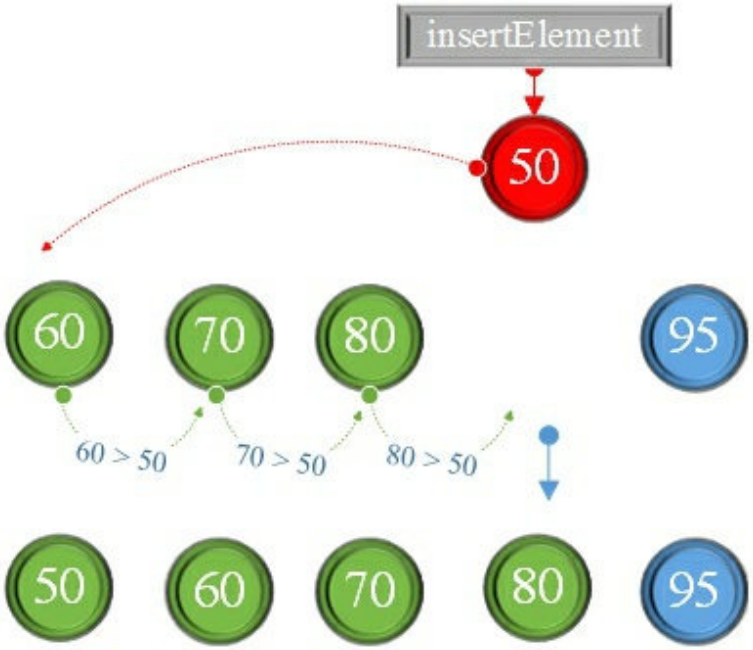
1. **First sorting:**



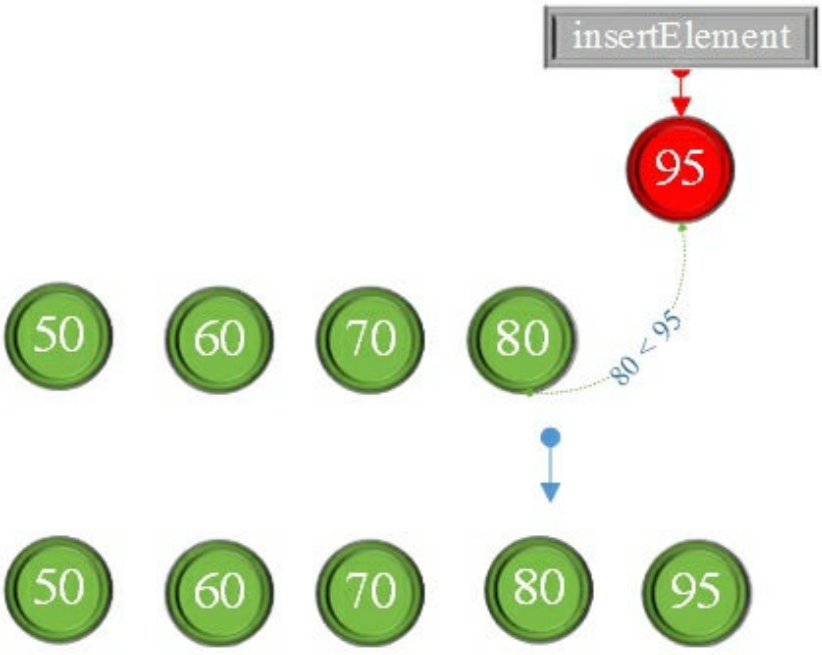
2. **Second sorting:**



3. Third sorting:



4 Third sorting:



TestInsertSort.go

```
package main

import "fmt"

func main() {
    // index starts from 0
    var scores = []int{90, 70, 50, 80, 60, 85}
    var length = len(scores)

    sort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}

func sort(arrays []int, length int) {
    for i := 0; i < length; i++ {
        var insertElement = arrays[i] //Take unsorted new elements
        var insertPosition = i        //Inserted position
        for j := insertPosition - 1; j >= 0; j-- {
            //If the new element is smaller than the sorted element, it is
shifted to the right
            if insertElement < arrays[j] {
                arrays[j+1] = arrays[j]
                insertPosition--
            }
        }
        arrays[insertPosition] = insertElement //Insert the new element
    }
}
```

Result:

50,60,70,80,85,90,

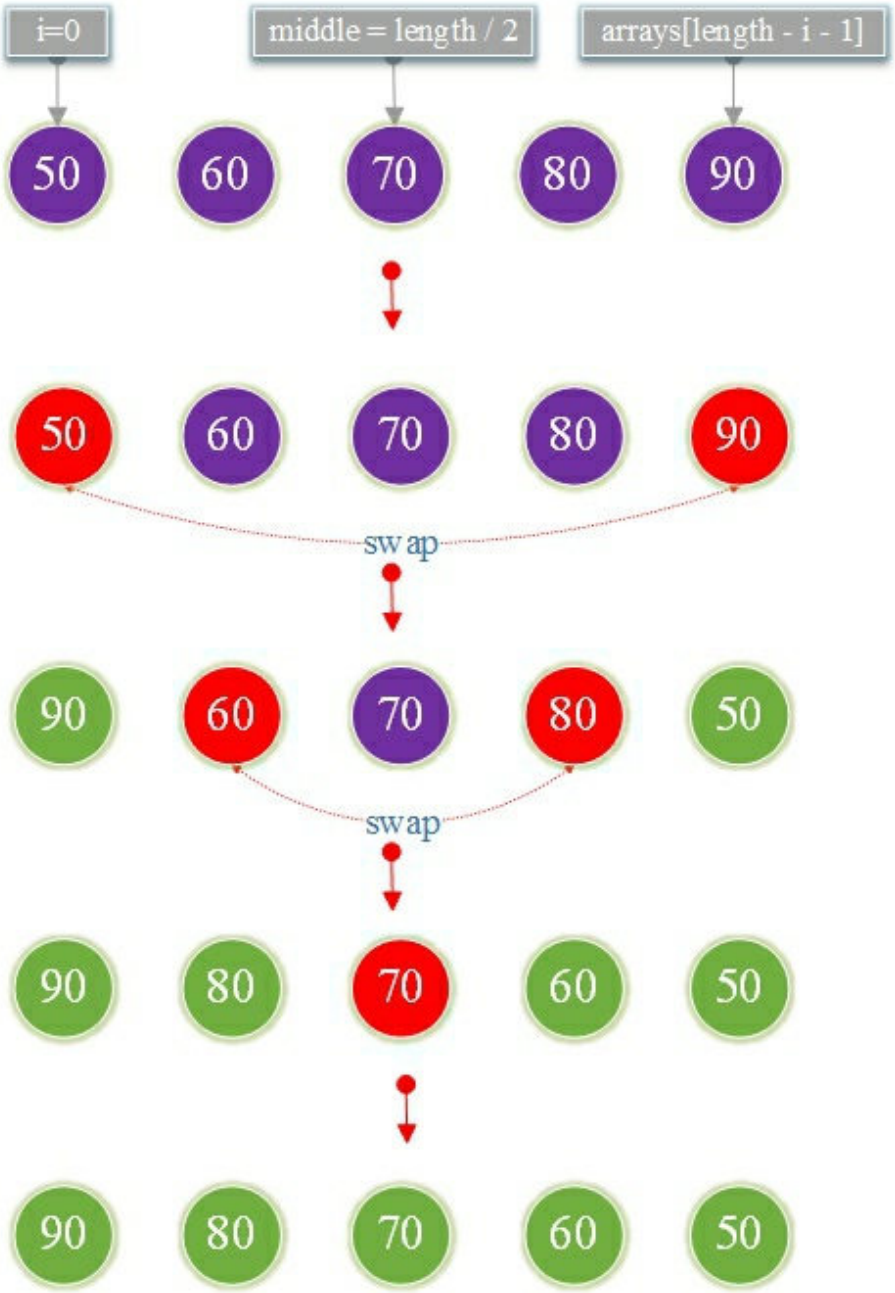
Reverse Array

Inversion of ordered sequences:

1. Algorithmic ideas

Initial i = 0 and then swap the first element arrays[i] with last element arrays[length - i - 1]

Repeat until index of middle i == length / 2.


i=0
middle = length / 2
arrays[length - i - 1]
50
60
70
80
90
50
60
70
80
90
swap
90
60
70
80
50
swap
90
80
70
60
50
90
80
70
60
50

TestReverse.go

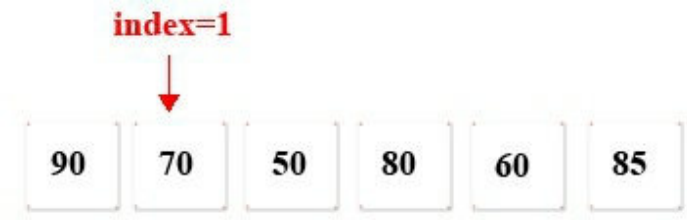
```
package main

import "fmt"

func reverse(arrays []int, length int) {
    var middle = length / 2
    for i := 0; i <= middle; i++ {
        var temp = arrays[i]
        arrays[i] = arrays[length-i-1]
        arrays[length-i-1] = temp
    }
}

func main() {
    var scores = []int{50, 60, 70, 80, 90}
    var length = len(scores)

    reverse(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}
```

Result:

90,80,70,60,50,

Linear Table Search

1. **Please enter the value you want to search like : 70 return index.**



Analysis:

Traverse the value in the array scores, if there is a value equal to the given value like 70, print out the current index

TestOneArraySearch.go

```
package main

import "fmt"

func search(array []int, value int) int {
    var length = len(array)
    for i := 0; i < length; i++ {
        if array[i] == value {
            return i
        }
    }
    return -1
}

func main() {
    var scores = []int{90, 70, 50, 80, 60, 85}
    fmt.Printf("Please enter the value you want to search : \n")
    var value int
    fmt.Scan(&value)

    var index = search(scores, value)

    if index > 0 {
        fmt.Printf("Found value: %d the index is: %d", value, index)
    } else {
        fmt.Printf("The value was not found : %d", value)
    }
}
```

Result:

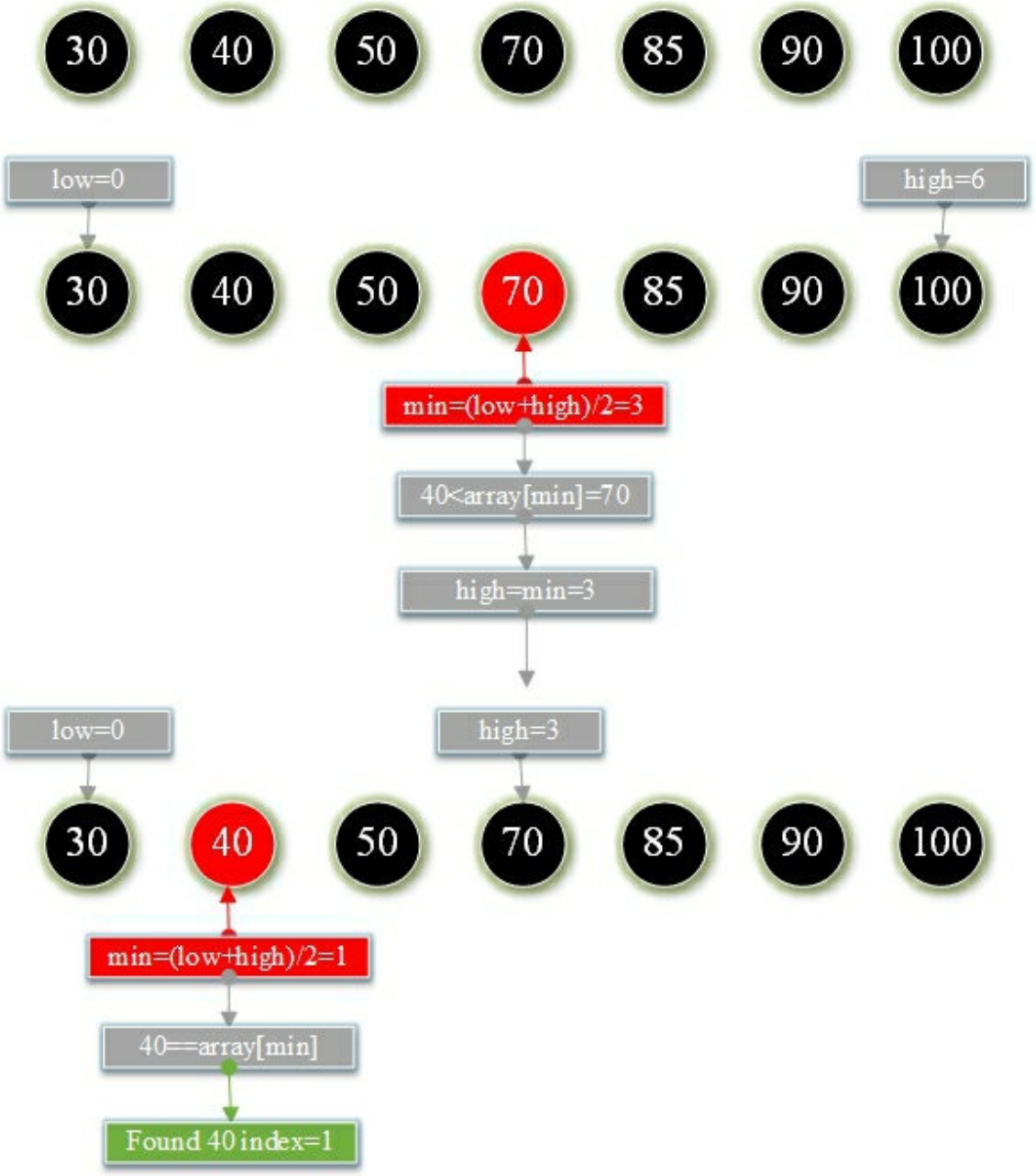
```
Please enter the value you want to search :
70
Found value: 70 the index is: 1
```

Dichotomy Binary Search

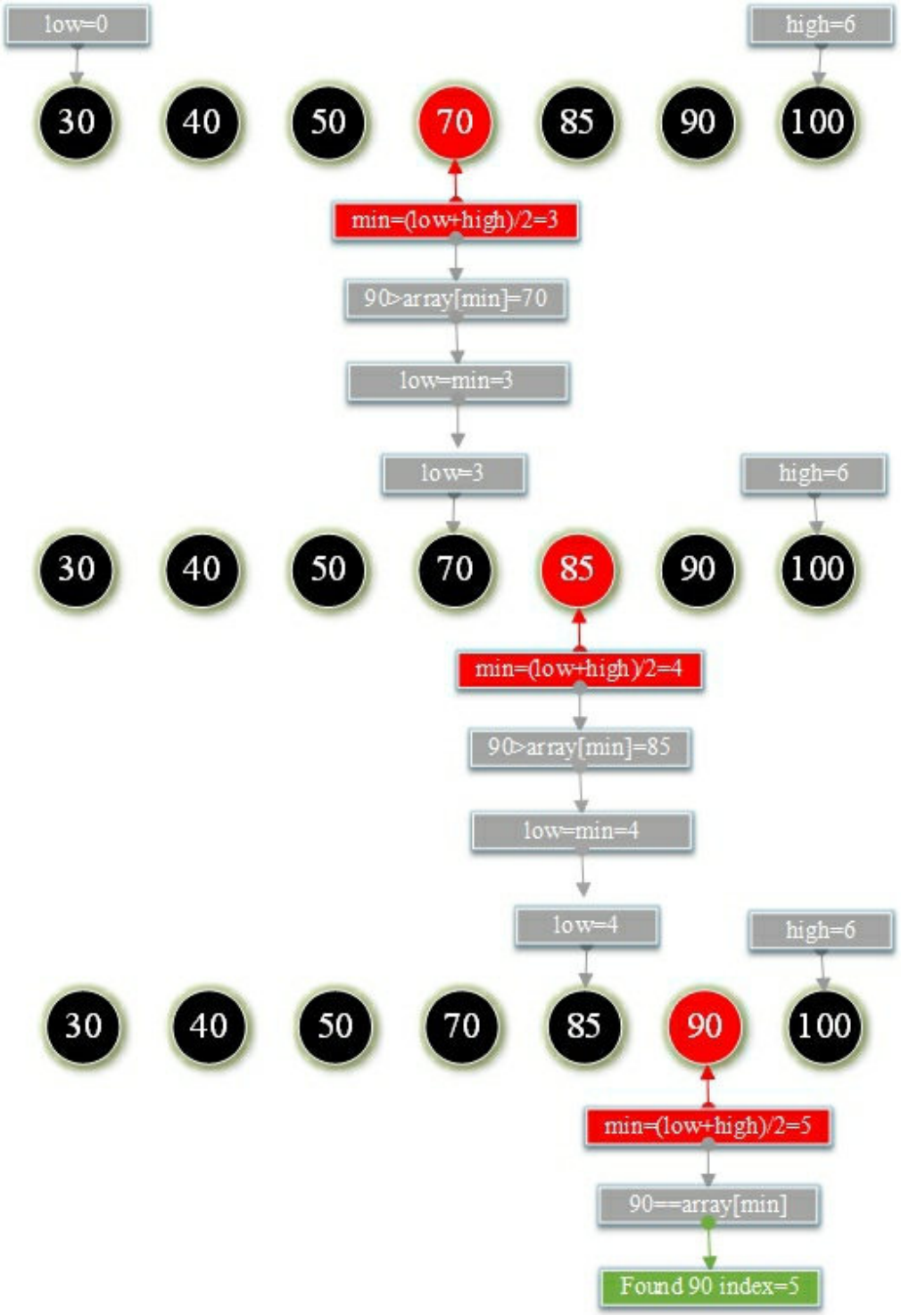
Dichotomy Binary Search:
Find the index position of a given value from an already ordered array.

1. Initialize the lowest index low=0, the highest index high=scores.length-1
2. Find the searchValue of the middle index mid=(low+high)/2 scores[mid]
3. Compare the scores[mid] with searchValue
   If the scores[mid]==searchValue print current mid index,
   If scores[mid]>searchValue that the searchValue will be found between low and mid-1
4. And so on. Repeat step 3 until you find searchValue or low>=high to terminate the loop.

Example 1 : Find the index of searchValue=40 in the array that has been sorted below.



Example 2 : Find the index of searchValue=90 in the array that has been sorted below.



TestBinarySearch.go

```
package main
import "fmt"
func main() {
    var scores = []int{30, 40, 50, 70, 85, 90, 100}
    var length = len(scores)

    var searchValue = 40
    var position = binarySearch(scores, length, searchValue)
    fmt.Printf("%d position : %d", searchValue, position)

    fmt.Printf("\n----------------------------\n")

    searchValue = 90
    position = binarySearch(scores, length, searchValue)
    fmt.Printf("%d position : %d", searchValue, position)
}

func binarySearch(arrays []int, length int, searchValue int) int {
    var low = 0
    var high = length
    var mid = 0

    for {
        if low >= high {
          break
        }
        mid = (low + high) / 2
        if arrays[mid] == searchValue {
          return mid
        } else if arrays[mid] < searchValue {
          low = mid + 1
        } else if arrays[mid] > searchValue {
          high = mid - 1
        }
    }
    return -1
}
```

Result:
40 position:1
---
90 position:5

Shell Sorting

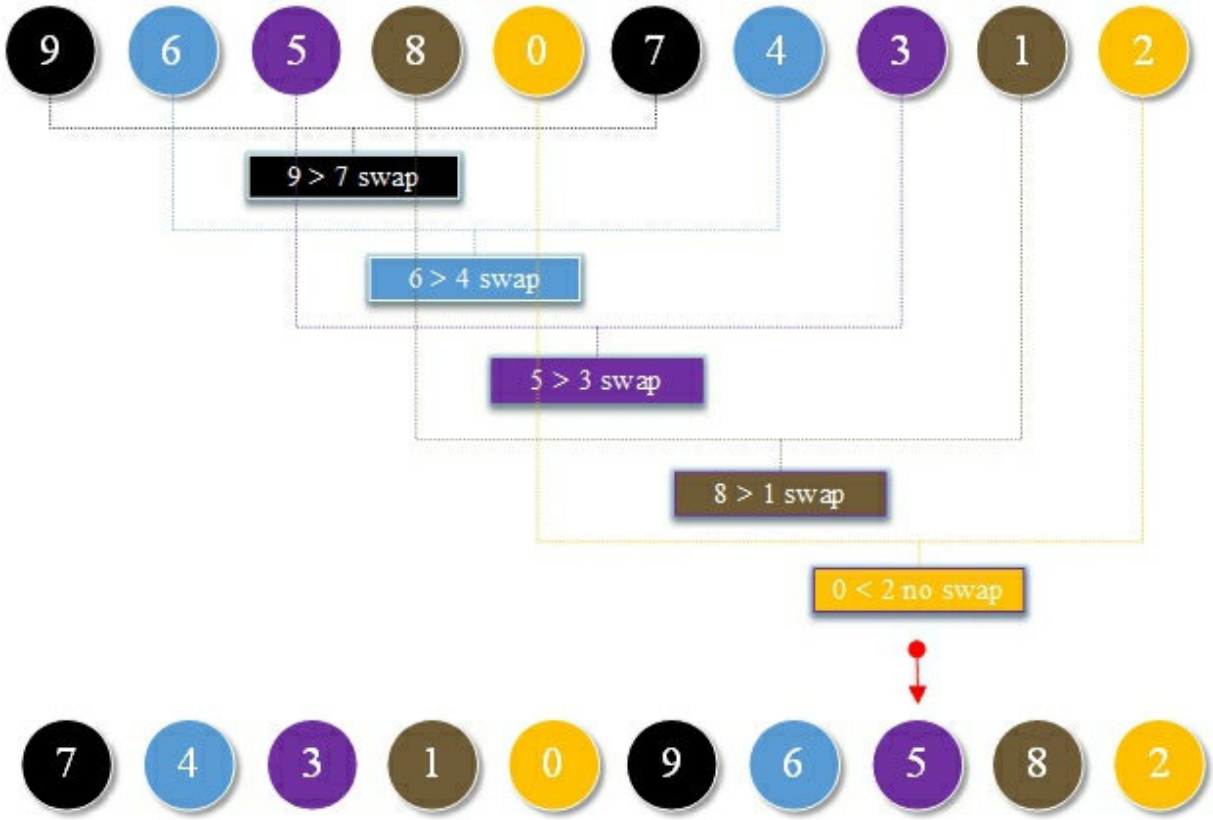
Shell Sorting:
Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Sort the following numbers from small to large by Shell Sorting

Algorithmic result:
The array is grouped according to a certain increment of subscripts, and the insertion of each group is sorted. As the increment decreases gradually until the increment is 1, the whole data is grouped and sorted.

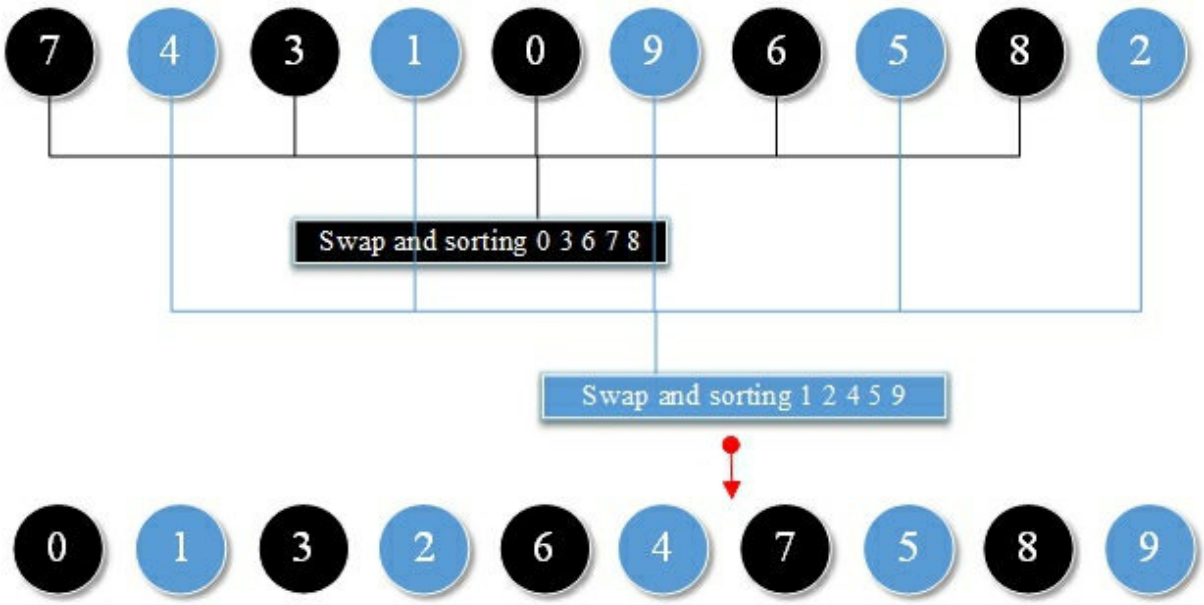
1. **The first sorting :**

   gap = array.length / 2 = 5



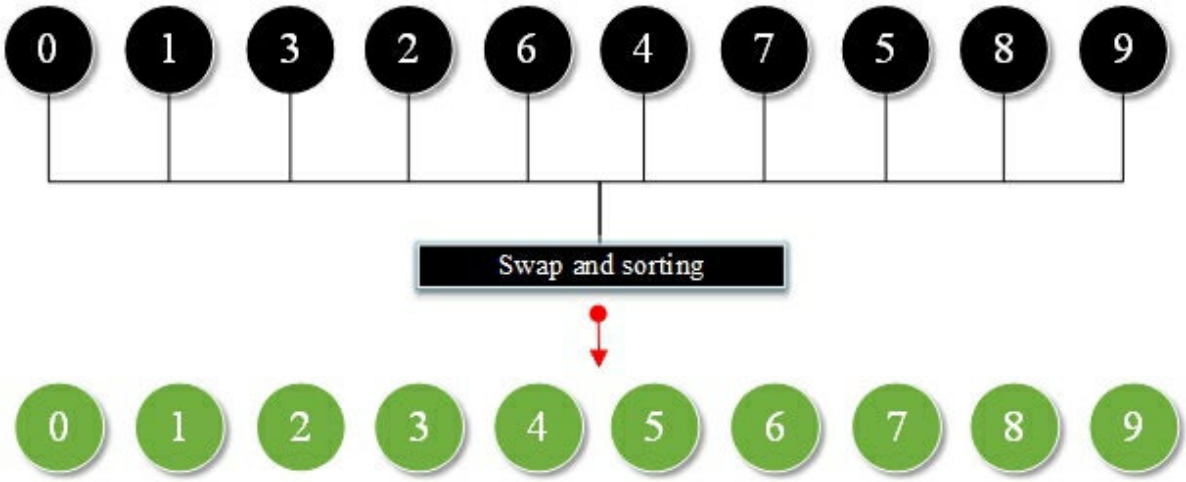
2. The second sorting :

gap = 5 / 2 = 2



3. The third sorting :

gap = 2 / 2 = 1



TestShellSort.go

```
package main

import "fmt"

func swap(array []int, a int, b int) {
    array[a] = array[a] + array[b]
    array[b] = array[a] - array[b]
    array[a] = array[a] - array[b]
}

func shellSort(array []int, length int) {
    for gap := length / 2; gap > 0; gap = gap / 2 {
        for i := gap; i < length; i++ {
        var j = i
        for {
          if j-gap < 0 || array[j] >= array[j-gap] {
            break
          }
          swap(array, j, j-gap)
          j = j - gap
        }
        }
    }
}

func main() {
    var scores = []int{9, 6, 5, 8, 0, 7, 4, 3, 1, 2}
    var length = len(scores)

    shellSort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d, ", scores[i])
    }
}
```

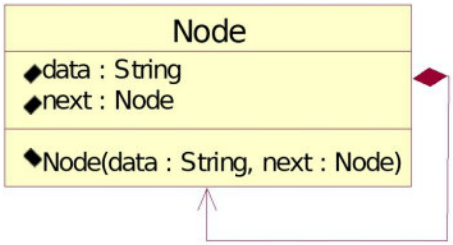
Result:

0,1,2,3,4,5,6,7,8,9,

Unidirectional Linked List

Unidirectional Linked List Single Link:
Is a chained storage structure of a linear table, which is connected by a node. Each node consists of data and next pointer to the next node.

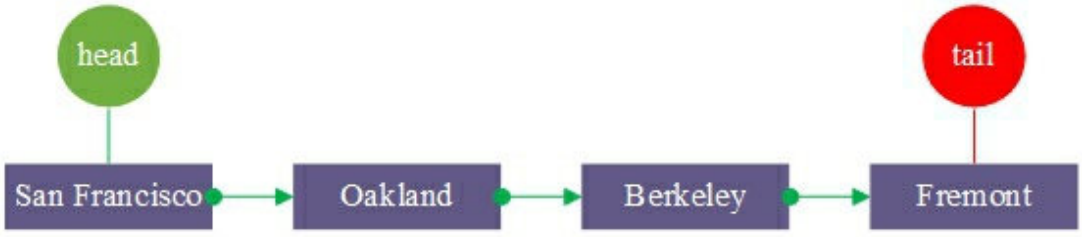
UML Diagram



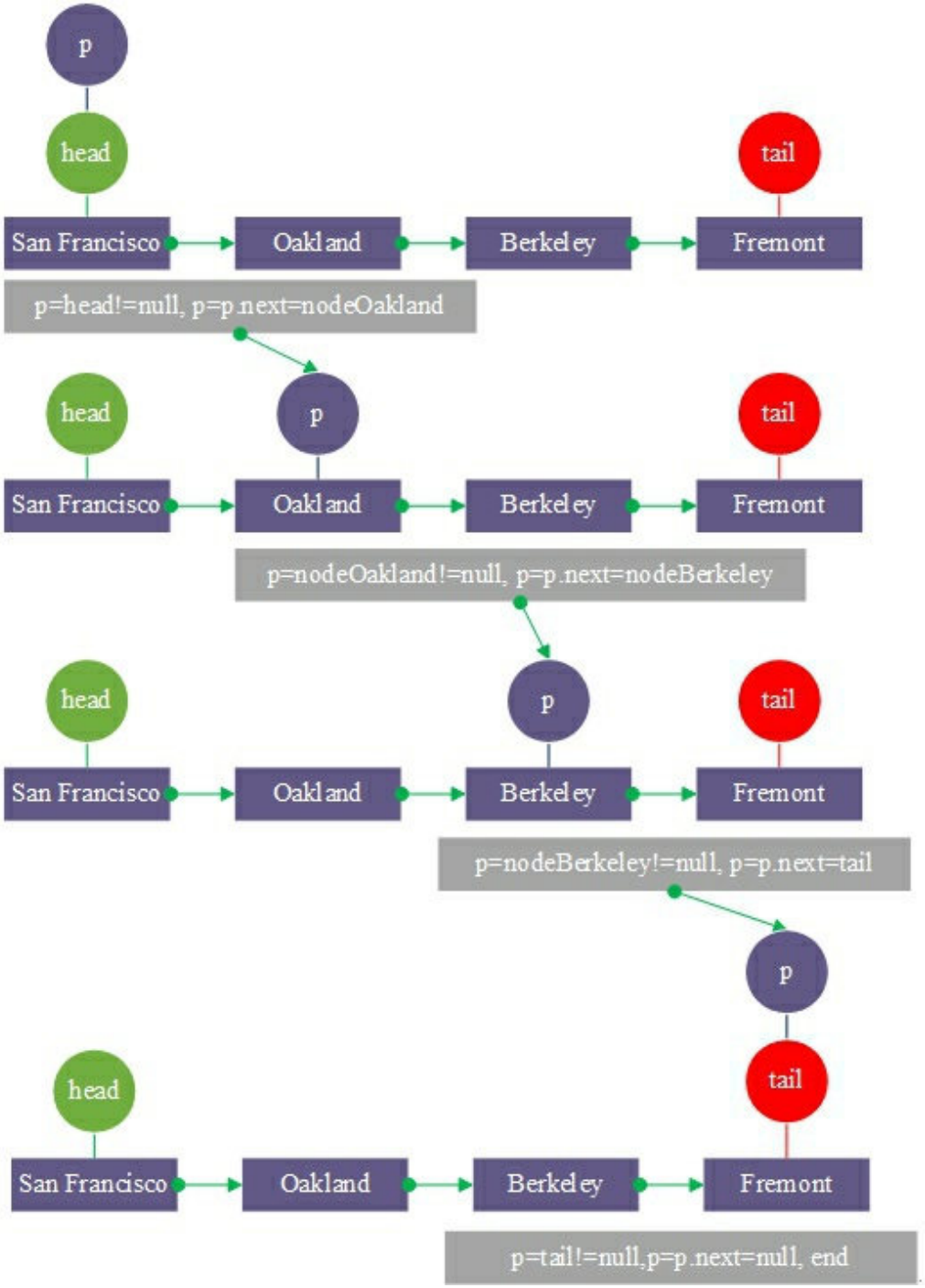

```
type Node struct {
    data string
    next *Node
}
```

1. Unidirectional Linked List initialization.

Example : Construct a San Francisco subway Unidirectional linked list



2. traversal output.



TestUnidirectionalLinkedList.go

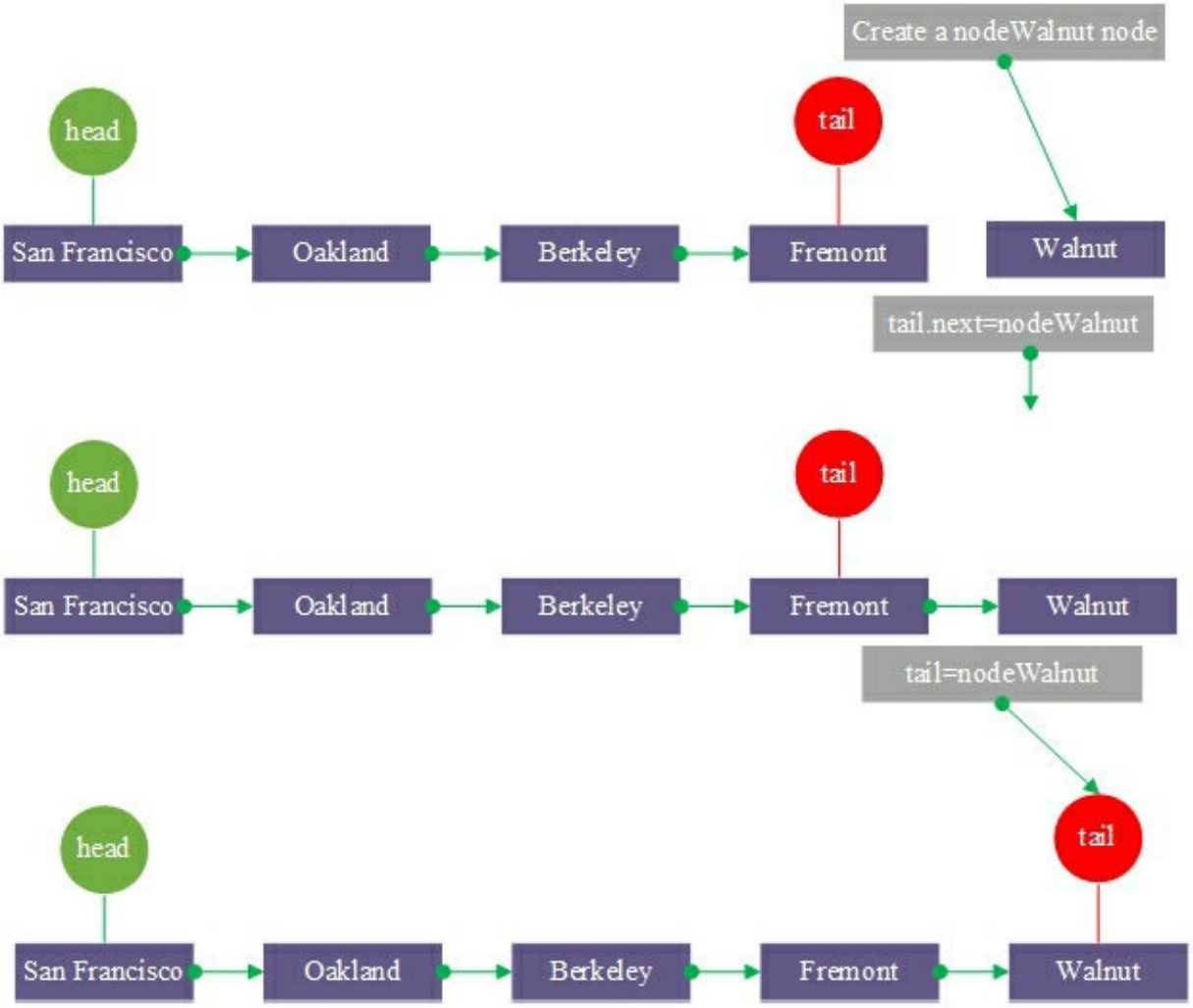
```
package main
import "fmt"

type Node struct {
    data string
    next *Node
}
var head *Node = new(Node) // the first node called head node

func initial() {
    head.data = "San Francisco"
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", next: nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", next: nil}
    nodeOakland.next = nodeBerkeley

    var tail *Node = &Node{data: "Fremont", next: nil}
    nodeBerkeley.next = tail
}

func output(node *Node) {
    var p = node
    for {
        if p == nil {
          break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.next
    }
    fmt.Printf("End\n\n")
}

func main() {
    initial()
    output(head)
```

```
}
```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End

3. Append a new node name: Walnut to the end.



TestUnidirectionalLinkedList.go

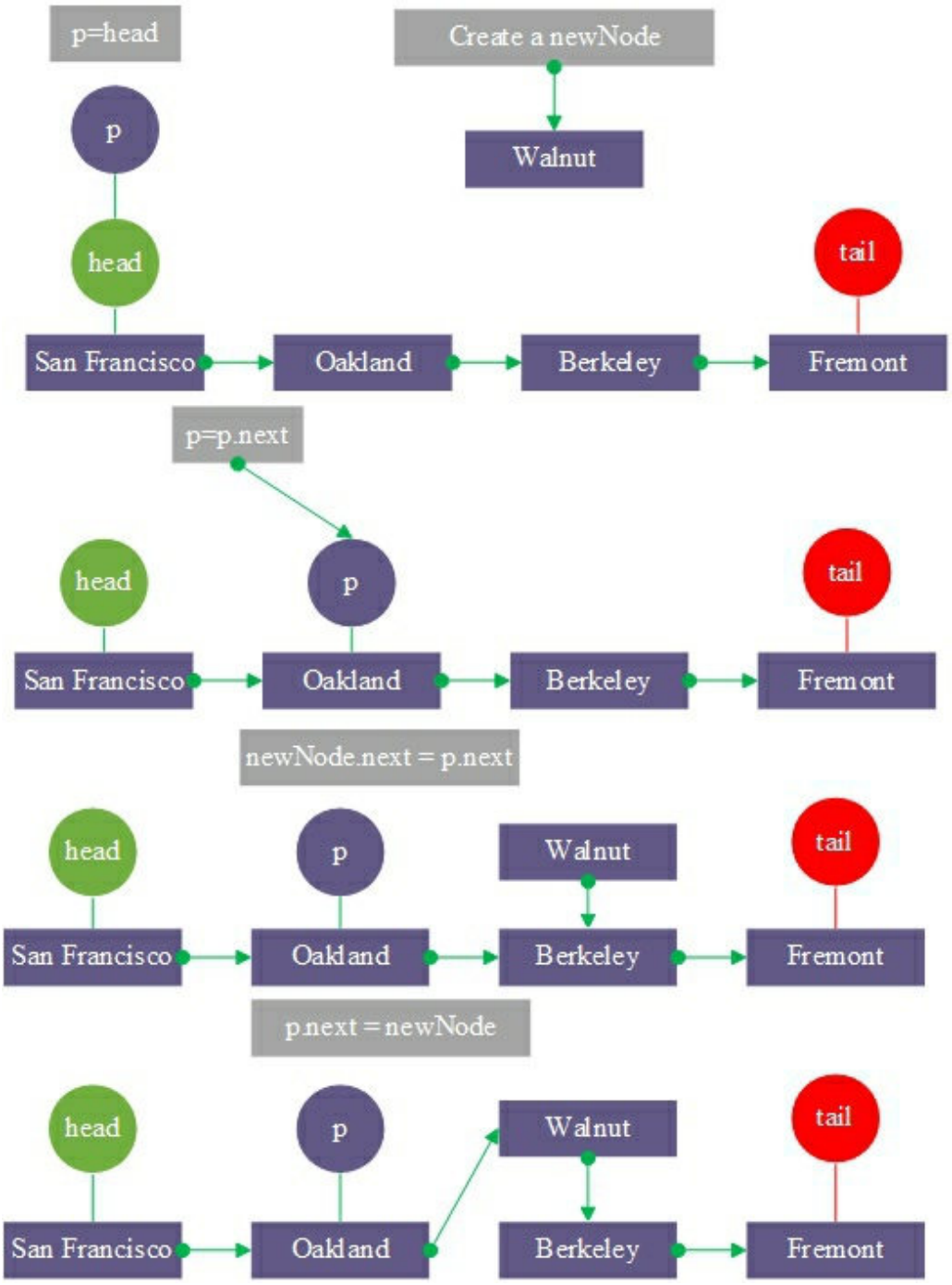
```
package main

import "fmt"

type Node struct {
    data string
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "San Francisco"
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", next: nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.next = nil
    nodeBerkeley.next = tail
}

func add(data string) {
    var newNode *Node = &Node{data: data, next: nil}
    tail.next = newNode
    tail = newNode
}
```

```
func output(node *Node) {
    var p = node

    for {
        if p == nil {
           break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.next
    }
    fmt.Printf("End\n\n")
}

func main() {
    initial()

    fmt.Printf("Append a new node name: Walnut  to the end: \n")
    add("Walnut")

    output(head)
}
```

Result:

Append a new node name: Walnut to the end:
San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End

3. Insert a node Walnut in position 2.



TestUnidirectionalLinkedList.go

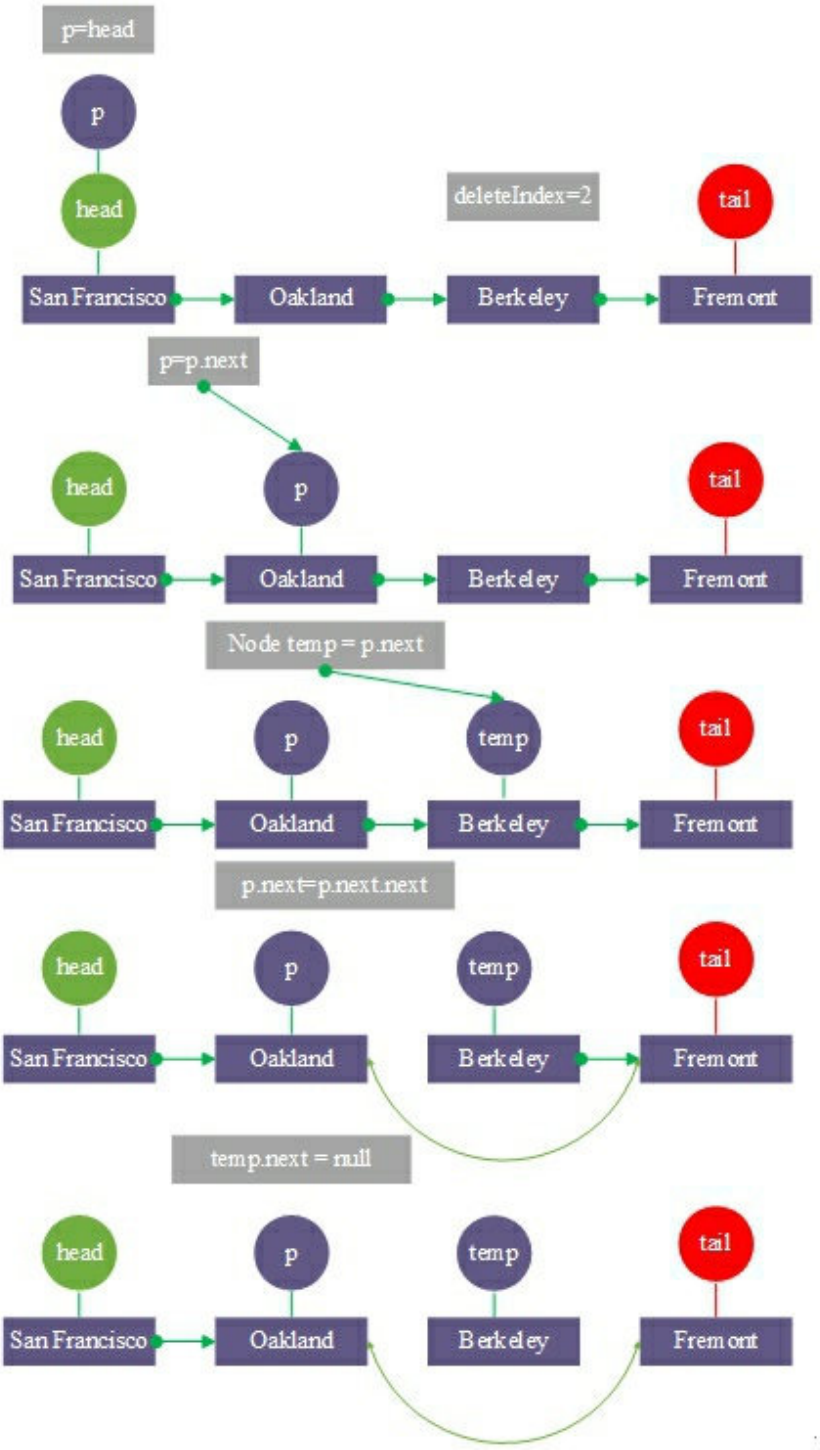
```
package main
import "fmt"
type Node struct {
    data string
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "San Francisco"
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", next: nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.next = nil
    nodeBerkeley.next = tail
}

func insert(insertPosition int, data string) {
    var p = head
    var i = 0
    // Move the node to the insertion position
    for {
        if p.next == nil || i >= insertPosition-1 {
          break
        }
        p = p.next
        i++
    }
    var newNode *Node = new(Node)
```

```
    newNode.data = data
    newNode.next = p.next // newNode next point to next node
    p.next = newNode      // current next point to newNode
}


func output(node *Node) {
    var p = node

    for {
        if p == nil {
          break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.next
    }
    fmt.Printf("End\n\n")
}

func main() {
    initial()

    fmt.Printf("Insert a new node Walnut at index = 2 : \n")
    insert(2, "Walnut")

    output(head)
}
```

Result:

Insert a new node Walnut at index = 2 :
San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End

4. Delete the index=2 node.



TestUnidirectionalLinkedList.go

```
package main
import "fmt"

type Node struct {
    data string
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "San Francisco"
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", next: nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.next = nil
    nodeBerkeley.next = tail
}

func removeNode(removePosition int) {
    var p = head
    var i = 0
    // Move the node to the previous node position that was deleted
    for {
        if p.next == nil || i >= removePosition-1 {
            break
        }
        p = p.next
        i++
    }
    var temp = p.next    // Save the node you want to delete
```

```
    p.next = p.next.next // Previous node next points to next of delete the node
    temp.next = nil
}


func output(node *Node) {
    var p = node

    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.next
    }
    fmt.Printf("End\n\n")
}

func main() {
    initial()

    fmt.Printf("Delete a new node Berkeley at index = 2 : \n")
    removeNode(2)

    output(head)
}
```

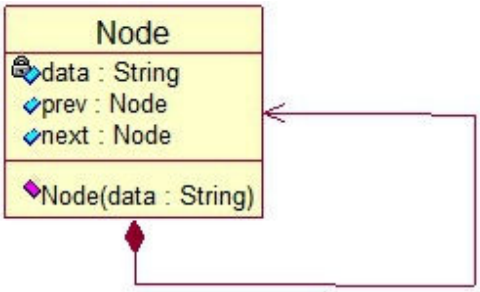
Result:

Delete a new node Berkeley at index = 2 :
San Francisco -> Oakland -> Fremont -> End

Doubly Linked List

Doubly Linked List:
It is a chained storage structure of a linear table. It is connected by nodes in two directions. Each node consists of data, pointing to the previous node and pointing to the next node.

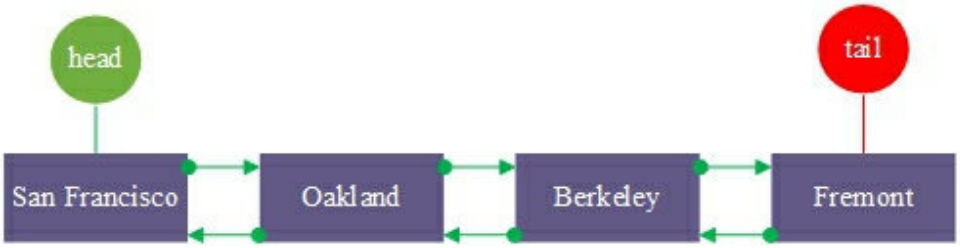
UML Diagram




```
type Node struct {
    data string
    prev *Node
    next *Node
}
```

1. Doubly Linked List initialization.

Example : Construct a San Francisco subway Doubly linked list



2. traversal output. TestDoubleLink.go

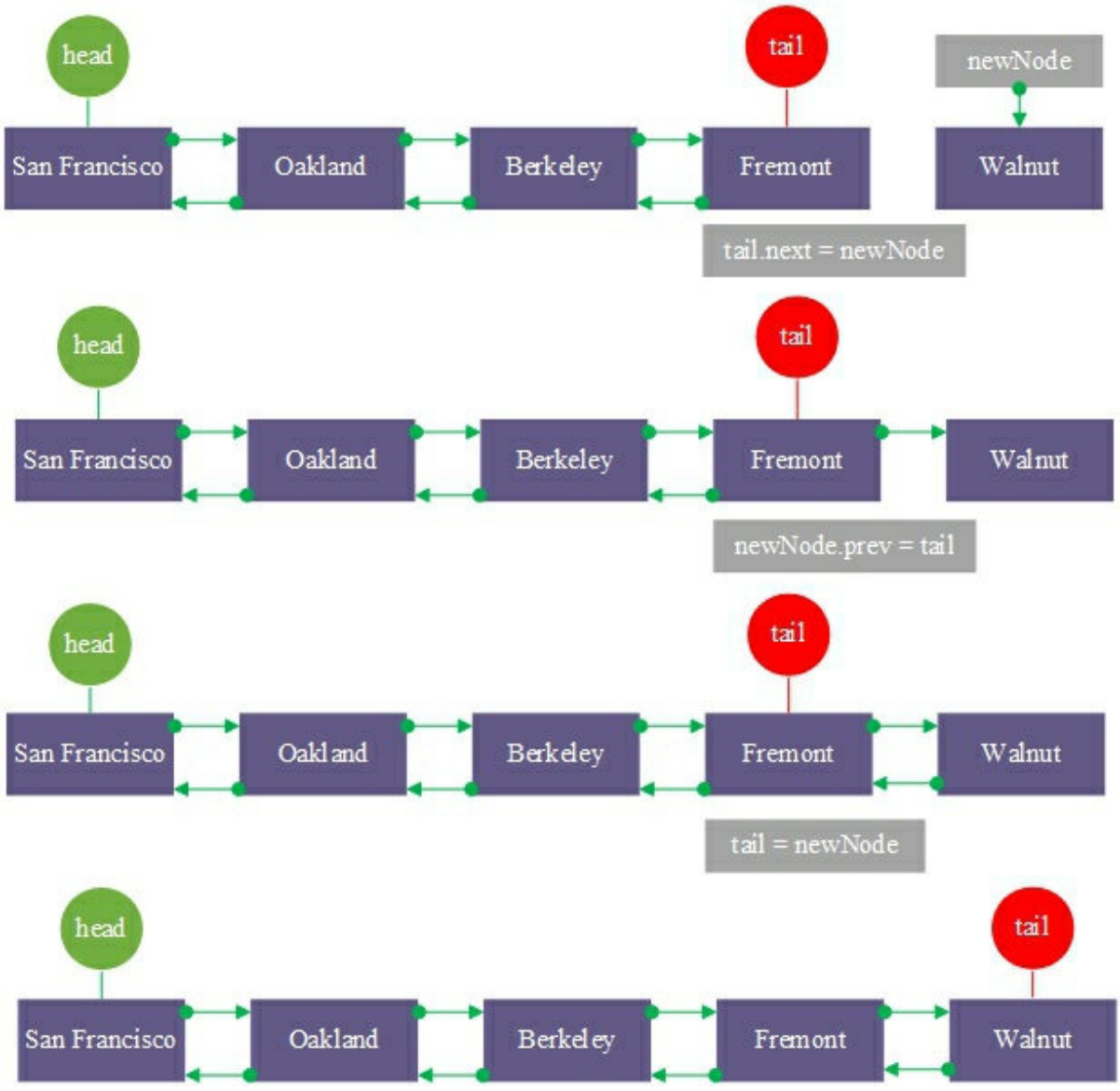
```
package main

import "fmt"

type Node struct {
    data string
    prev *Node
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "San Francisco"
    head.prev = nil
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", prev: head, next:
nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", prev:
nodeOakland, next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.prev = nodeBerkeley
    tail.next = nil
    nodeBerkeley.next = tail
}
```

```
func output(node *Node) {
    var p = node
    var end *Node = nil
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        end = p
        p = p.next
    }
    fmt.Printf("End\n")

    p = end
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.prev
    }
    fmt.Printf("Start\n\n")
}

func main() {
    initial()
    output(head)
}
```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End
Fremont -> Berkeley -> Oakland -> San Francisco -> Start

3. add a node Walnut at the end of Fremont.



TestDoubleLink.go

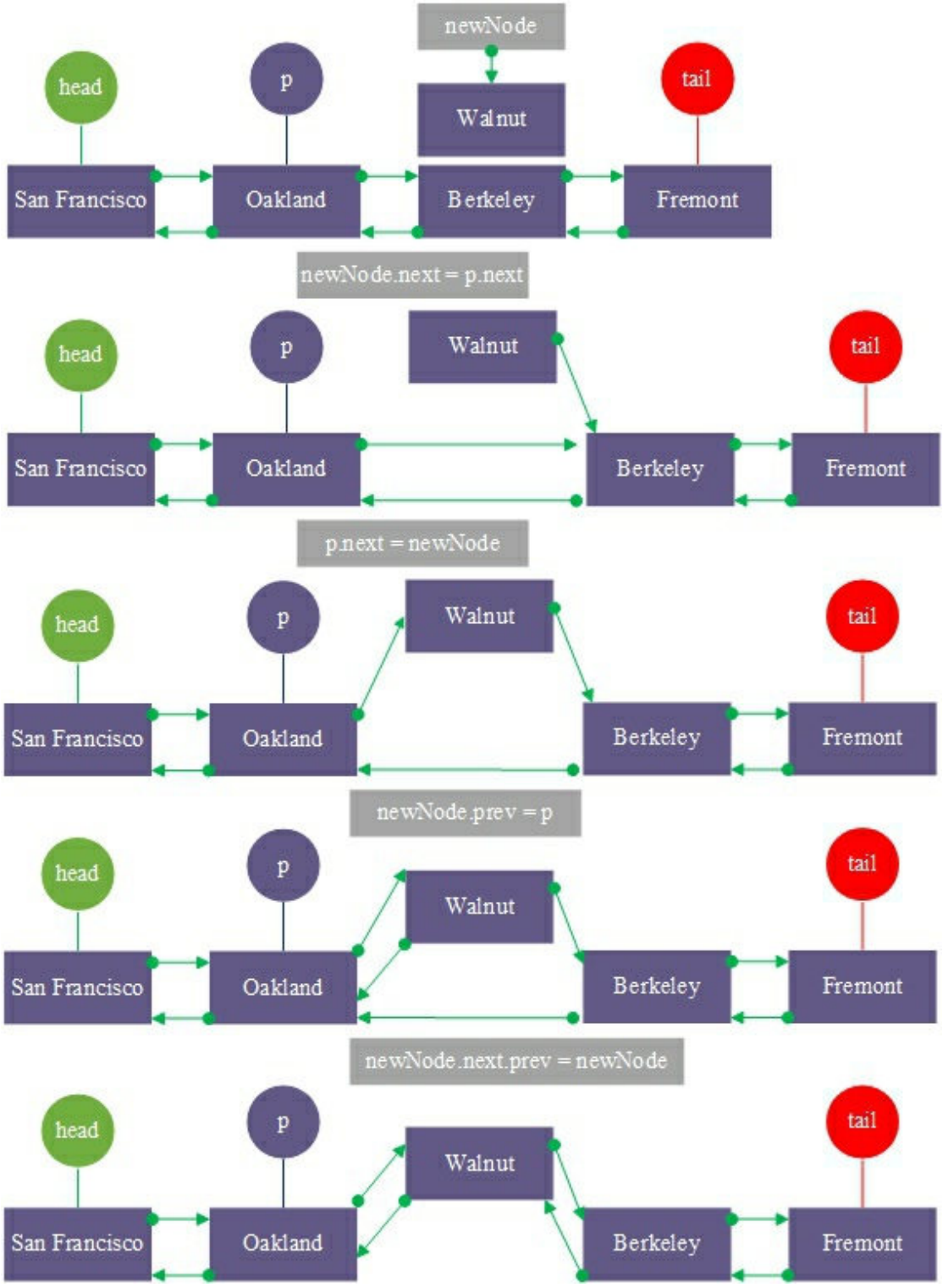
```
package main

import "fmt"

type Node struct {
    data string
    prev *Node
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "San Francisco"
    head.prev = nil
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", prev: head, next:
nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", prev:
nodeOakland, next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.prev = nodeBerkeley
    tail.next = nil
    nodeBerkeley.next = tail
}

func add(data string) {
    var newNode *Node = new(Node)
    newNode.data = data
    newNode.next = nil
    tail.next = newNode
```

```
    newNode.prev = tail
    tail = newNode
}



func output(node *Node) {
    var p = node
    var end *Node = nil

    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        end = p
        p = p.next
    }
    fmt.Printf("End\n")

    p = end
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.prev
    }
    fmt.Printf("Start\n\n")
}

func main() {
    initial()

    fmt.Printf("Add a new node Walnut : \n")
    add("Walnut")

    output(head)
}
```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End
Walnut -> Fremont -> Berkeley -> Oakland -> San Francisco -> Start

3. Insert a node Walnut in position 2.



TestDoubleLink.go

```
package main
import "fmt"
type Node struct {
    data string
    prev *Node
    next *Node
}
var head *Node = new(Node)
var tail *Node = new(Node)
func initial() {
    head.data = "San Francisco"
    head.prev = nil
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", prev: head, next:
nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", prev:
nodeOakland, next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.prev = nodeBerkeley
    tail.next = nil
    nodeBerkeley.next = tail
}

func insert(insertPosition int, data string) {
    var p = head
    var i = 0
    for {
        if p.next == nil || i >= insertPosition-1 {
            break
        }
        p = p.next
        i++
```

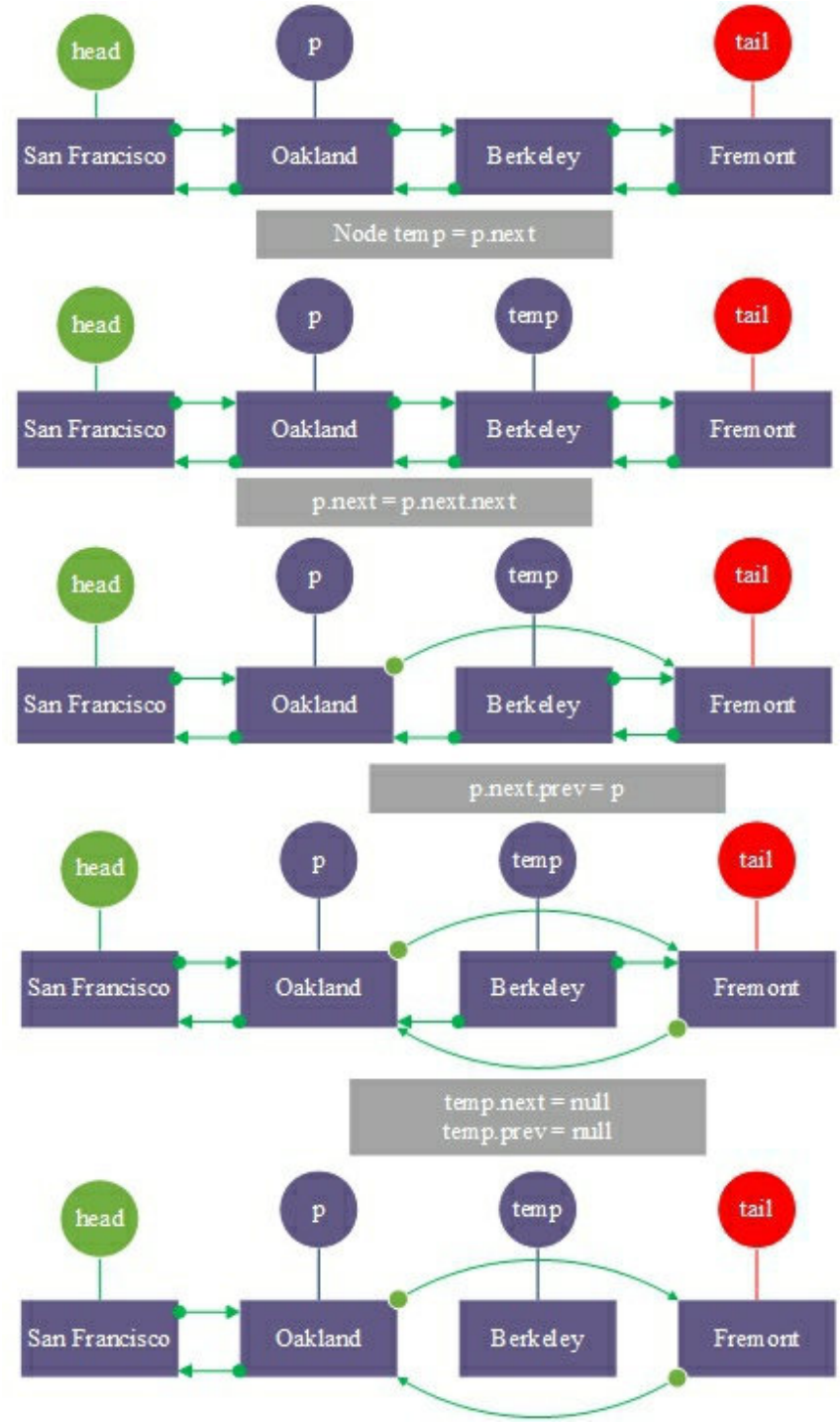
```
    }
    var newNode *Node = new(Node)
    newNode.data = data
    newNode.next = p.next // newNode next point to next node
    p.next = newNode      // current next point to newNode
    newNode.prev = p
    newNode.next.prev = newNode
}

func output(node *Node) {
    var p = node
    var end *Node = nil
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        end = p
        p = p.next
    }
    fmt.Printf("End\n")

    p = end
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.prev
    }
    fmt.Printf("Start\n\n")
}

func main() {
    initial()

    fmt.Printf("Insert a new node Walnut at index 2 : \n")
    insert(2, "Walnut")
```

```
    output(head)
}
```

Result:

San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End
Fremont -> Berkeley -> Walnut -> Oakland -> San Francisco -> Start

4. Delete the index=2 node.



TestDoubleLink.go

```
package main
import "fmt"
type Node struct {
    data string
    prev *Node
    next *Node
}
var head *Node = new(Node)
var tail *Node = new(Node)
func initial() {
    head.data = "San Francisco"
    head.prev = nil
    head.next = nil

    var nodeOakland *Node = &Node{data: "Oakland", prev: head, next:
nil}
    head.next = nodeOakland

    var nodeBerkeley *Node = &Node{data: "Berkeley", prev:
nodeOakland, next: nil}
    nodeOakland.next = nodeBerkeley

    tail.data = "Fremont"
    tail.prev = nodeBerkeley
    tail.next = nil
    nodeBerkeley.next = tail
}

func removeNode(removePosition int) {
    var p = head
    var i = 0
    // Move the node to the previous node position that was deleted
    for {
        if p.next == nil || i >= removePosition-1 {
            break
        }
        p = p.next
```

```
        i++
    }
    var temp = p.next    // Save the node you want to delete
    p.next = p.next.next // Previous node next points to next of delete the
node
    p.next.prev = p
    temp.next = nil // Set the delete node next to null
    temp.prev = nil // Set the delete node prev to null
}

func output(node *Node) {
    var p = node
    var end *Node = nil
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        end = p
        p = p.next
    }
    fmt.Printf("End\n")

    p = end
    for {
        if p == nil {
            break
        }
        fmt.Printf("%s -> ", p.data)
        p = p.prev
    }

    fmt.Printf("Start\n\n")
}

func main() {
    initial()

    fmt.Printf("Delete a new node Berkeley at index = 2 : \n")
```

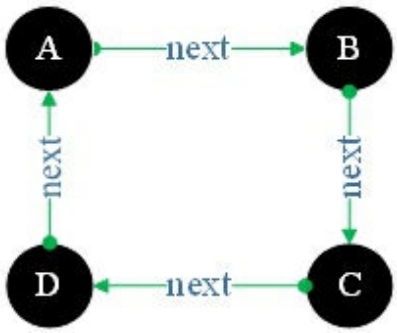
```
    removeNode(2)

    output(head)
}
```

Result:

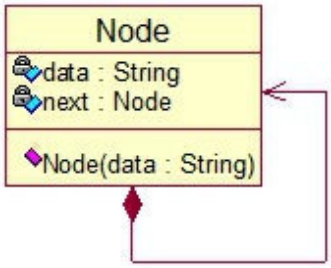
San Francisco -> Oakland -> Fremont -> End
Fremont -> Oakland -> San Francisco -> Start

One-way Circular LinkedList

One-way Circular List:
It is a chain storage structure of a linear table, which is connected to form a ring, and each node is composed of data and a pointer to next.

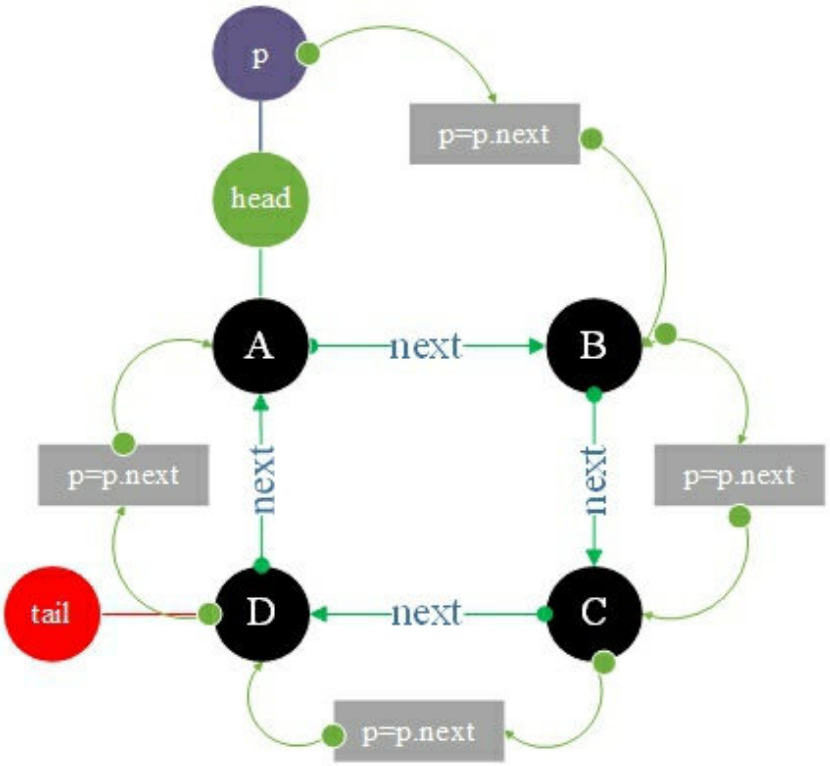


UML Diagram




```
type Node struct {
    data string
    next *Node
}
```

1. One-way Circular Linked List initialization and traversal output.



TestSingleCircleLink.go

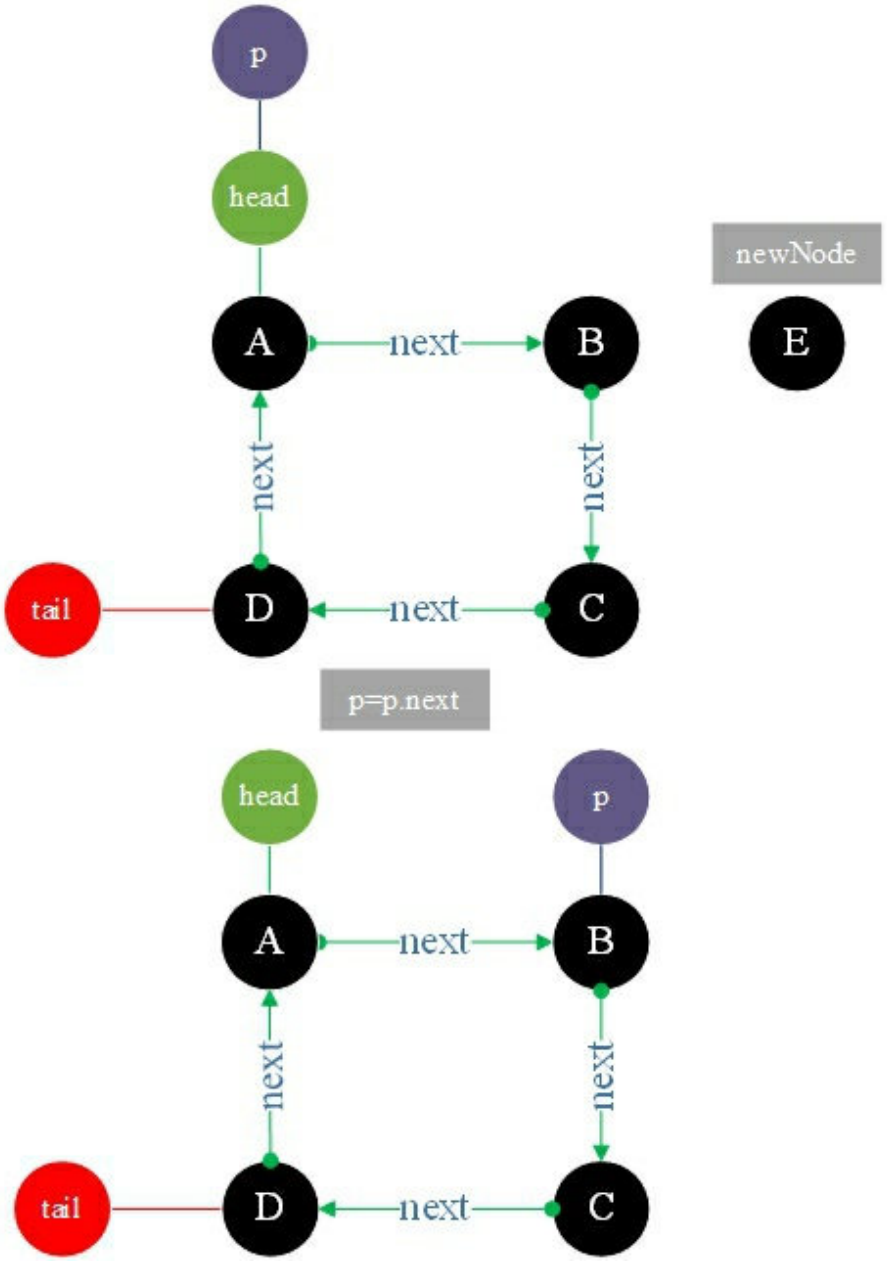
```
package main
import "fmt"

type Node struct {
    data string
    next *Node
}
var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "A"
    head.next = nil

    var nodeB *Node = &Node{data: "B", next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.next = head
    nodeC.next = tail
}

func output(node *Node) {
    var p = node
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
        break
        }
    }
    fmt.Printf("%s \n\n", p.data)
}

func main() {
```

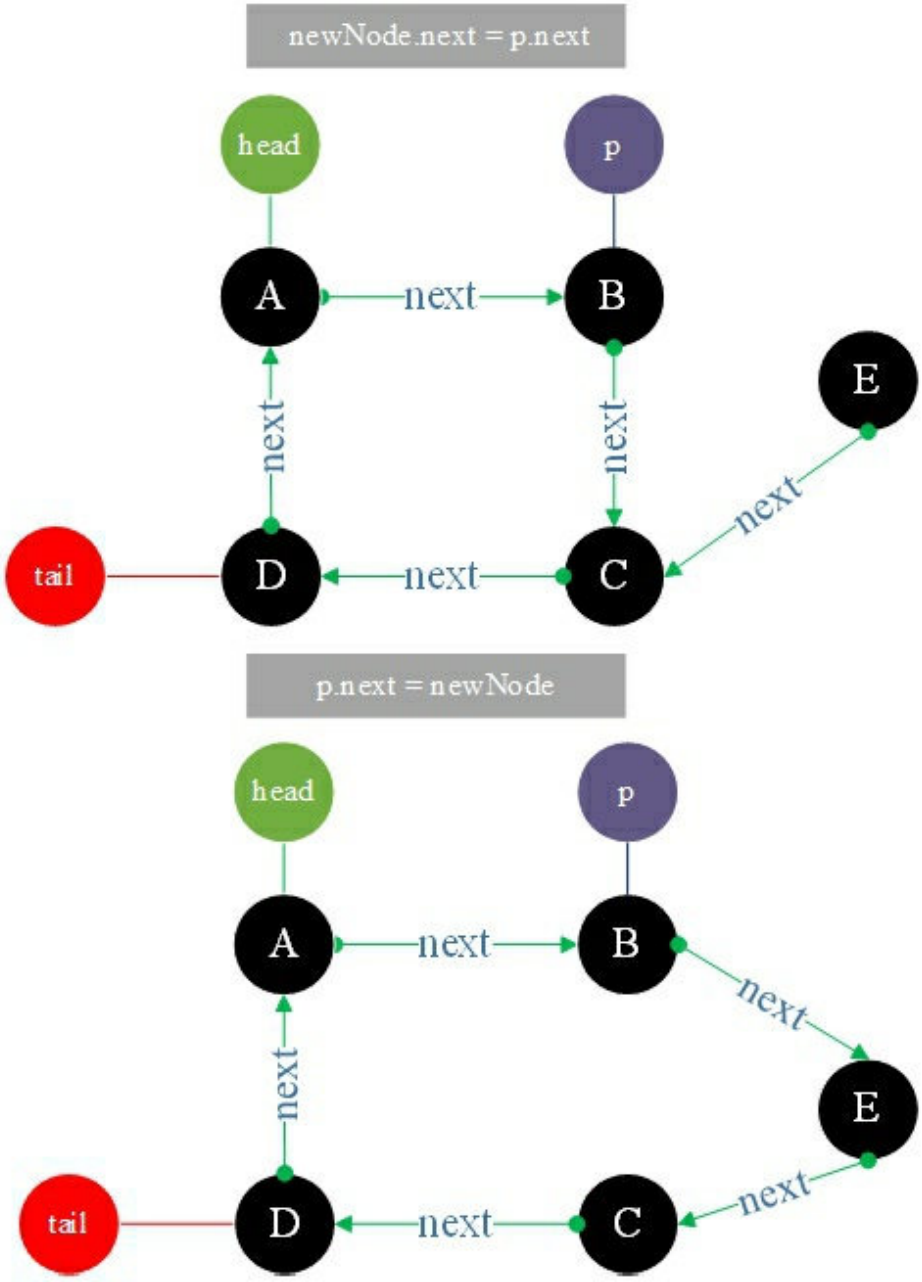
```
    initial()
    output(head)
}
```

Result:

A -> B -> C -> D -> A

3. Insert a node E in position 2.




newNode.next = p.next
head
p
A
next
B
E
next
next
next
tail
D
next
C
p.next = newNode
head
p
A
next
B
next
E
next
tail
D
next
C

TestSingleCircleLink.go

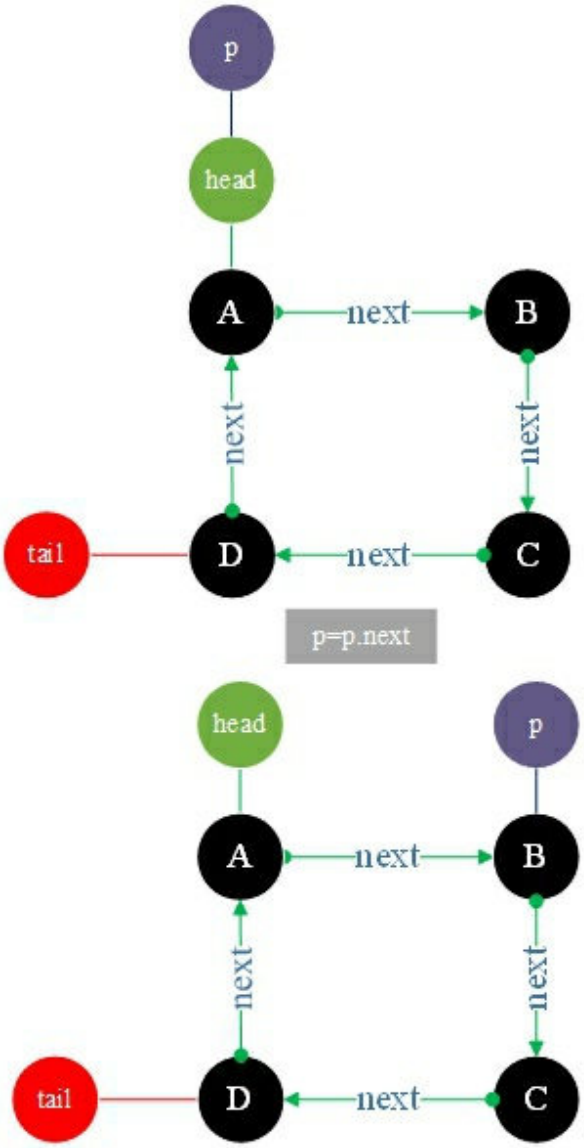
```
package main
import "fmt"
type Node struct {
    data string
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "A"
    head.next = nil

    var nodeB *Node = &Node{data: "B", next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.next = head
    nodeC.next = tail
}

func insert(insertPosition int, data string) {
    var p = head
    var i = 0
    // Move the node to the insertion position
    for {
        if p.next == nil || i >= insertPosition-1 {
          break
        }
        p = p.next
        i++
    }
    var newNode *Node = new(Node)
```

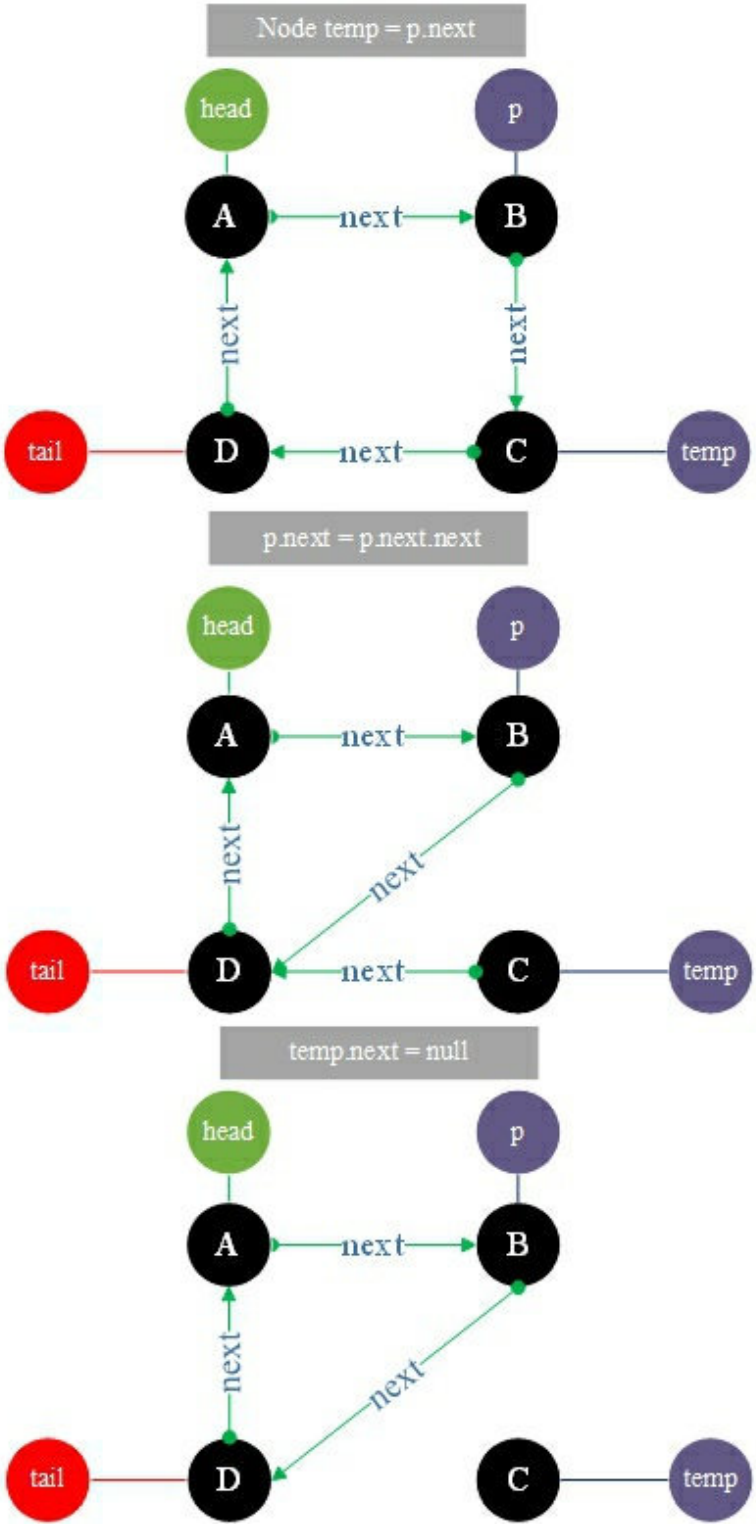
```
    newNode.data = data
    newNode.next = p.next // newNode next point to next node
    p.next = newNode      // current next point to newNode
}


func output(node *Node) {
    var p = node
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
          break
        }
    }

    fmt.Printf("%s \n\n", p.data)
}

func main() {
    initial()

    fmt.Printf("Insert a new node E at index = 2 : \n")
    insert(2, "E")

    output(head)
}
```

Result:

A -> B -> E -> C -> D -> A

4. Delete the index=2 node.




Node temp = p.next
head
p
A
next
B
next
next
tail
D
next
C
temp
p.next = p.next.next
head
p
A
next
B
next
next
tail
D
next
C
temp
temp.next = null
head
p
A
next
B
next
next
tail
D
C
temp

TestSingleCircleLink.go

```
package main
import "fmt"
type Node struct {
    data string
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "A"
    head.next = nil

    var nodeB *Node = &Node{data: "B", next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.next = head
    nodeC.next = tail
}

func removeNode(removePosition int) {
    var p = head
    var i = 0
    // Move the node to the previous node position that was deleted
    for {
        if p.next == nil || i >= removePosition-1 {
            break
        }
        p = p.next
        i++
    }

    var temp = p.next    // Save the node you want to delete
```

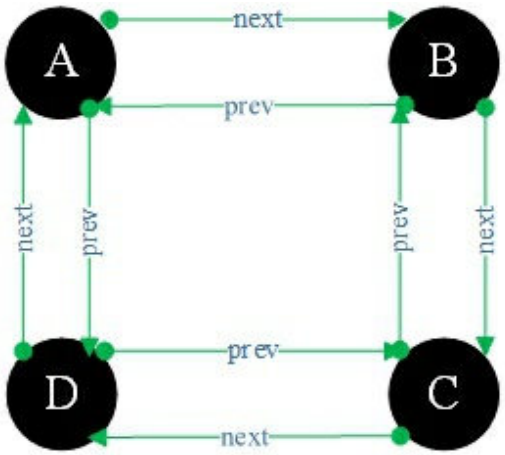
```
    p.next = p.next.next // Previous node next points to next of delete the node
    temp.next = nil
}


func output(node *Node) {
    var p = node
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
            break
        }
    }

    fmt.Printf("%s \n\n", p.data)
}

func main() {
    initial()

    fmt.Printf("Delete a new node E at index = 2 : \n")
    removeNode(2)

    output(head)
}
```

Result:

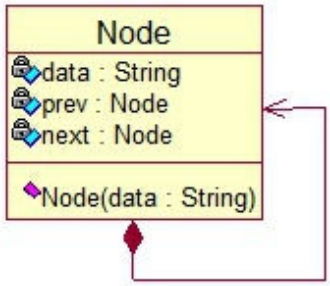
A -> B -> D -> A

Two-way Circular LinkedList

Two-way Circular List:
It is a chain storage structure of a linear table. The nodes are connected in series by two directions, and is connected to form a ring. Each node is composed of data, pointing to the previous node prev and pointing to the next node next.

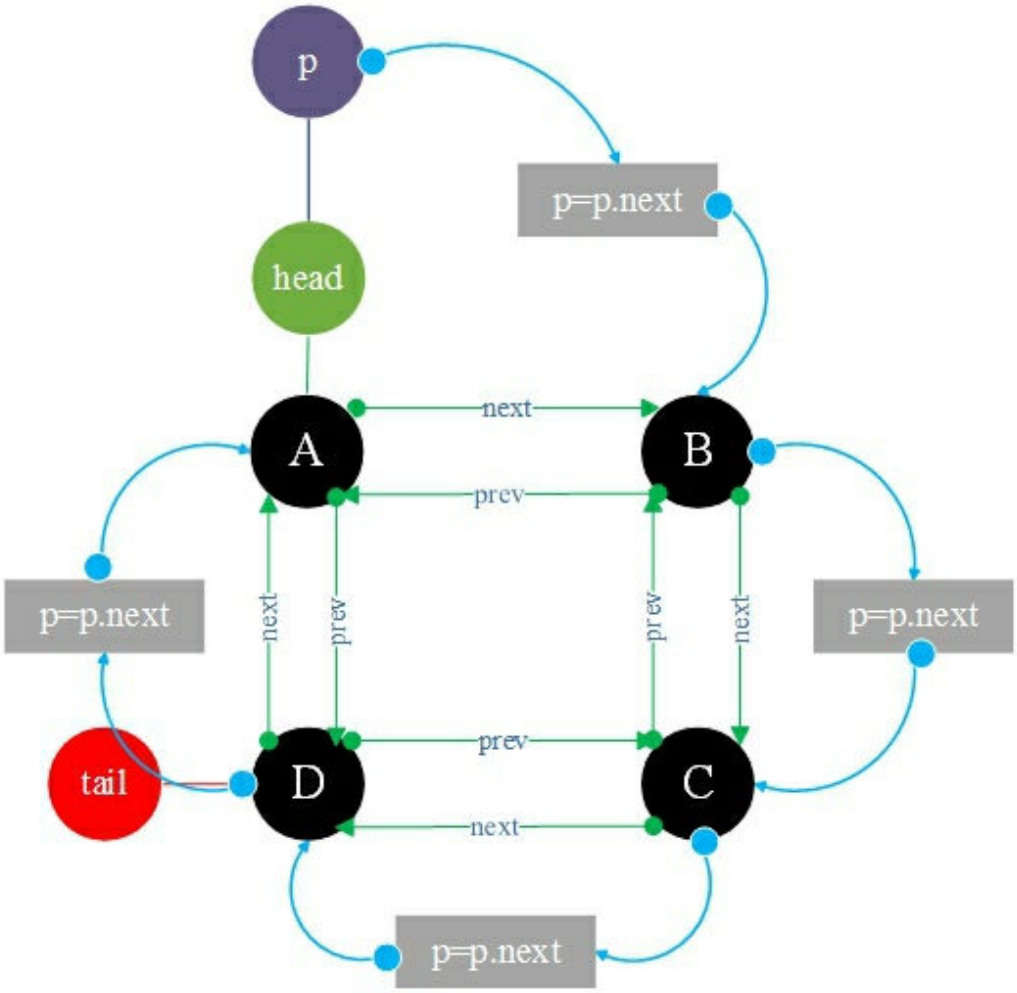


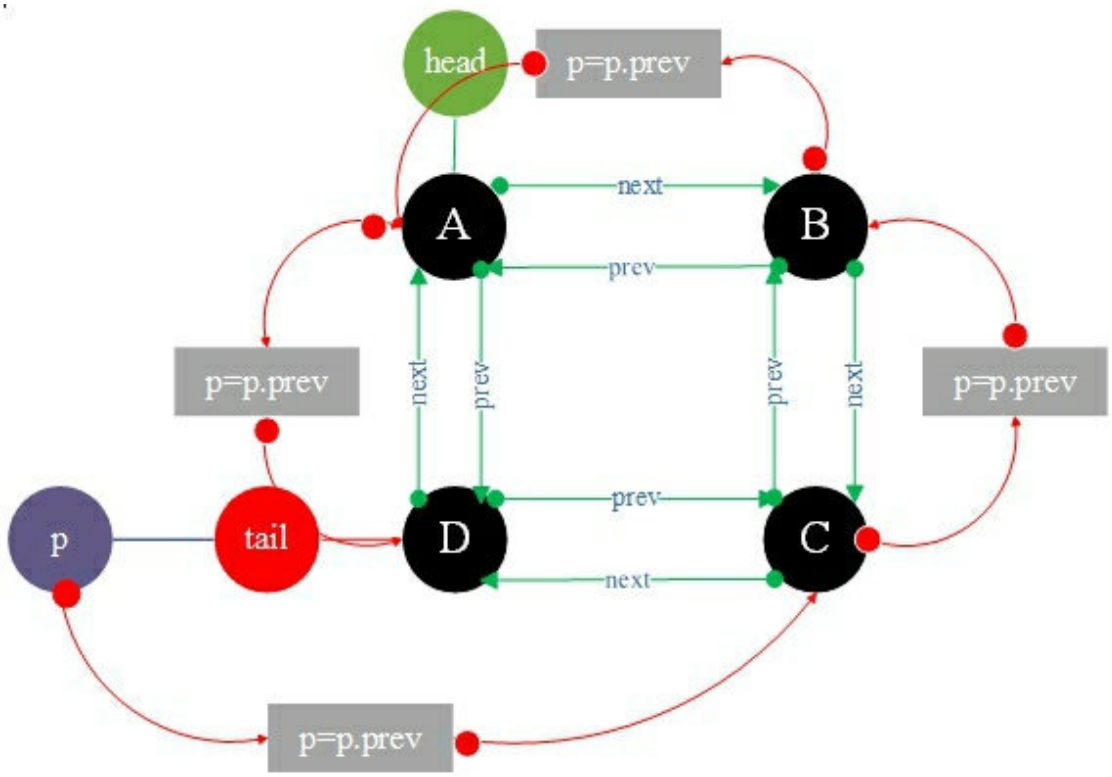
UML Diagram




```
type Node struct {
    data string
    prev *Node
    next *Node
}
```

1. Two-way Circular Linked List initialization and traversal output.




head
p=p.prev
next
A
B
prev
p=p.prev
next
prev
prev
next
p=p.prev
p
tail
D
prev
C
next
p=p.prev

TestDoubleCircleLink.go

```go
package main

import "fmt"

type Node struct {
    data string
    prev *Node
    next *Node
}

var head *Node = new(Node)
var tail *Node = new(Node)

func initial() {
    head.data = "A"
    head.prev = nil
    head.next = nil

    var nodeB *Node = &Node{data: "B", prev: head, next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", prev: nodeB, next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.prev = nodeC
    tail.next = head
    nodeC.next = tail
    head.prev = tail
}
```

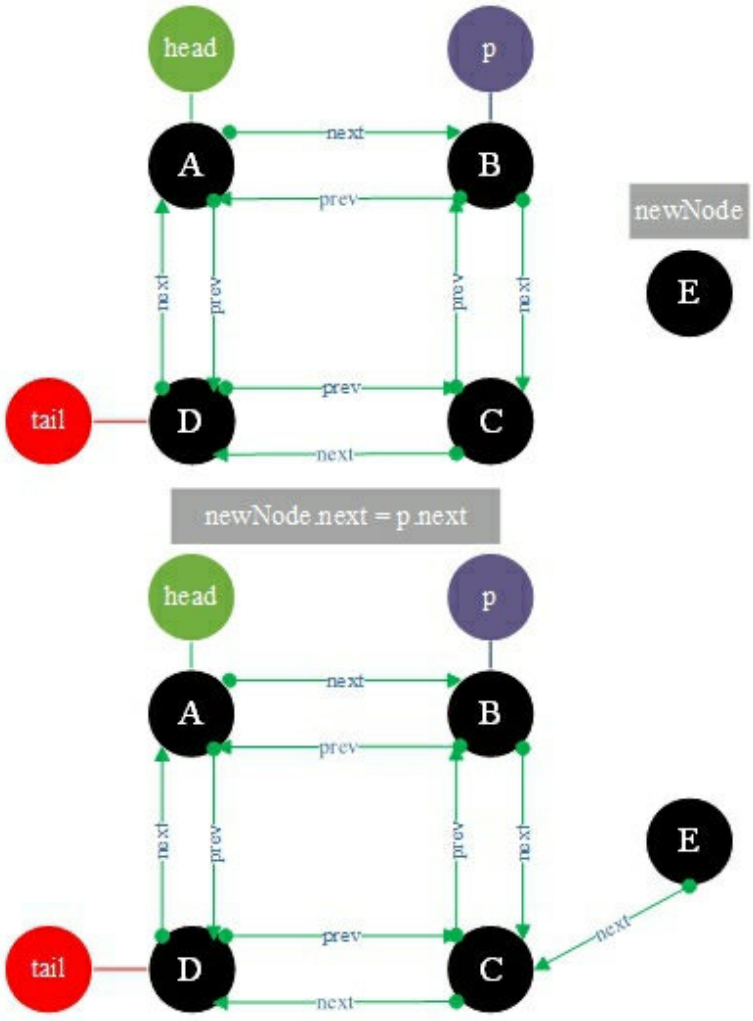
```
func output() {
    var p = head
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
          break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("End\n")

    p = tail
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.prev
        if p == tail {
          break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("Start\n\n")
}

func main() {
    initial()
    output()
}
```

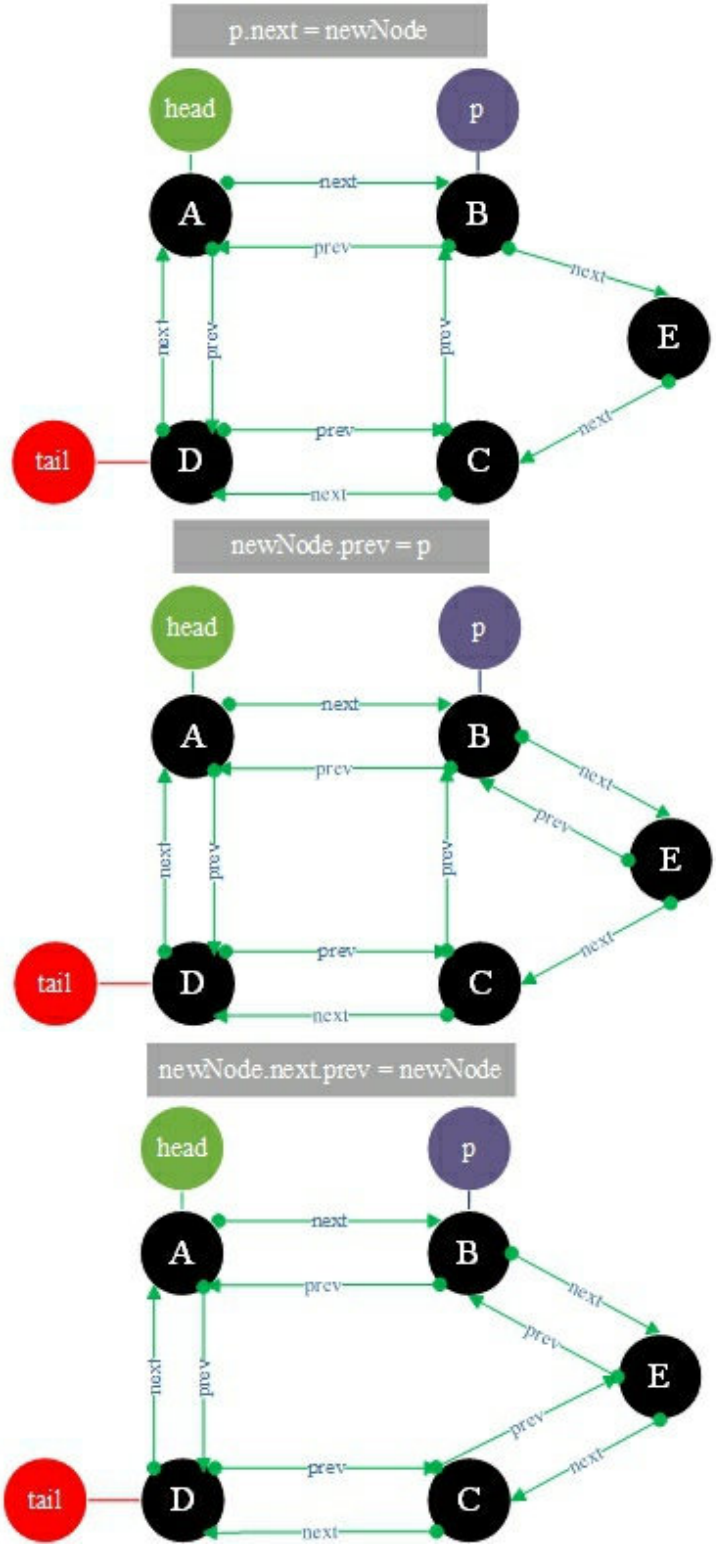
Result:

A -> B -> C -> D -> A

D -> C -> B -> A -> D

3. Insert a node E in position 2.




p.next = newNode
head
p
A
B
E
tail
D
C
next
prev
newNode.prev = p
newNode.next.prev = newNode

TestDoubleCircleLink.go

```
package main
import "fmt"
type Node struct {
    data string
    prev *Node
    next *Node
}
var head *Node = new(Node)
var tail *Node = new(Node)
func initial() {
    head.data = "A"
    head.prev = nil
    head.next = nil

    var nodeB *Node = &Node{data: "B", prev: head, next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", prev: nodeB, next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.prev = nodeC
    tail.next = head
    nodeC.next = tail
    head.prev = tail
}
func insert(insertPosition int, data string) {
    var p = head
    var i = 0
    for {
        if p.next == nil || i >= insertPosition-1 {
          break
        }
        p = p.next
        i++
    }
    var newNode *Node = new(Node)
```

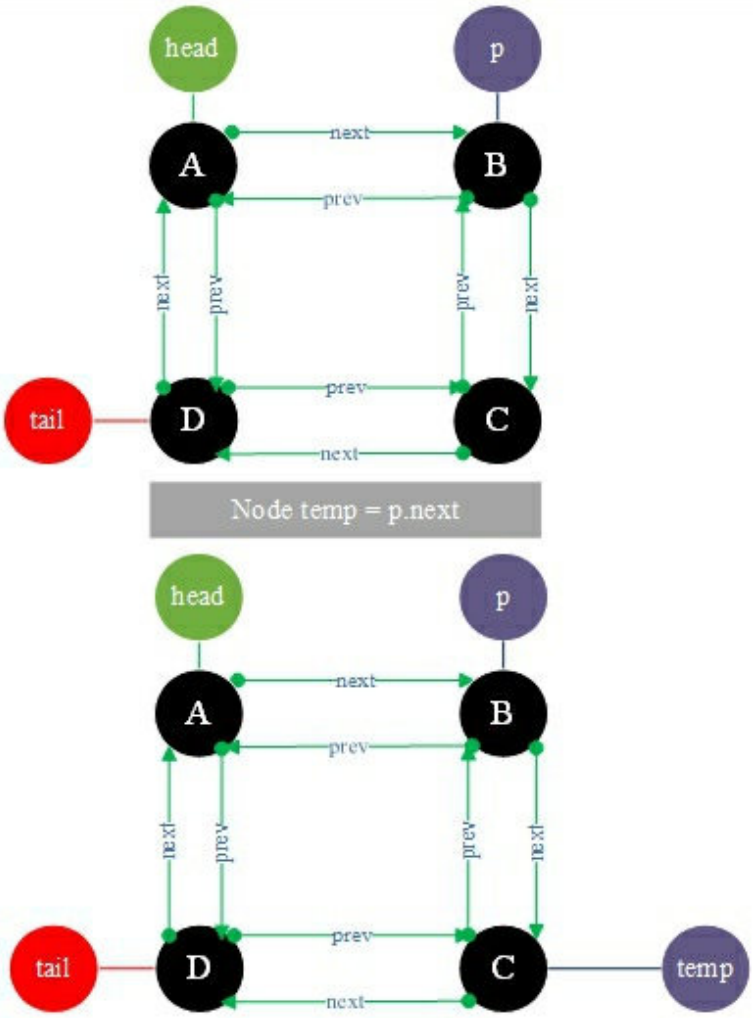
```
    newNode.data = data
    newNode.next = p.next // newNode next point to next node
    p.next = newNode      // current next point to newNode
    newNode.prev = p
    newNode.next.prev = newNode
}

func output() {
    var p = head
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
          break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("End\n")

    p = tail
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.prev
        if p == tail {
          break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("Start\n\n")
}

func main() {
    initial()

    fmt.Printf("Insert a new node E at index 2 : \n")
    insert(2, "E")

    output()
}
```

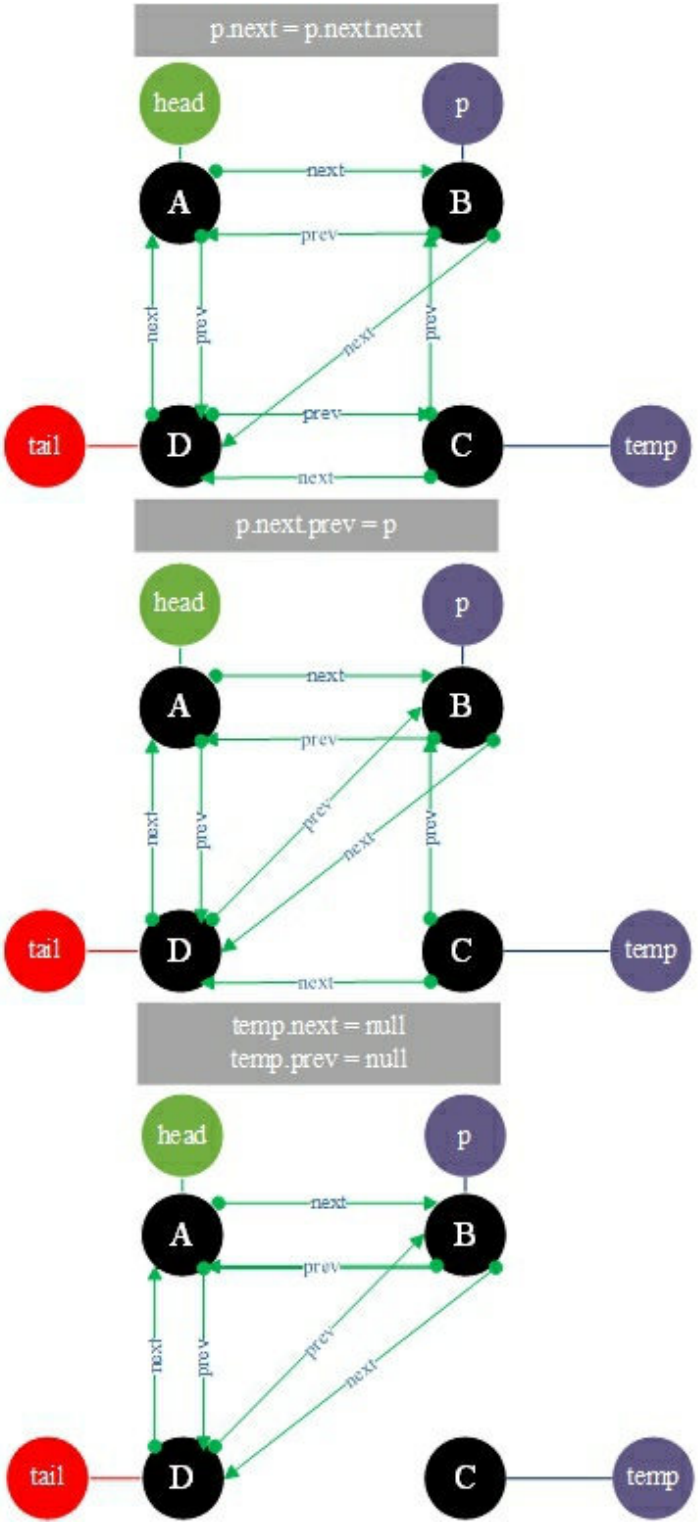
Result:

A -> B -> E -> C -> D -> A

D -> C -> E -> B -> A -> D

4. Delete the index=2 node.




p.next = p.next.next
head
p
A
B
next
prev
next
prev
next
prev
tail
D
C
temp
next
p.next.prev = p
head
p
A
B
next
prev
prev
next
prev
next
tail
D
C
temp
next
temp.next = null
temp.prev = null
head
p
A
B
next
prev
prev
next
prev
next
tail
D
C
temp

TestDoubleCircleLink.go

```
package main
import "fmt"
type Node struct {
    data string
    prev *Node
    next *Node
}
var head *Node = new(Node)
var tail *Node = new(Node)
func initial() {
    head.data = "A"
    head.prev = nil
    head.next = nil

    var nodeB *Node = &Node{data: "B", prev: head, next: nil}
    head.next = nodeB

    var nodeC *Node = &Node{data: "C", prev: nodeB, next: nil}
    nodeB.next = nodeC

    tail.data = "D"
    tail.prev = nodeC
    tail.next = head
    nodeC.next = tail
    head.prev = tail
}

func removeNode(removePosition int) {
    var p = head
    var i = 0
    for {
        if p.next == nil || i >= removePosition-1 {
            break
        }
        p = p.next
        i++
    }
```

```
    var temp = p.next    // Save the node you want to delete
    p.next = p.next.next // Previous node next points to next of delete the
node
    p.next.prev = p
    temp.next = nil // Set the delete node next to null
    temp.prev = nil // Set the delete node prev to null
}

func output() {
    var p = head
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.next
        if p == head {
            break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("End\n")

    p = tail
    for {
        fmt.Printf("%s -> ", p.data)
        p = p.prev
        if p == tail {
            break
        }
    }
    fmt.Printf("%s ", p.data)
    fmt.Printf("Start\n\n")
}

func main() {
    initial()

    fmt.Printf("Delete a new node C at index = 2 : \n")
    removeNode(2)

    output()
```

```
}
```

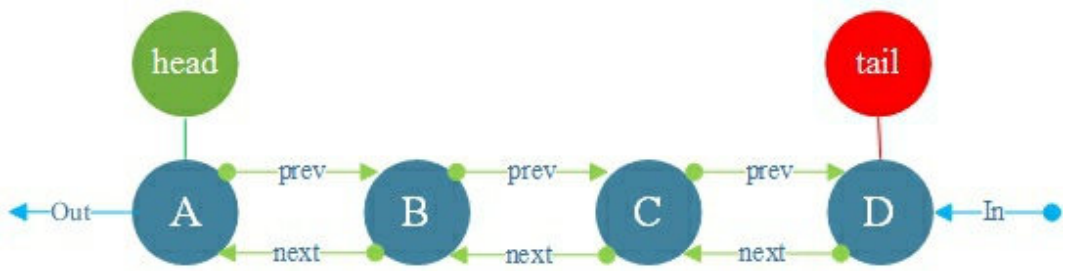
Result:

A -> B -> D -> A

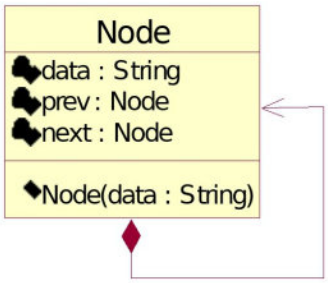
D -> B -> A -> D

Queue

Queue:
FIFO (First In First Out) sequence.



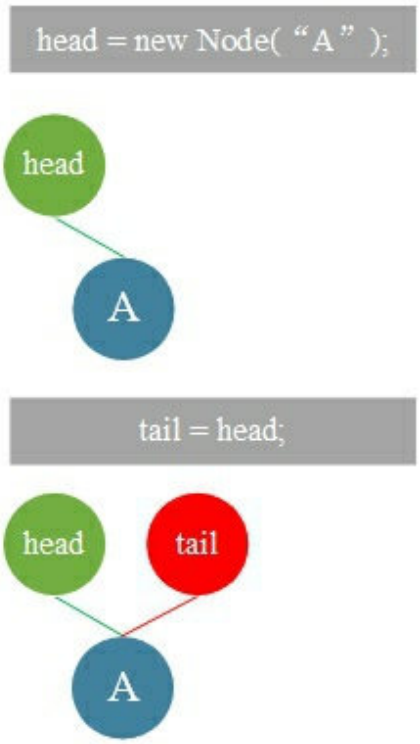
UML Diagram



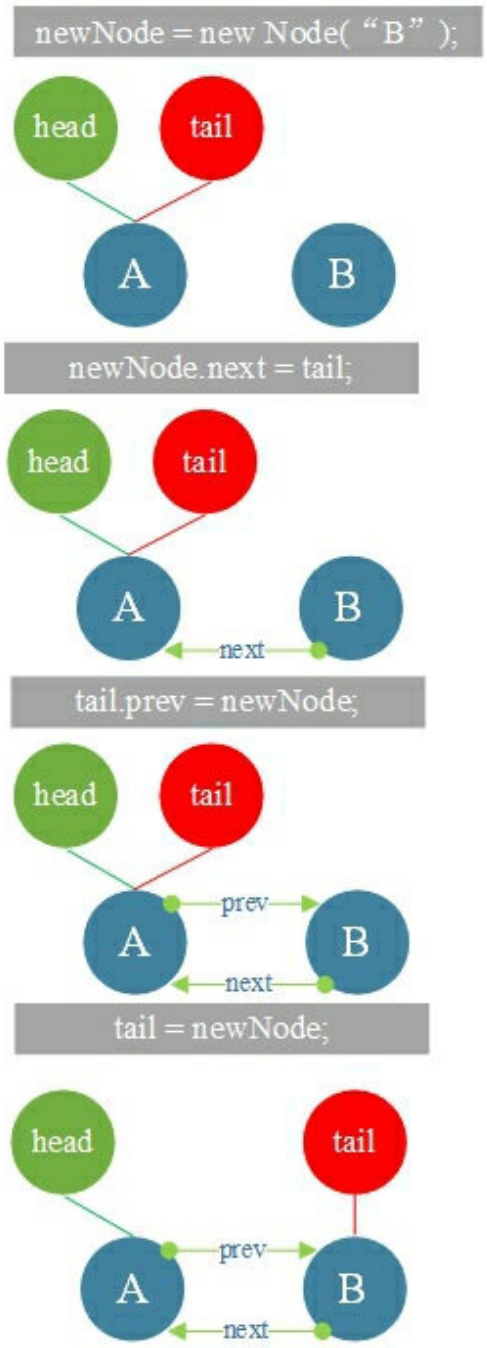

```
type Node struct {
    data string
    prev *Node
    next *Node
}
```

1. Queue initialization and traversal output.

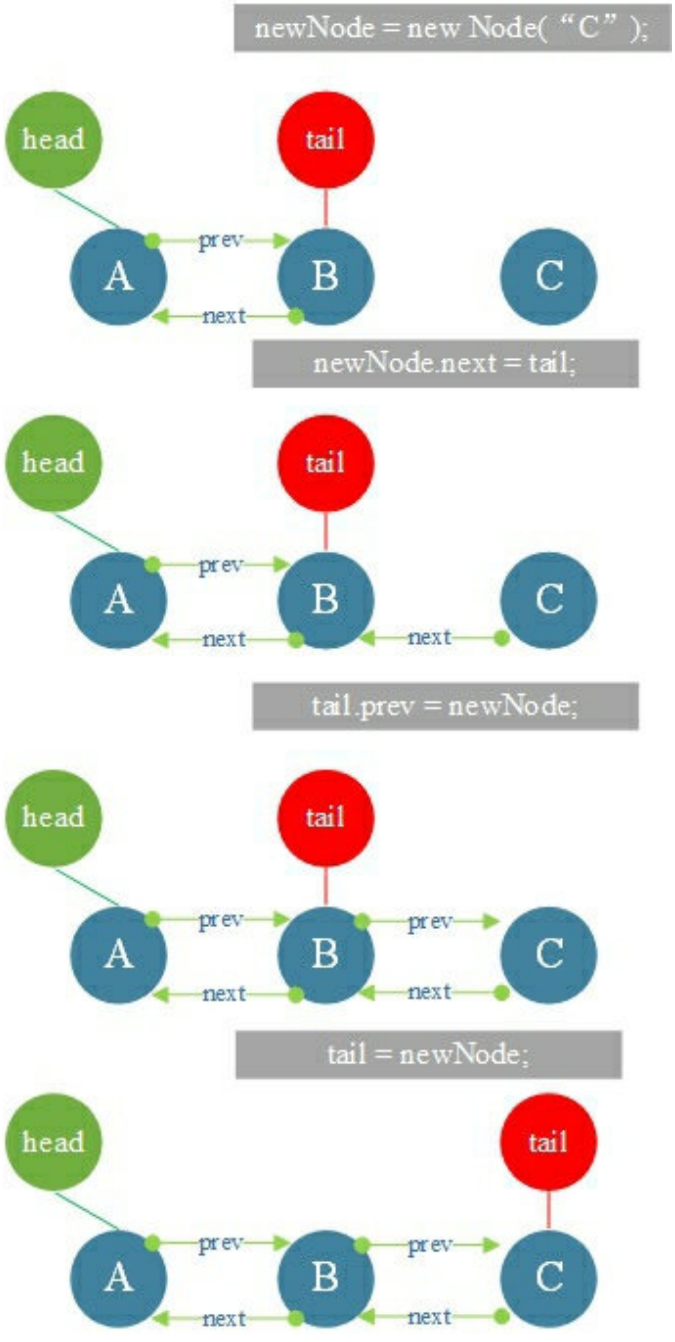
Initialization Insert A



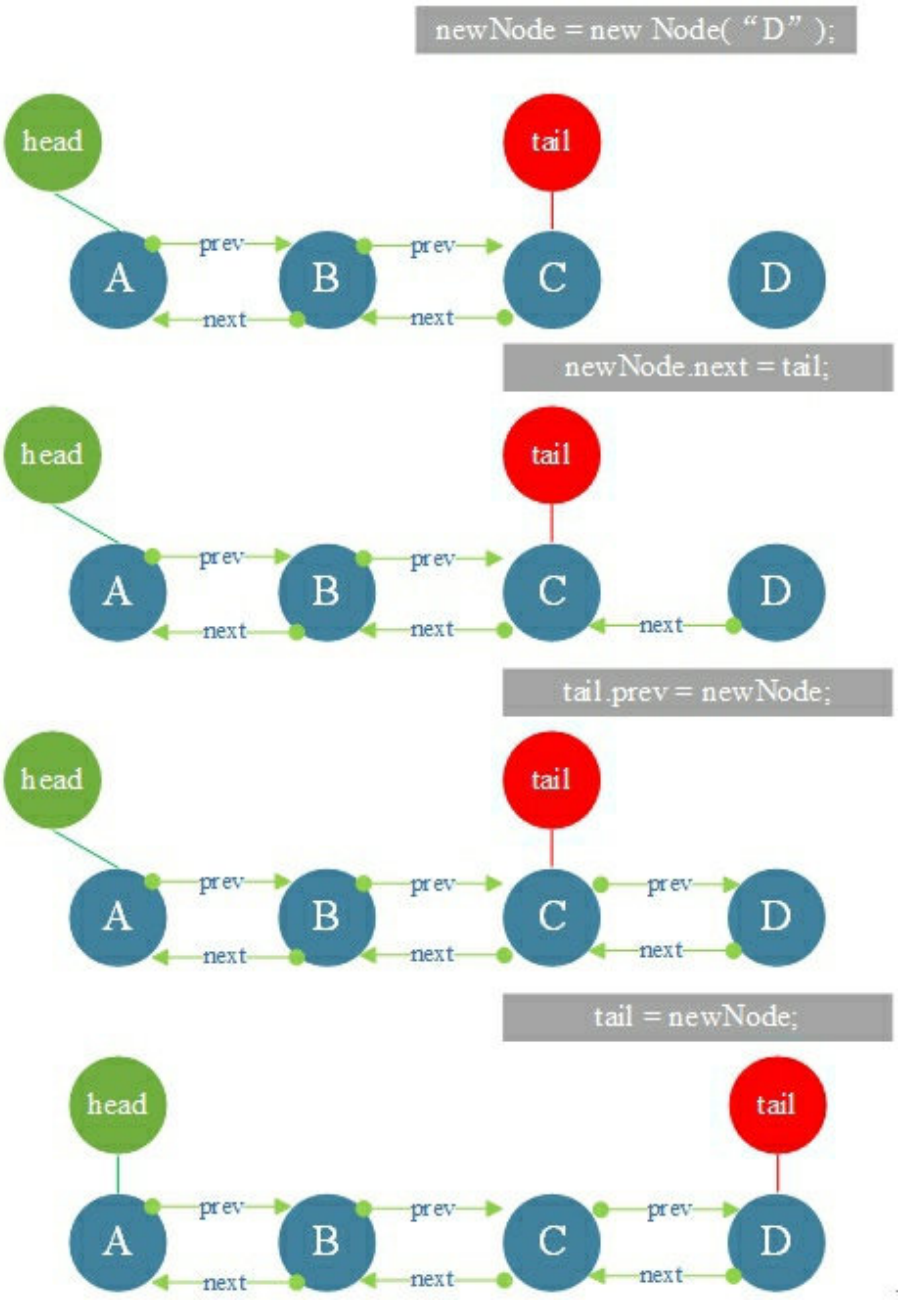
Initialization Insert B



Initialization Insert C



Initialization Insert D



Queue.go

```
package main
import "fmt"

type Node struct {
    data string
    prev *Node
    next *Node
}

var head *Node = nil
var tail *Node = new(Node)
var size int

func offer(element string) {
    if head == nil {
        head = new(Node)
        head.data = element
        tail = head
    } else {
        var newNode *Node = new(Node)
        newNode.data = element
        newNode.next = tail
        tail.prev = newNode
        tail = newNode
    }
    size++
}

func poll() *Node {
    var p = head

    if p == nil {
        return nil
    }
    head = head.prev
    p.next = nil
```

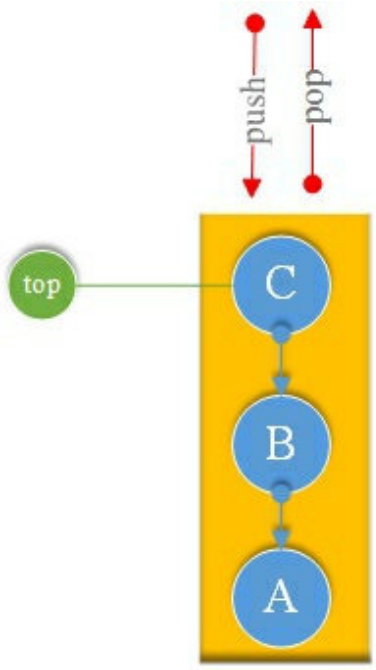
```
    p.prev = nil
    size--
    return p
}


func output() {
    fmt.Printf("Head ")
    var node *Node = nil
    for {
        node = poll()
        if node == nil {
       break
        }
        fmt.Printf("%s <- ", node.data)
    }
    fmt.Printf("Tail\n")
}

func main() {
    offer("A")
    offer("B")
    offer("C")
    offer("D")

    output()
}
```

Result:

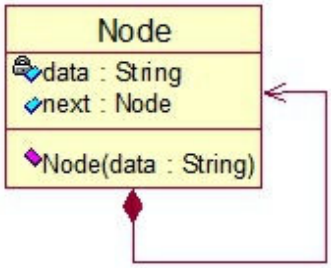
Head A <- B <- C <- D <- Tail

Stack

Stack:
FILO (First In Last Out) sequence.



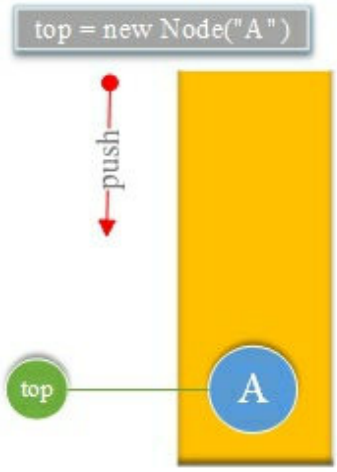
UML Diagram



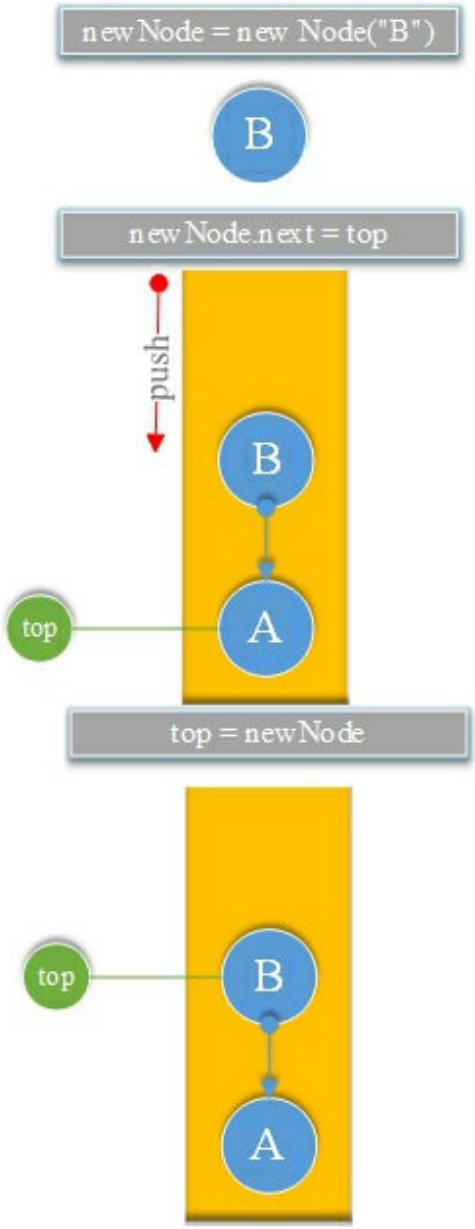
```
type Node struct {
    data string
    next *Node
}
```

1. Stack initialization and traversal output.

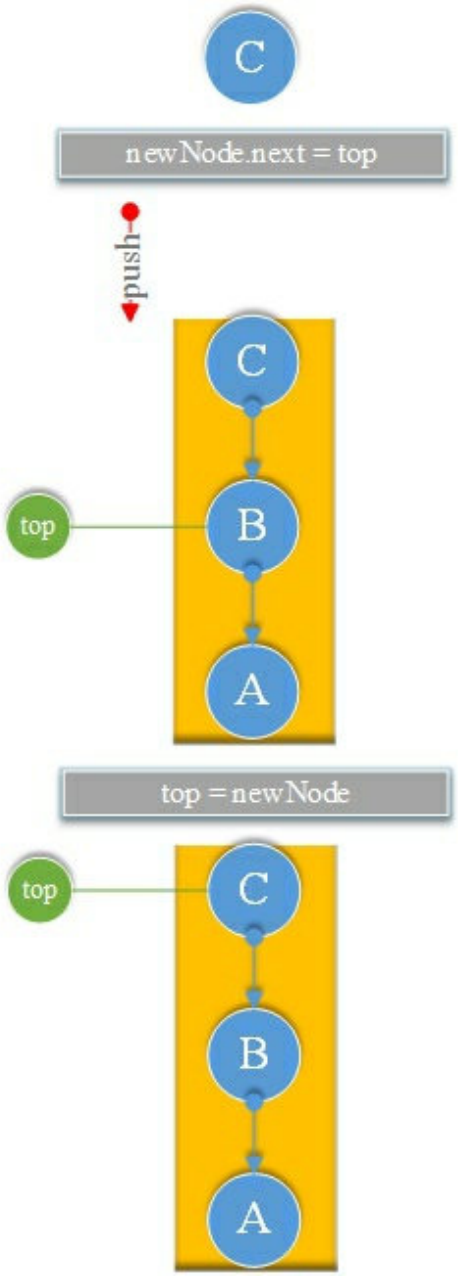
Push A into Stack



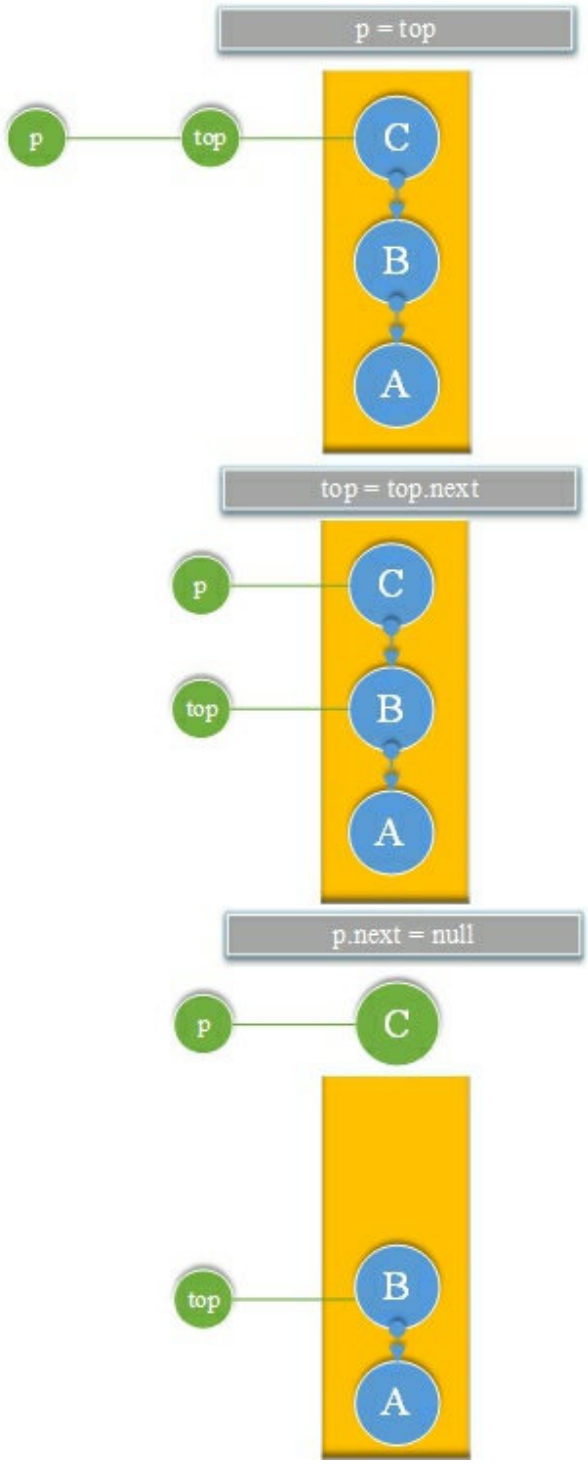
Push B into Stack



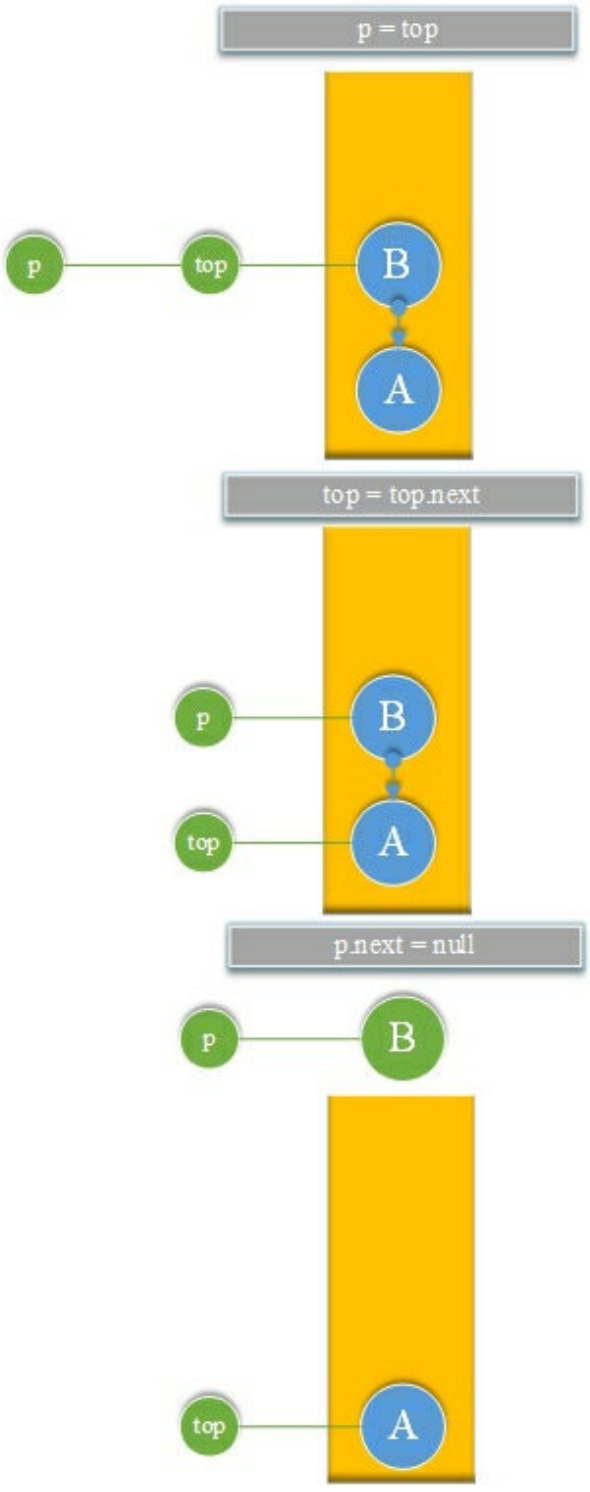
Push C into Stack



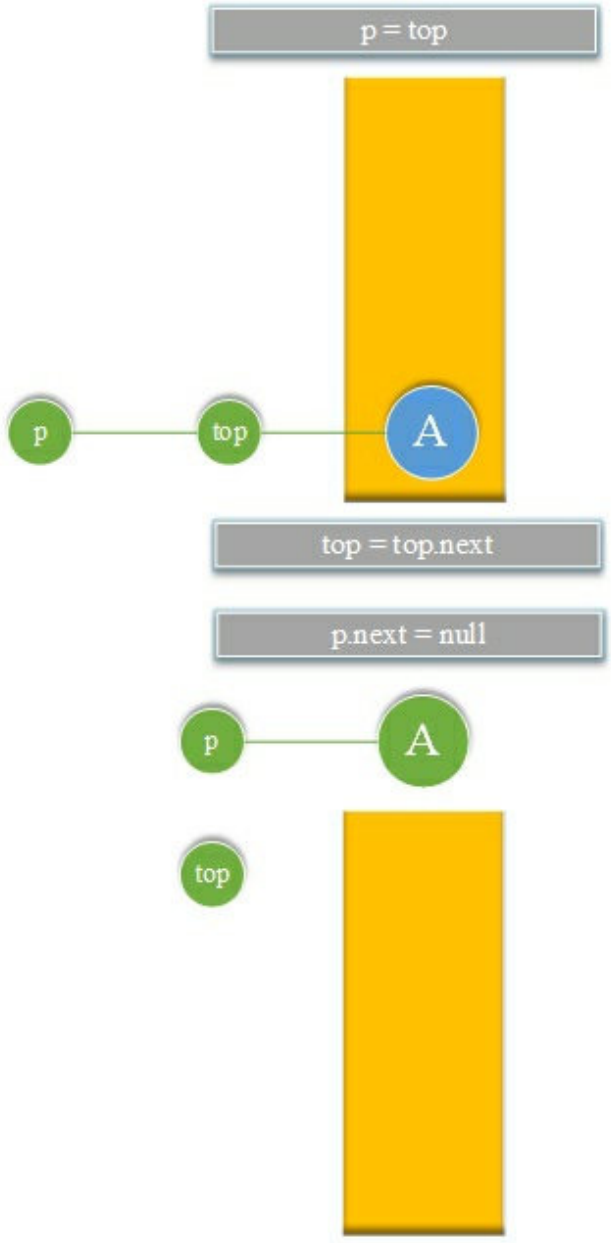
If pop C from Stack:



If pop B from Stack:



If pop A from Stack:



Stack.go

```
package main
import "fmt"

type Node struct {
    data string
    next *Node
}

var top *Node = nil
var size int

func push(element string) {
    if top == nil {
        top = new(Node)
        top.data = element
    } else {
        var newNode *Node = new(Node)
        newNode.data = element
        newNode.next = top
        top = newNode
    }
    size++
}

func pop() *Node {
    if top == nil {
        return nil
    }

    var p = top
    top = top.next // top move down

    p.next = nil
    size--
    return p
}
```

```
func output() {
    fmt.Printf("Top ")
    var node *Node = nil
    for {
        node = pop()
        if node == nil {
       break
        }
        fmt.Printf("%s -> ", node.data)
    }
    fmt.Printf("End\n")
}

func main() {
    push("A")
    push("B")
    push("C")
    push("D")

    output()
}
```

Result:

Top D -> C -> B -> A -> End

Recursive Algorithm

Recursive Algorithm:
The program function itself calls its own layer to progress until it reaches a certain condition and step by step returns to the end..

1. Factorial of n : n*(n-1)*(n-2) *2*1

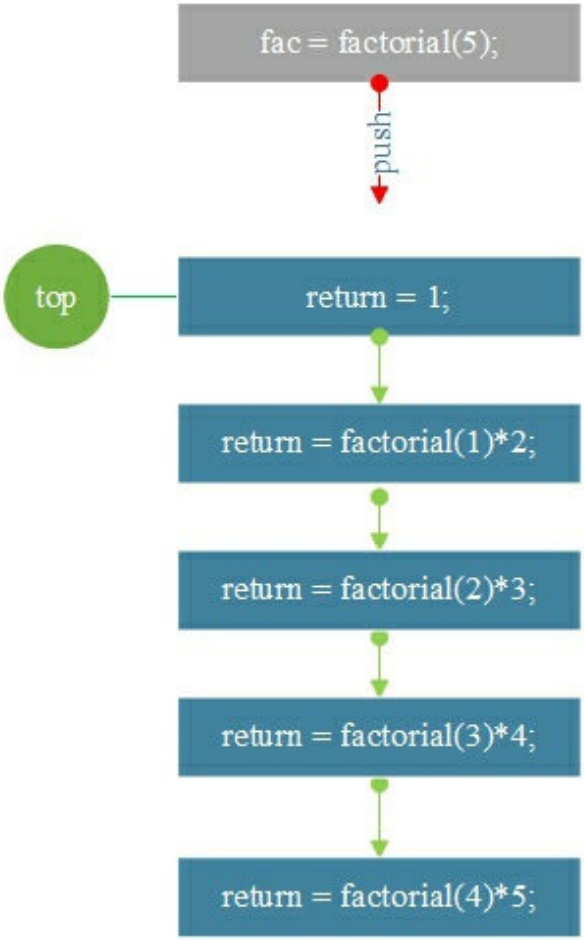
TestFactorial.go

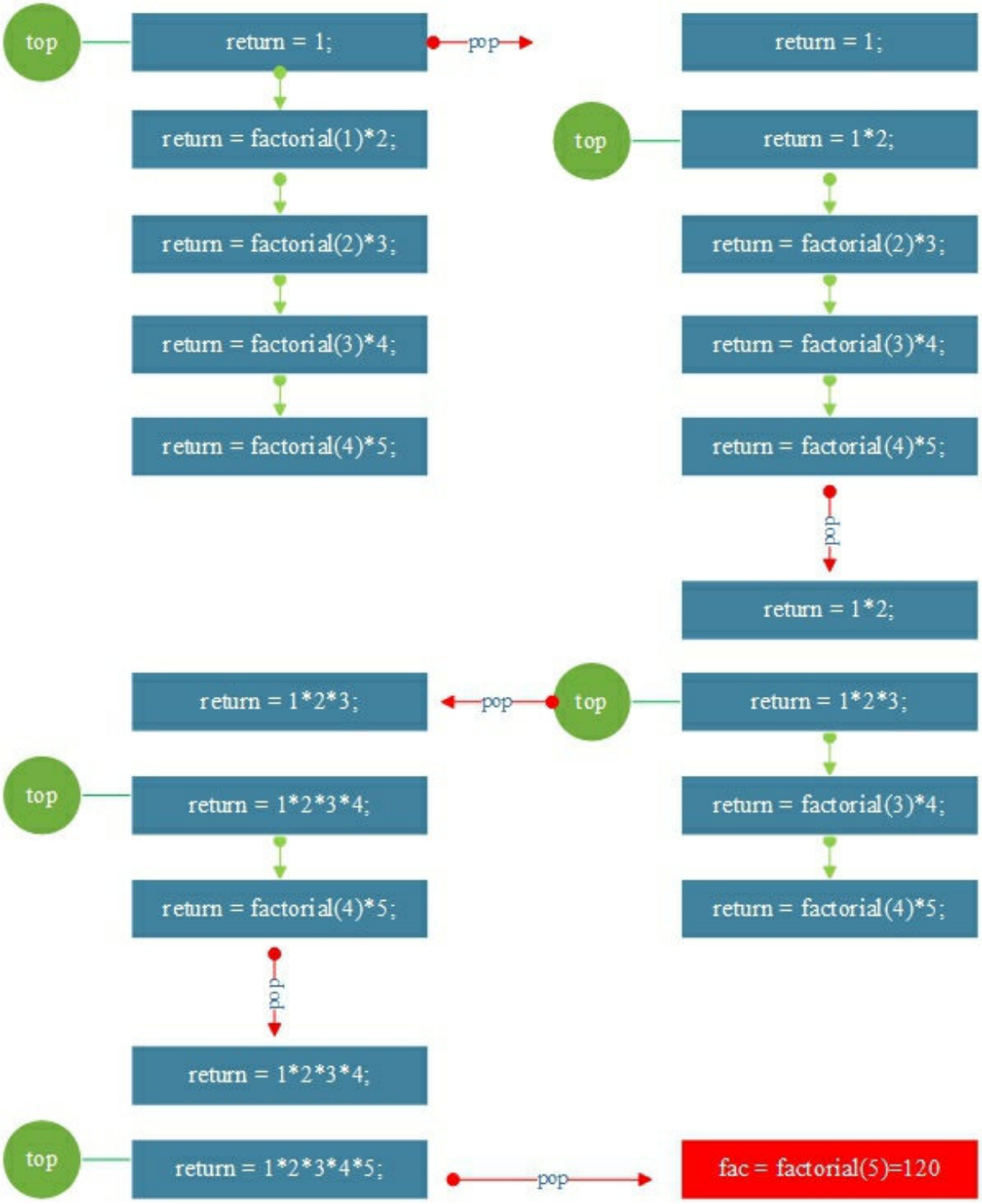
```
package main

import "fmt"

func factorial(n int) int {
    if n == 1 {
        return 1
    } else {
        return factorial(n-1) * n //Recursively call yourself until the end of the return
    }
}

func main() {
    var n = 5
    var fac = factorial(n)
    fmt.Printf("The factorial of 5 is : %d", fac)
}
```

Result:

The factorial of 5 is :120

Graphical analysis:

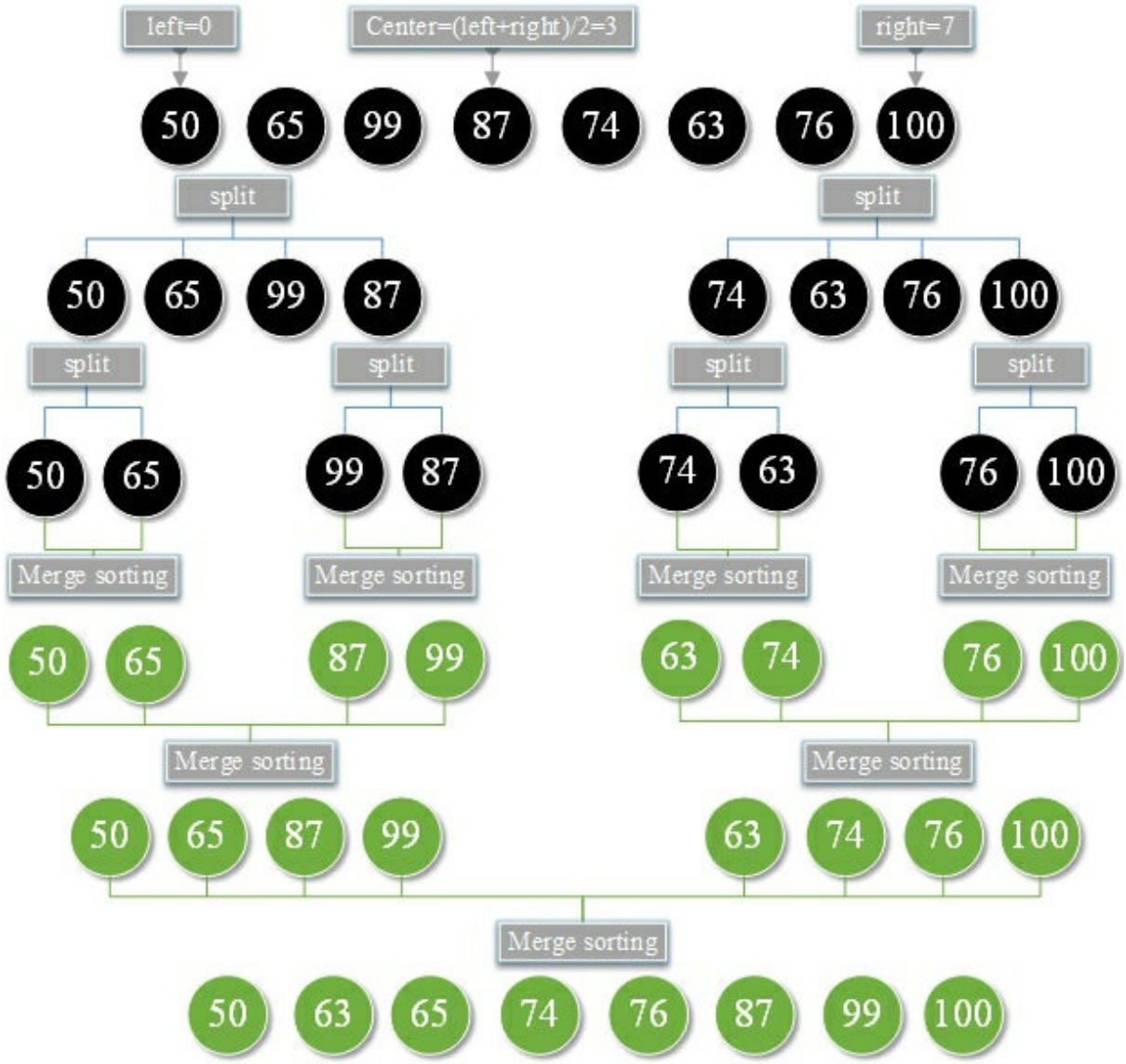



top
return = 1;
pop
return = 1;
return = factorial(1)*2;
top
return = 1*2;
return = factorial(2)*3;
return = factorial(2)*3;
return = factorial(3)*4;
return = factorial(3)*4;
return = factorial(4)*5;
return = factorial(4)*5;
pop
return = 1*2;
return = 1*2*3;
pop
top
return = 1*2*3;
top
return = 1*2*3*4;
return = factorial(3)*4;
return = factorial(4)*5;
return = factorial(4)*5;
pop
return = 1*2*3*4;
top
return = 1*2*3*4*5;
pop
fac = factorial(5)=120

Two-way Merge Algorithm

Two-way Merge Algorithm:
The data of the first half and the second half are sorted, and the two ordered sub-list are merged into one ordered list, which continue to recursive to the end.

1. The scores {50, 65, 99, 87, 74, 63, 76, 100} by merge sort



TestMergeSort.go

```
package main

import "fmt"

func main() {
    var scores = []int{50, 65, 99, 87, 74, 63, 76, 100, 92}
    var length = len(scores)

    sort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}

func sort(array []int, length int) {

    var temp = make([]int, length)
    mergeSort(array, temp, 0, length-1)

}

func mergeSort(array []int, temp []int, left int, right int) {
    if left < right {
        var center = (left + right) / 2
        mergeSort(array, temp, left, center)      // Left merge sort
        mergeSort(array, temp, center+1, right)   // Right merge sort
        merge(array, temp, left, center+1, right) // Merge two ordered arrays
    }
}


/**
Combine two ordered list into an ordered list
temp : Temporary array
left :    Start the subscript on the left
right :  Start the subscript on the right
rightEndIndex : End subscript on the right
```

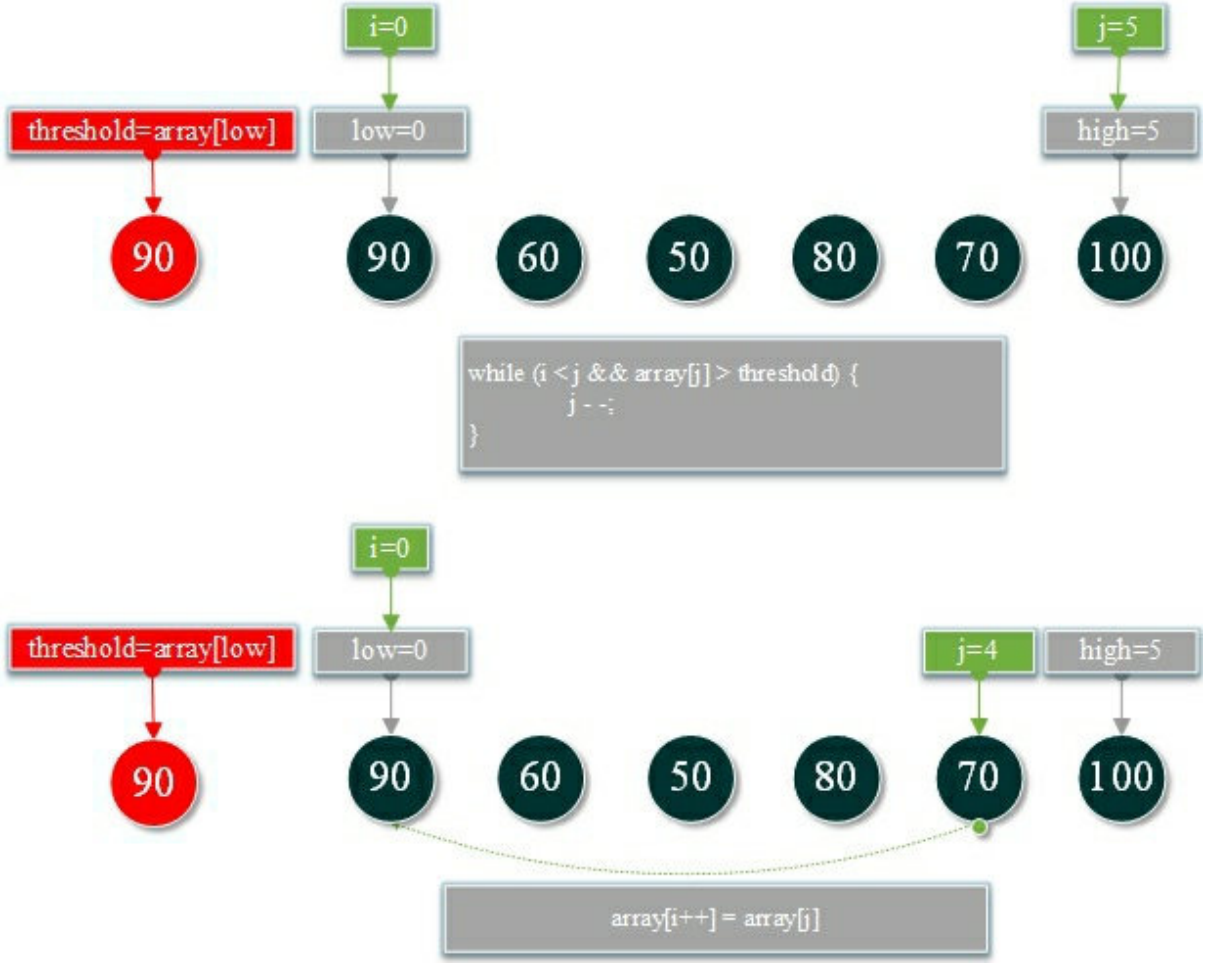
```
*/

func merge(array []int, temp []int, left int, right int, rightEndIndex int) {
    var leftEndIndex = right - 1 // End subscript on the left
    var tempIndex = left         // Starting from the left count
    var elementNumber = rightEndIndex - left + 1

    for {
        if left > leftEndIndex || right > rightEndIndex {
            break
        }
        if array[left] <= array[right] {
            temp[tempIndex] = array[left]
            tempIndex++
            left++
        } else {
            temp[tempIndex] = array[right]
            tempIndex++
            right++
        }
    }

    for {
        if left > leftEndIndex {
            break
        }
        temp[tempIndex] = array[left] // If there is element on the left
        tempIndex++
        left++
    }

    for {
        if right > rightEndIndex {
            break
        }
        temp[tempIndex] = array[right] // If there is element on the right
        tempIndex++
        right++
```

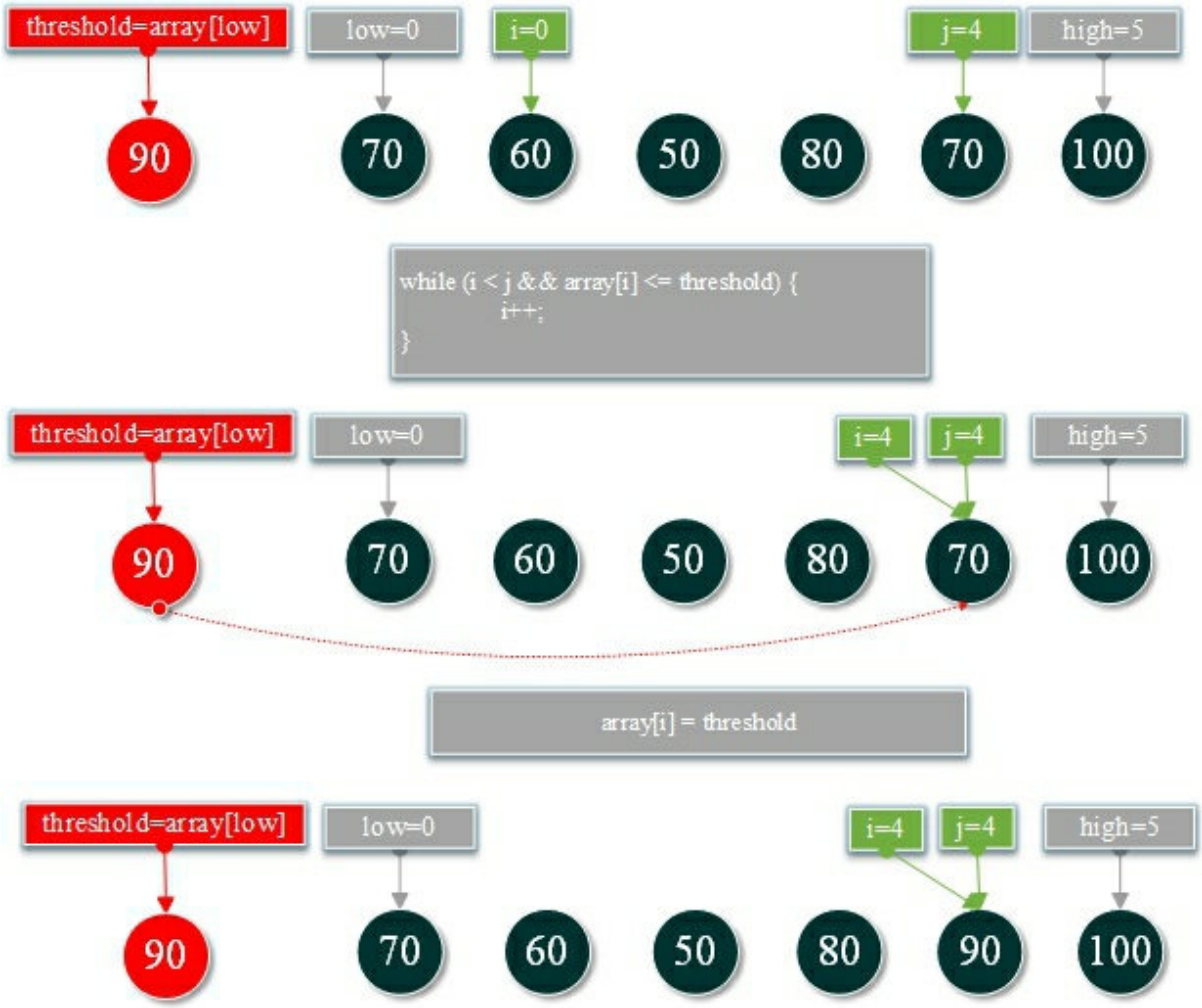
```
    }

    for i := 0; i < elementNumber; i++ {
        array[rightEndIndex] = temp[rightEndIndex]
        rightEndIndex--
    }
}
```

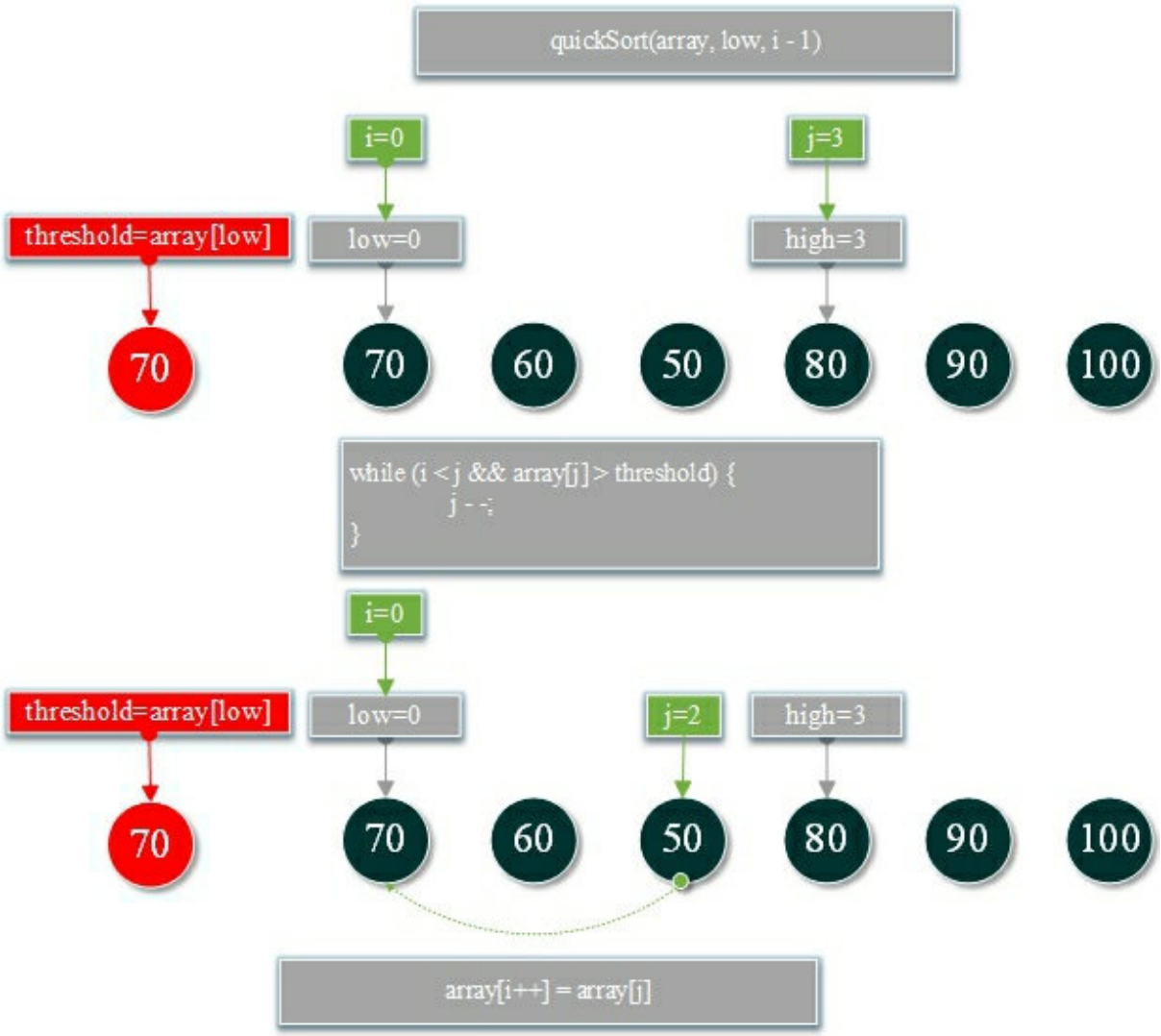
Quick Sort Algorithm

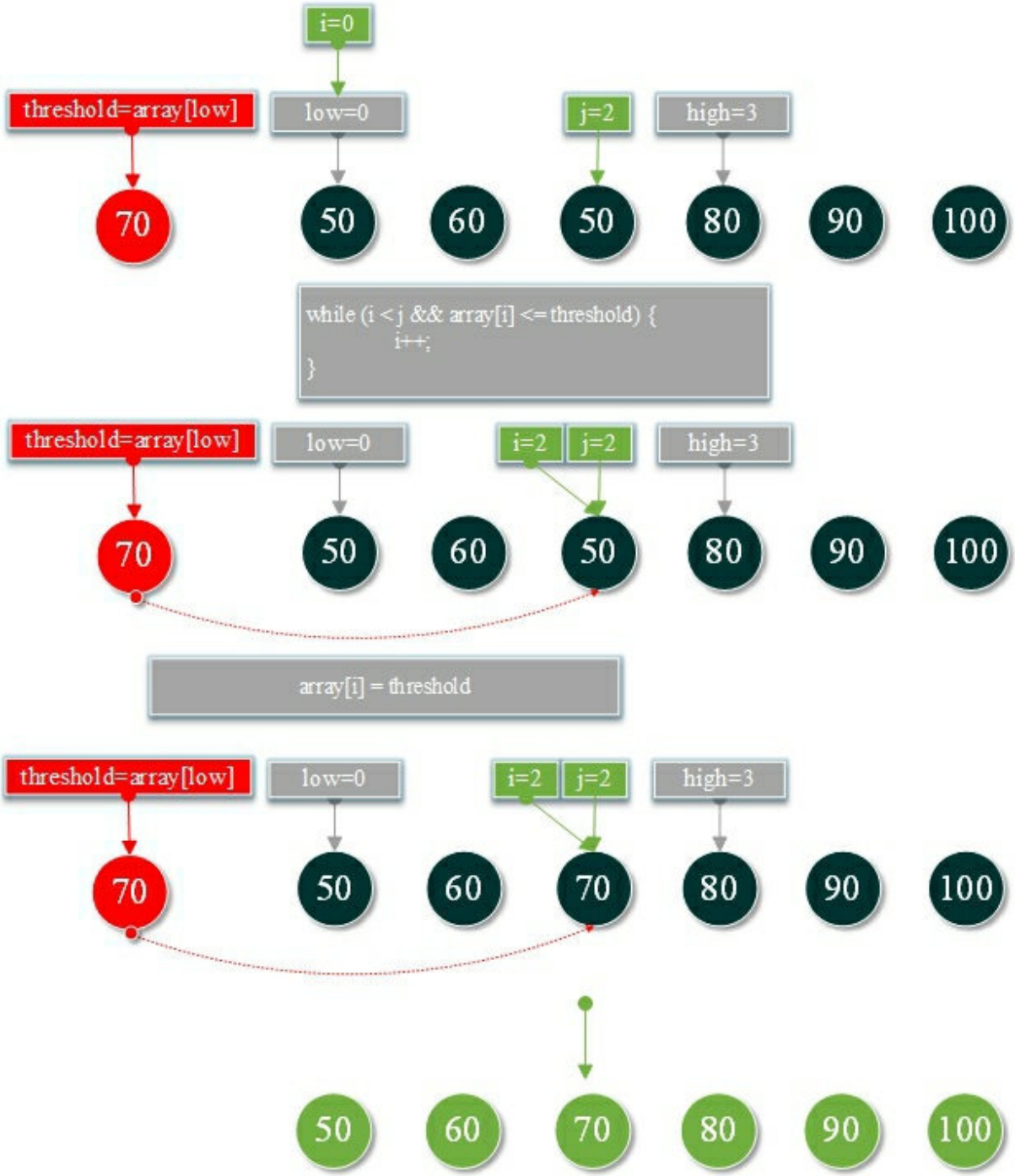
Quick Sort Algorithm:
Quicksort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.

1. The scores {90, 60, 50, 80, 70, 100} by quick sort




threshold=array[low]
low=0
i=0
j=4
high=5
90
70
60
50
80
70
100
while (i < j && array[i] <= threshold) {
i++;
}
threshold=array[low]
low=0
i=4
j=4
high=5
90
70
60
50
80
70
100
array[i] = threshold
threshold=array[low]
low=0
i=4
j=4
high=5
90
70
60
50
80
90
100


quickSort(array, low, i - 1)
i=0
j=3
threshold=array[low]
low=0
high=3
70
70
60
50
80
90
100
while (i < j && array[j] > threshold) {
j - -;
}
i=0
threshold=array[low]
low=0
j=2
high=3
70
70
60
50
80
90
100
array[i++] = array[j]


i=0
threshold=array[low]
low=0
j=2
high=3
70
50
60
50
80
90
100
while (i < j && array[i] <= threshold) {
i++;
}
threshold=array[low]
low=0
i=2
j=2
high=3
70
50
60
50
80
90
100
array[i] = threshold
threshold=array[low]
low=0
i=2
j=2
high=3
70
50
60
70
80
90
100
50
60
70
80
90
100

TestQuickSort.go

```go
package main

import "fmt"

func main() {
    var scores = []int{50, 65, 99, 87, 74, 63, 76, 100, 92}
    var length = len(scores)

    sort(scores, length)

    for i := 0; i < length; i++ {
        fmt.Printf("%d,", scores[i])
    }
}

func sort(array []int, length int) {
    if length > 0 {
        quickSort(array, 0, length-1)
    }
}









func quickSort(array []int, low int, high int) {
    if low > high {
```

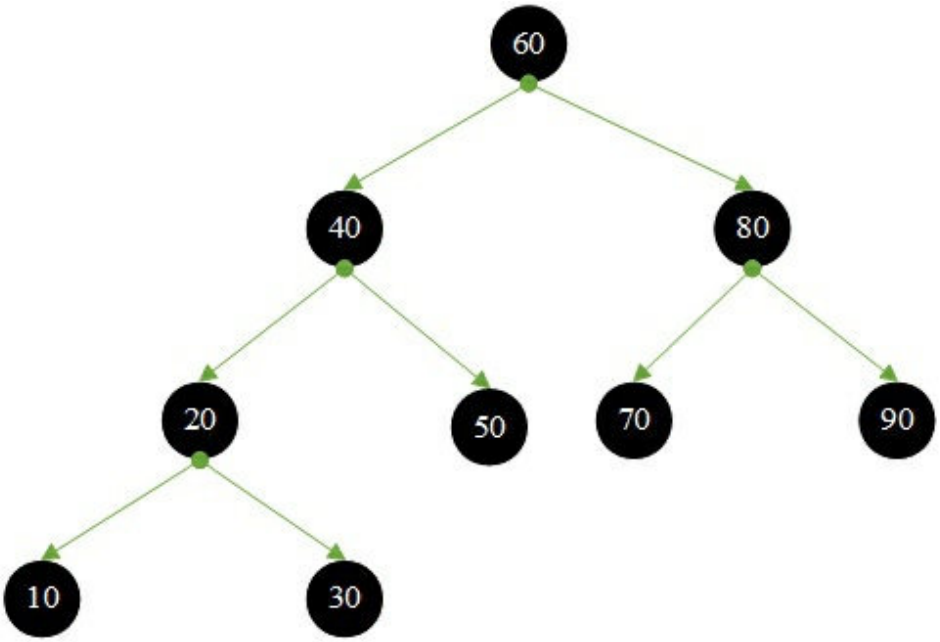
```
    return
}
var i = low
var j = high
var threshold = array[low]
// Alternately scanned from both ends of the list
for {
    if i >= j {
        break
    }
    // Find the first position less than threshold from right to left
    for {
        if i >= j || array[j] <= threshold {
            break
        }
        j--
    }
    //Replace the low with a smaller number than the threshold
    if i < j {
        array[i] = array[j]
        i++
    }
    // Find the first position greater than threshold from left to right
    for {
        if i >= j || array[i] > threshold {
            break
        }
        i++
    }
    //Replace the high with a number larger than the threshold
    if i < j {
        array[j] = array[i]
        j--
    }
}
array[i] = threshold
// left quickSort
```

```
    quickSort(array, low, i-1)
    // right quickSort
    quickSort(array, i+1, high)
}
```

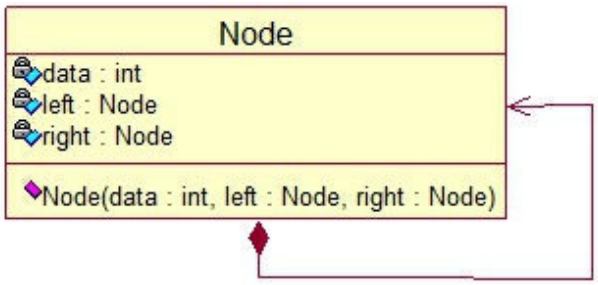
Binary Search Tree

Binary Search Tree:

1. If the left subtree of any node is not empty, the value of all nodes on the left subtree is less than the value of its root node;
2. If the right subtree of any node is not empty, the value of all nodes on the right subtree is greater than the value of its root node;
3. The left subtree and the right subtree of any node are also binary search trees.



Node UML Diagram



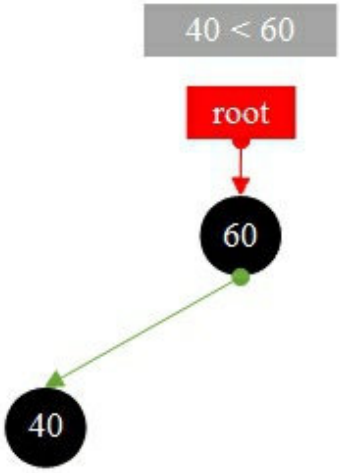

```
type Node struct {
    data  int
    left  *Node
    right *Node
}
```

1.Construct a binary search tree, insert node

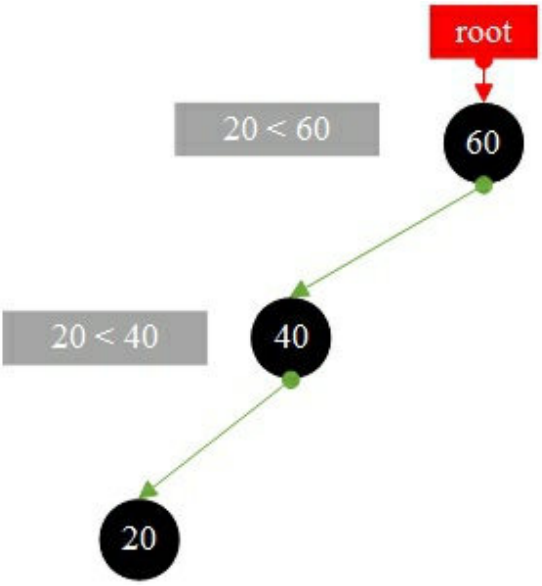
The inserted nodes are compared from the root node, and the smaller than the root node is compared with the left subtree of the root node, otherwise, compared with the right subtree until the left subtree is empty or the right subtree is empty, then is inserted.

Insert 60

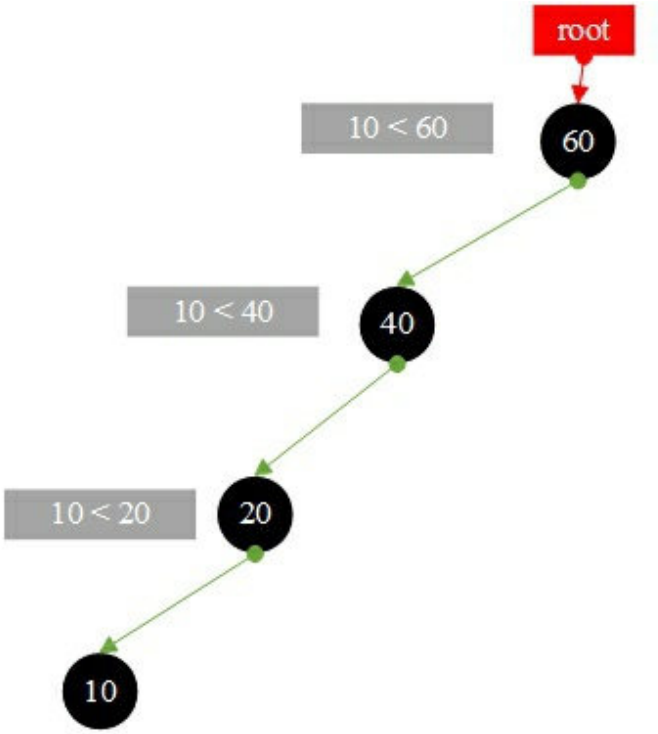
Insert 40



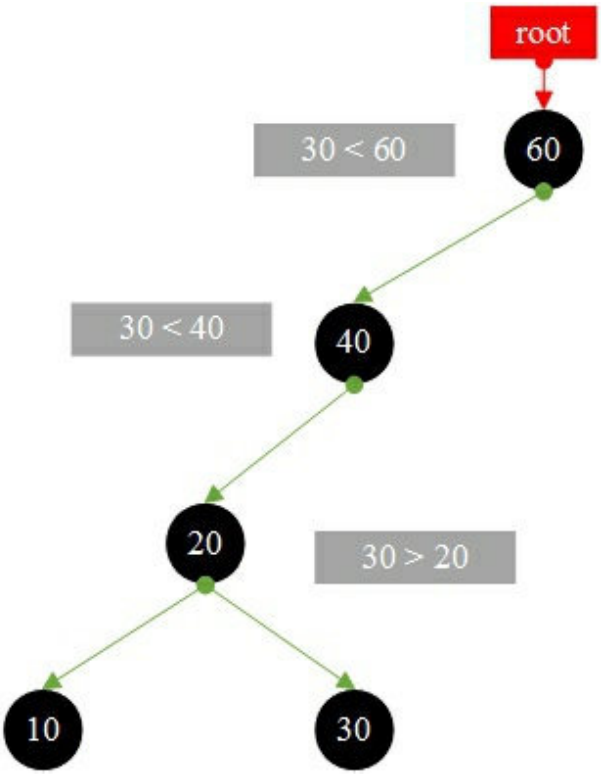
Insert 20



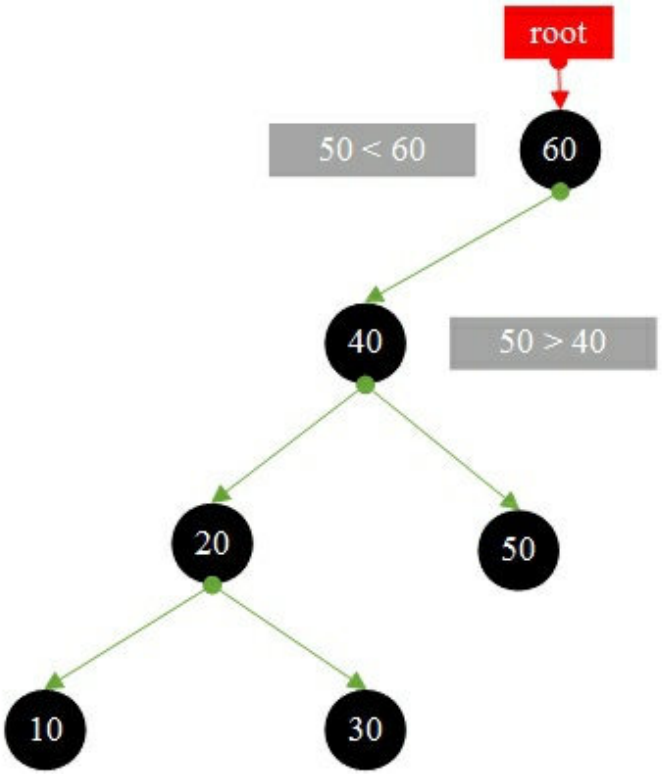
Insert 10



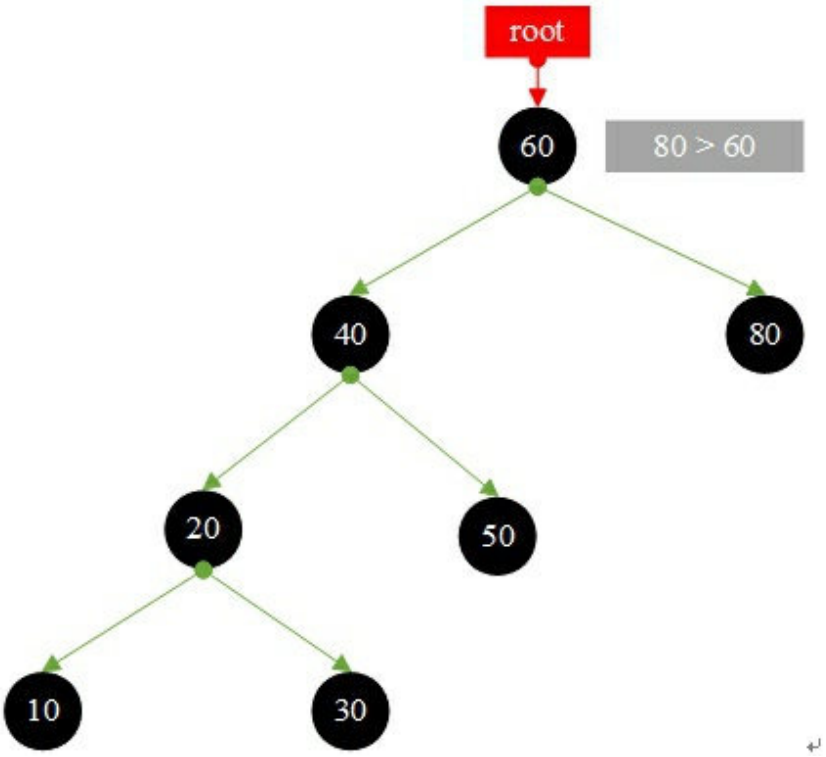
Insert 30



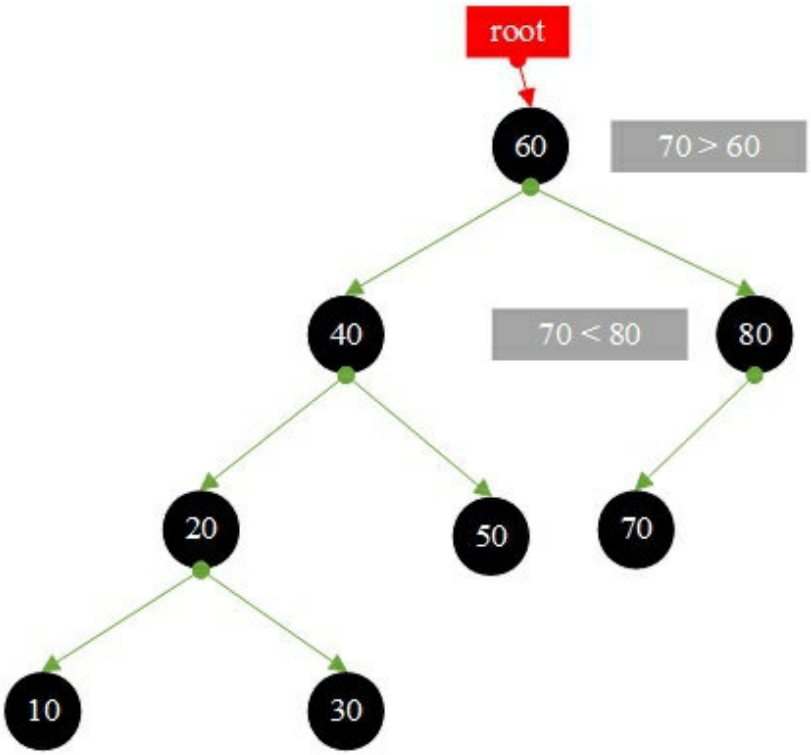
Insert 50



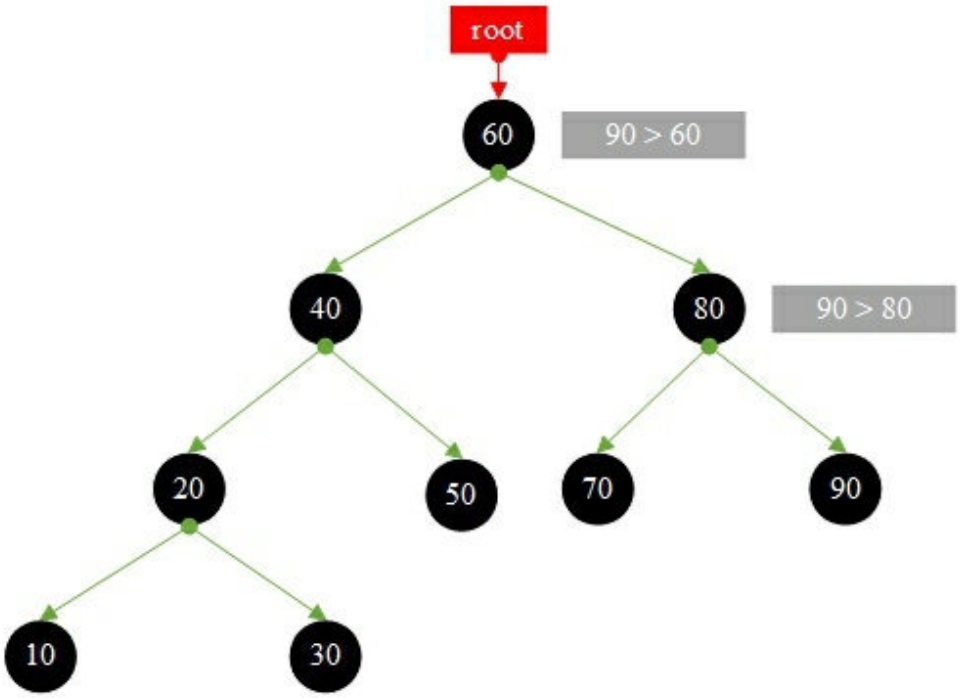
Insert 80



Insert 70

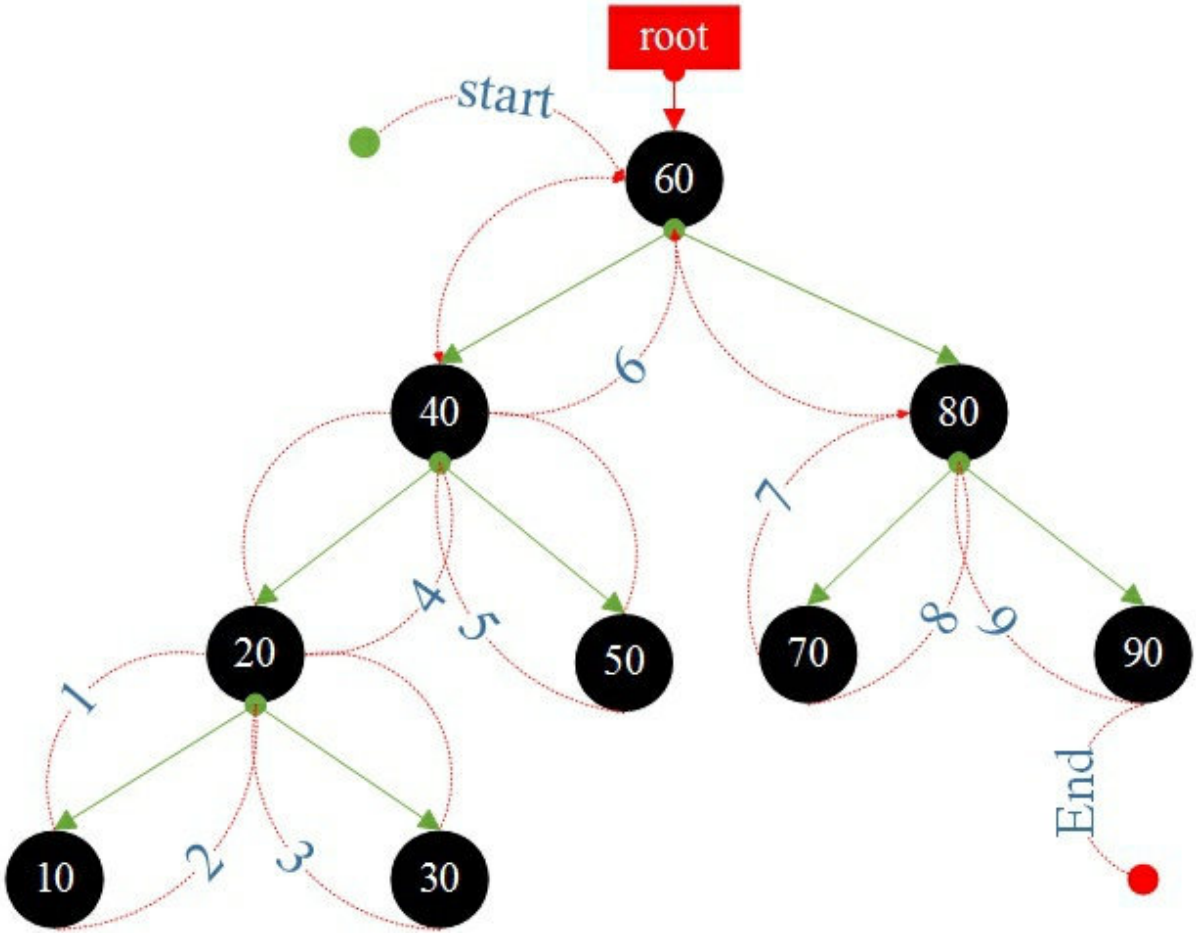


Insert 90

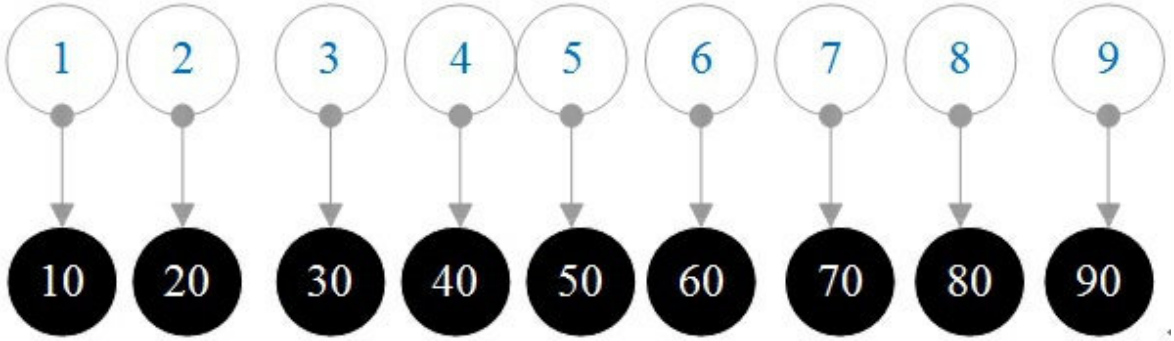


2. binary search tree In-order traversal

In-order traversal : left subtree -> root node -> right subtree



Result:



BinaryTree.go

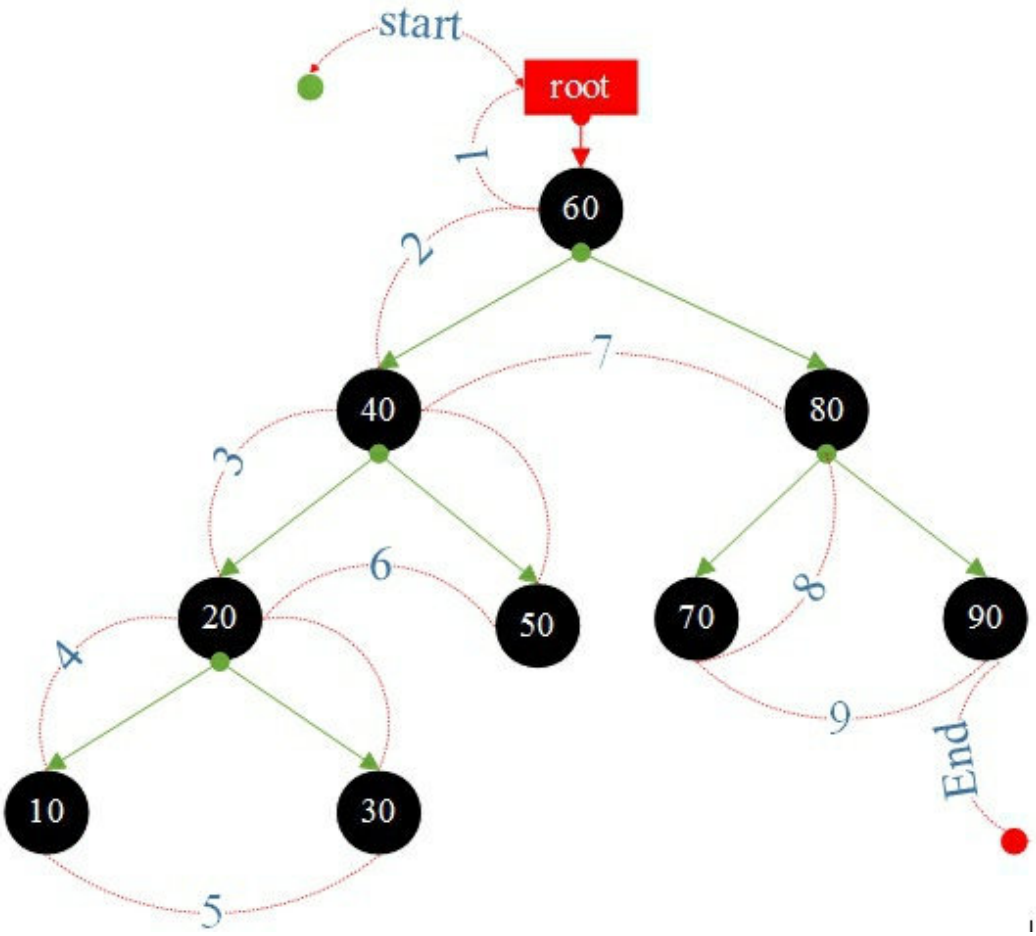
```
package main

import "fmt"

type Node struct {
    data  int
    left  *Node
    right *Node
}

var root *Node = nil

func createNewNode(newData int) *Node {
    var newNode *Node = new(Node)
    newNode.data = newData
    newNode.left = nil
    newNode.right = nil
    return newNode
}

// In-order traversal binary search tree
func inOrder(root *Node) {
    if root == nil {
        return
    }
    inOrder(root.left) // Traversing the left subtree
    fmt.Printf("%d, ", root.data)
    inOrder(root.right) // Traversing the right subtree
}
```

```
func insert(node *Node, newData int) {
    if root == nil {
        root = &Node{data: newData, left: nil, right: nil}
        return
    }
    var compareValue = newData - node.data
    //Recursive left subtree, continue to find the insertion position
    if compareValue < 0 {
        if node.left == nil {
            node.left = createNewNode(newData)
        } else {
            insert(node.left, newData)
        }
    } else if compareValue > 0 {
        //Recursive right subtree, continue to find the insertion position
        if node.right == nil {
            node.right = createNewNode(newData)
        } else {
            insert(node.right, newData)
        }
    }
}

func main() {
    //Constructing a binary search tree
    insert(root, 60)
    insert(root, 40)
    insert(root, 20)
    insert(root, 10)
    insert(root, 30)
    insert(root, 50)
    insert(root, 80)
    insert(root, 70)
    insert(root, 90)
    fmt.Printf("In-order traversal binary search tree \n")
    inOrder(root)
}
```

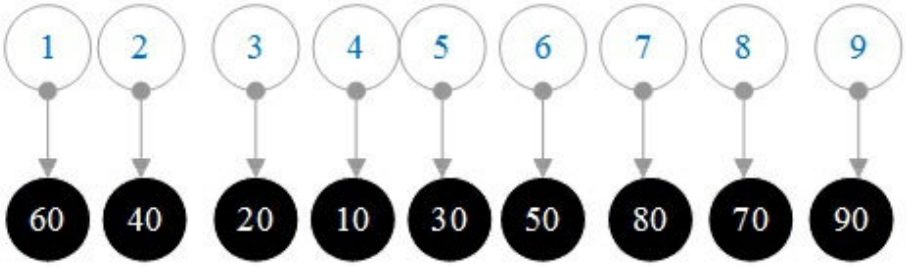
Result:
In-order traversal binary search tree
10, 20, 30, 40, 50, 60, 70, 80, 90,

3. binary search tree Pre-order traversal

Pre-order traversal : root node -> left subtree -> right subtree



Result:



BinaryTree.go

```
package main

import "fmt"

type Node struct {
    data  int
    left  *Node
    right *Node
}

var root *Node = nil

func createNewNode(newData int) *Node {
    var newNode *Node = new(Node)
    newNode.data = newData
    newNode.left = nil
    newNode.right = nil
    return newNode
}

//Preorder traversal binary search tree
func preOrder(root *Node) {
    if root == nil {
        return
    }
    fmt.Printf("%d, ", root.data)
    preOrder(root.left)  // Recursive Traversing the left subtree
    preOrder(root.right) // Recursive Traversing the right subtree
}
```

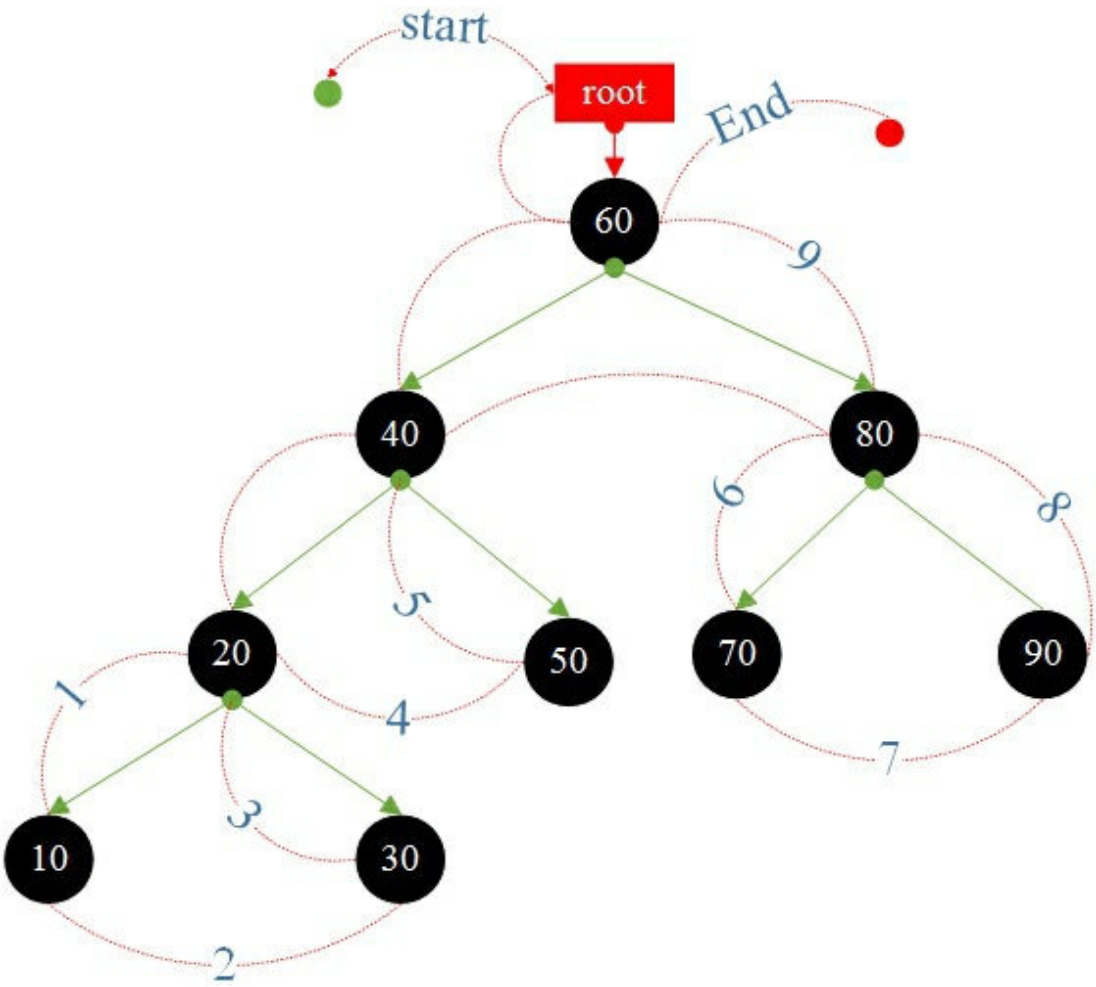
```
func insert(node *Node, newData int) {
    if root == nil {
        root = &Node{data: newData, left: nil, right: nil}
        return
    }
    var compareValue = newData - node.data
    //Recursive left subtree, continue to find the insertion position
    if compareValue < 0 {
        if node.left == nil {
            node.left = createNewNode(newData)
        } else {
            insert(node.left, newData)
        }
    } else if compareValue > 0 {
        //Recursive right subtree, continue to find the insertion position
        if node.right == nil {
            node.right = createNewNode(newData)
        } else {
            insert(node.right, newData)
        }
    }
}

func main() {
    //Constructing a binary search tree
    insert(root, 60)
    insert(root, 40)
    insert(root, 20)
    insert(root, 10)
    insert(root, 30)
    insert(root, 50)
    insert(root, 80)
    insert(root, 70)
    insert(root, 90)

    fmt.Printf("Pre-order traversal binary search tree \n")
    preOrder(root)
}
```

Result:

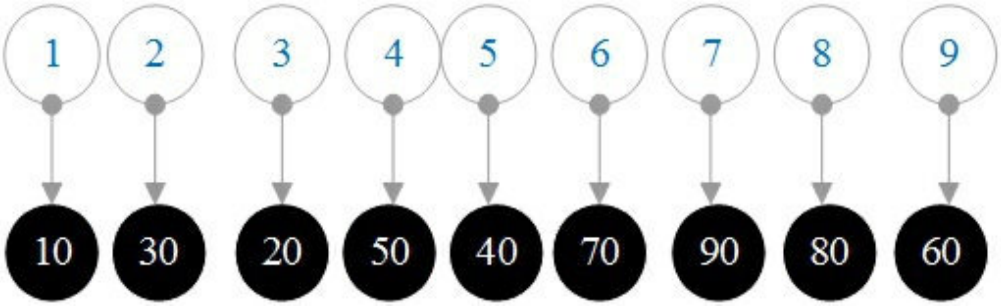
Pre-order traversal binary search tree
60, 40, 20, 10, 30, 50, 80, 70, 90,

4. binary search tree Post-order traversal

Post-order traversal : right subtree -> root node -> left subtree



Result:



BinaryTree.go

```
package main

import "fmt"

type Node struct {
    data  int
    left  *Node
    right *Node
}

var root *Node = nil

func createNewNode(newData int) *Node {
    var newNode *Node = new(Node)
    newNode.data = newData
    newNode.left = nil
    newNode.right = nil
    return newNode
}

//Post-order traversal binary search tree
func postOrder(root *Node) {
    if root == nil {
        return
    }

    postOrder(root.left)  // Recursive Traversing the left subtree
    postOrder(root.right) // Recursive Traversing the right subtree
    fmt.Printf("%d, ", root.data)
}
```

```
func insert(node *Node, newData int) {
    if root == nil {
        root = &Node{data: newData, left: nil, right: nil}
        return
    }
    var compareValue = newData - node.data
    //Recursive left subtree, continue to find the insertion position
    if compareValue < 0 {
        if node.left == nil {
            node.left = createNewNode(newData)
        } else {
            insert(node.left, newData)
        }
    } else if compareValue > 0 {
        //Recursive right subtree, continue to find the insertion position
        if node.right == nil {
            node.right = createNewNode(newData)
        } else {
            insert(node.right, newData)
        }
    }
}

func main() {
    //Constructing a binary search tree
    insert(root, 60)
    insert(root, 40)
    insert(root, 20)
    insert(root, 10)
    insert(root, 30)
    insert(root, 50)
    insert(root, 80)
    insert(root, 70)
    insert(root, 90)
    fmt.Printf("Post-order traversal binary search tree \n")
    postOrder(root)
}
```

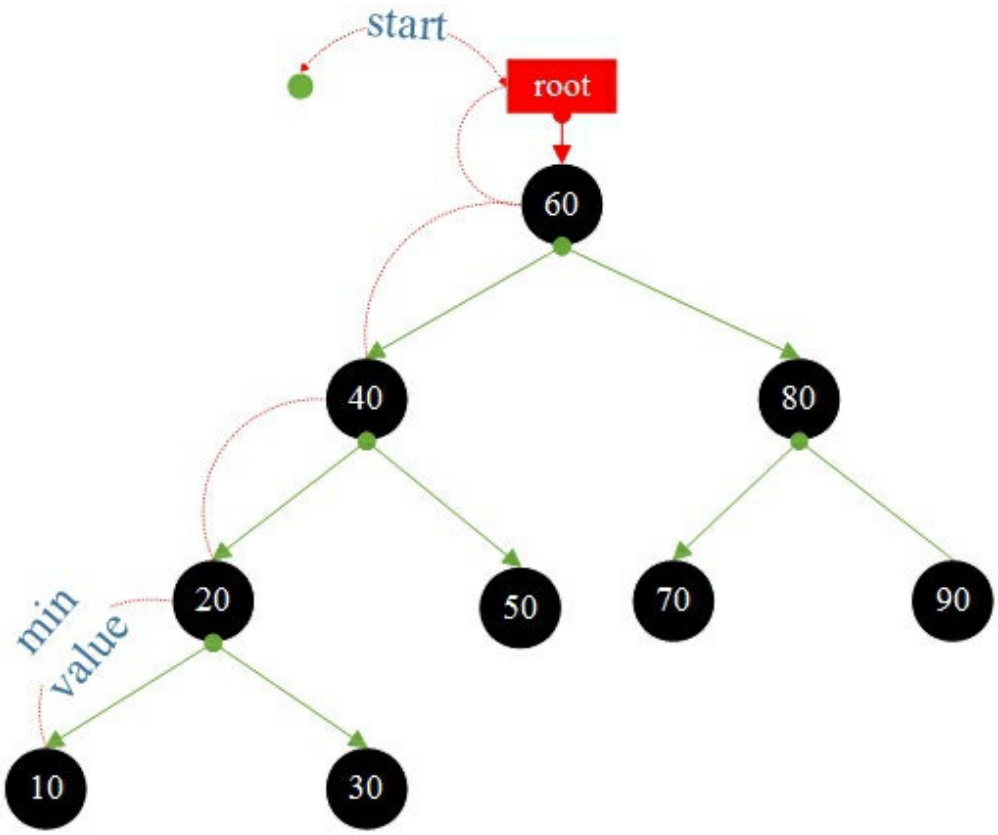
Result:

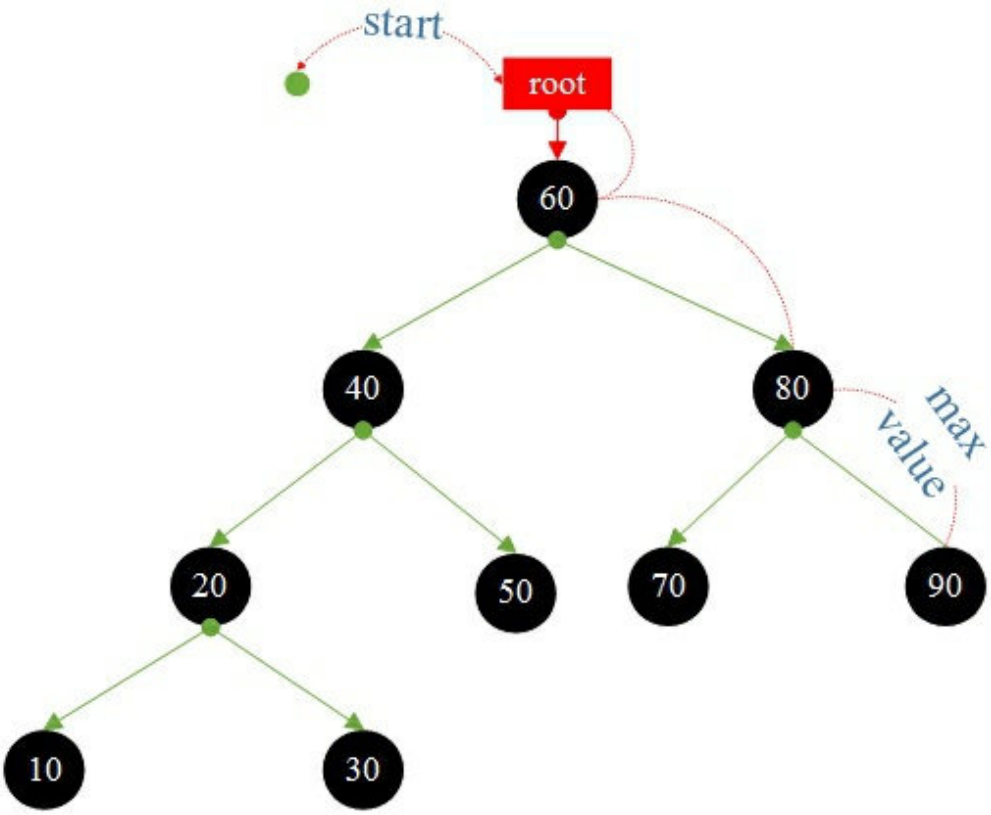
Post-order traversal binary search tree
10, 30, 20, 50, 40, 70, 90, 80, 60,

5. binary search tree Maximum and minimum

Minimum value: The small value is on the left child node, as long as the recursion traverses the left child until be empty, the current node is the minimum node.

Maximum value: The large value is on the right child node, as long as the recursive traversal is the right child until be empty, the current node is the largest node.




start
root
60
40
80
max
value
20
50
70
90
10
30

BinaryTree.go

```
package main

import "fmt"

type Node struct {
    data  int
    left  *Node
    right *Node
}

var root *Node = nil

func createNewNode(newData int) *Node {
    var newNode *Node = new(Node)
    newNode.data = newData
    newNode.left = nil
    newNode.right = nil
    return newNode
}

func searchMinValue(node *Node) *Node { //Minimum value
    if node == nil || node.data == 0 {
        return nil
    }
    if node.left == nil {
        return node
    }
    return searchMinValue(node.left) //Recursively find the minimum
from the left subtree
}

func searchMaxValue(node *Node) *Node { //Maximum value
    if node == nil || node.data == 0 {
        return nil
    }
    if node.right == nil {
        return node
```

```
    }
    return searchMaxValue(node.right) //Recursively find the minimum from the right subtree
}

func insert(node *Node, newData int) {
    if root == nil {
        root = &Node{data: newData, left: nil, right: nil}
        return
    }
    var compareValue = newData - node.data
    //Recursive left subtree, continue to find the insertion position
    if compareValue < 0 {
        if node.left == nil {
            node.left = createNewNode(newData)
        } else {
            insert(node.left, newData)
        }
    } else if compareValue > 0 {
        //Recursive right subtree, continue to find the insertion position
        if node.right == nil {
            node.right = createNewNode(newData)
        } else {
            insert(node.right, newData)
        }
    }
}

func main() {
    //Constructing a binary search tree
    insert(root, 60)
    insert(root, 40)
    insert(root, 20)
    insert(root, 10)
    insert(root, 30)
    insert(root, 50)
    insert(root, 80)
```

```
    insert(root, 70)
    insert(root, 90)

    fmt.Printf("\nMinimum Value \n")
    var minNode = searchMinValue(root)
    fmt.Printf("%d", minNode.data)

    fmt.Printf("\nMaximum Value \n")
    var maxNode = searchMaxValue(root)
    fmt.Printf("%d", maxNode.data)
}
```

Result:

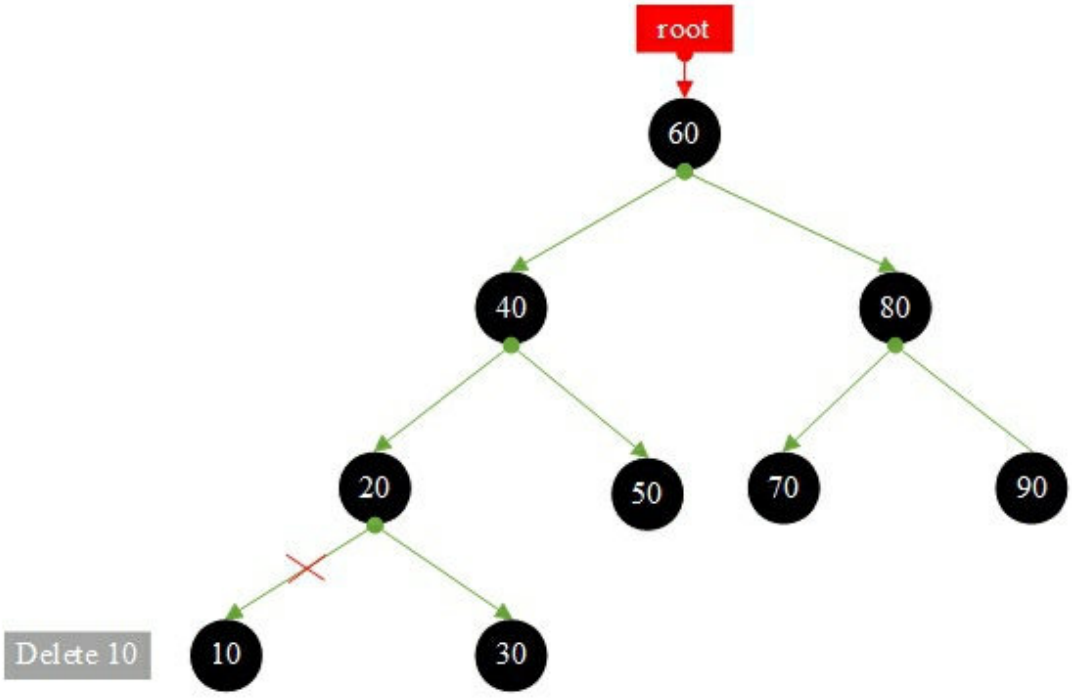
Minimum Value
10

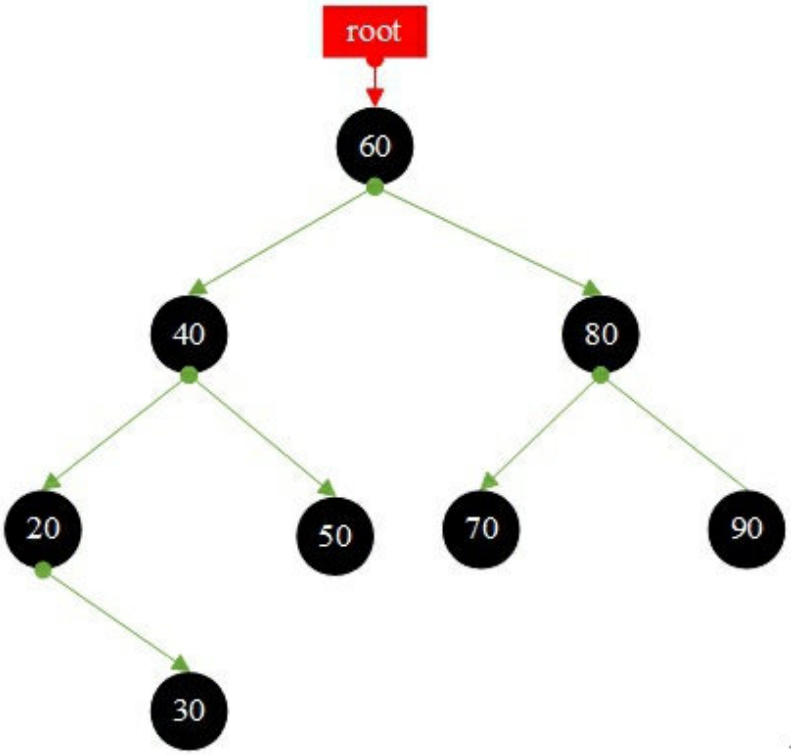
Maximum Value
90

6. binary search tree Delete Node

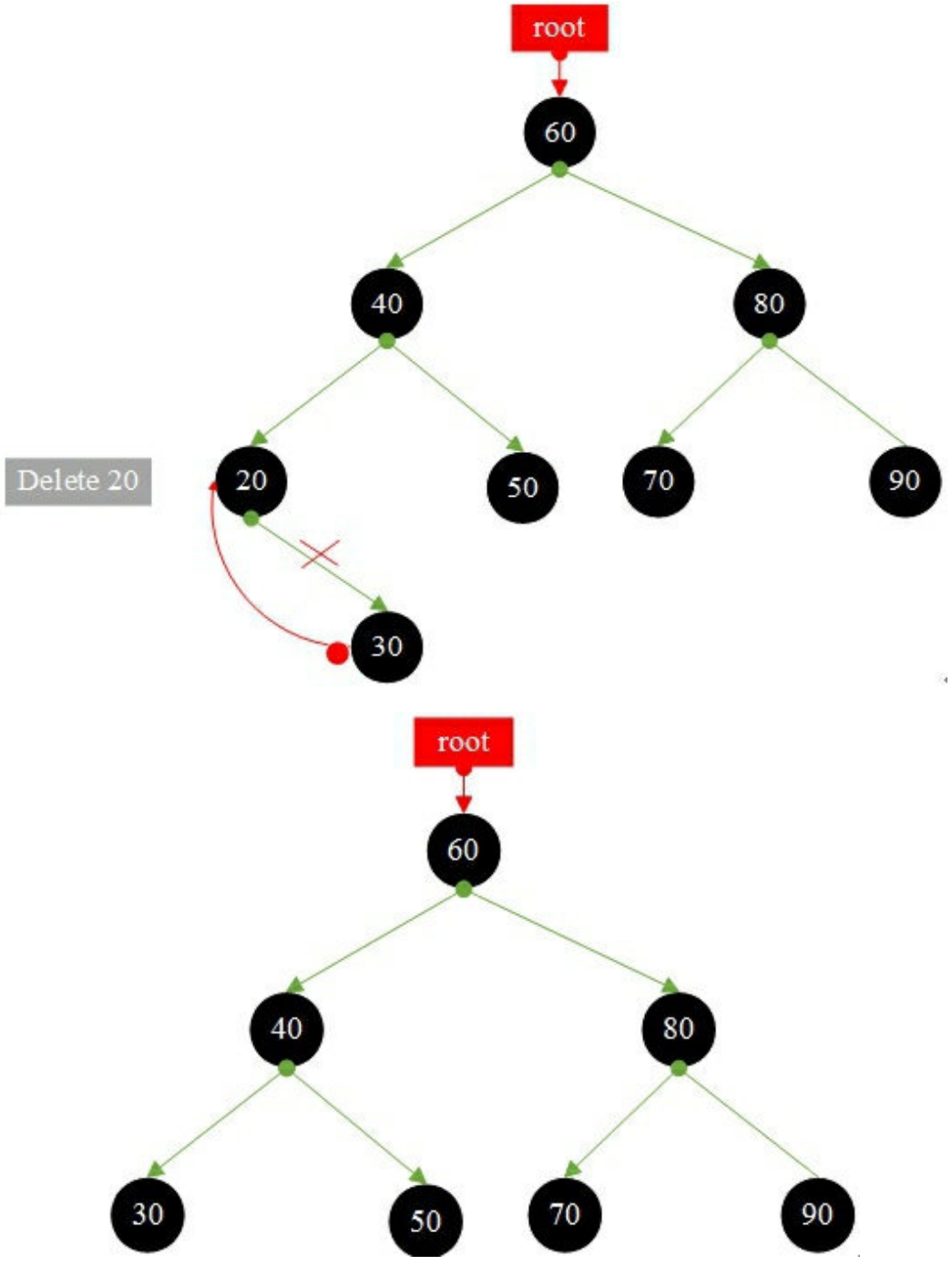
Binary search tree delete node 3 cases
1. If there is no child node, delete it directly
2. If there is only one child node, the child node replaces the current node, and then deletes the current node.
3. If there are two child nodes, replace the current node with the smallest node from the right subtree, because the smallest node on the right is also larger than the value on the left.

1. If there is no child node, delete it directly: delete node 10

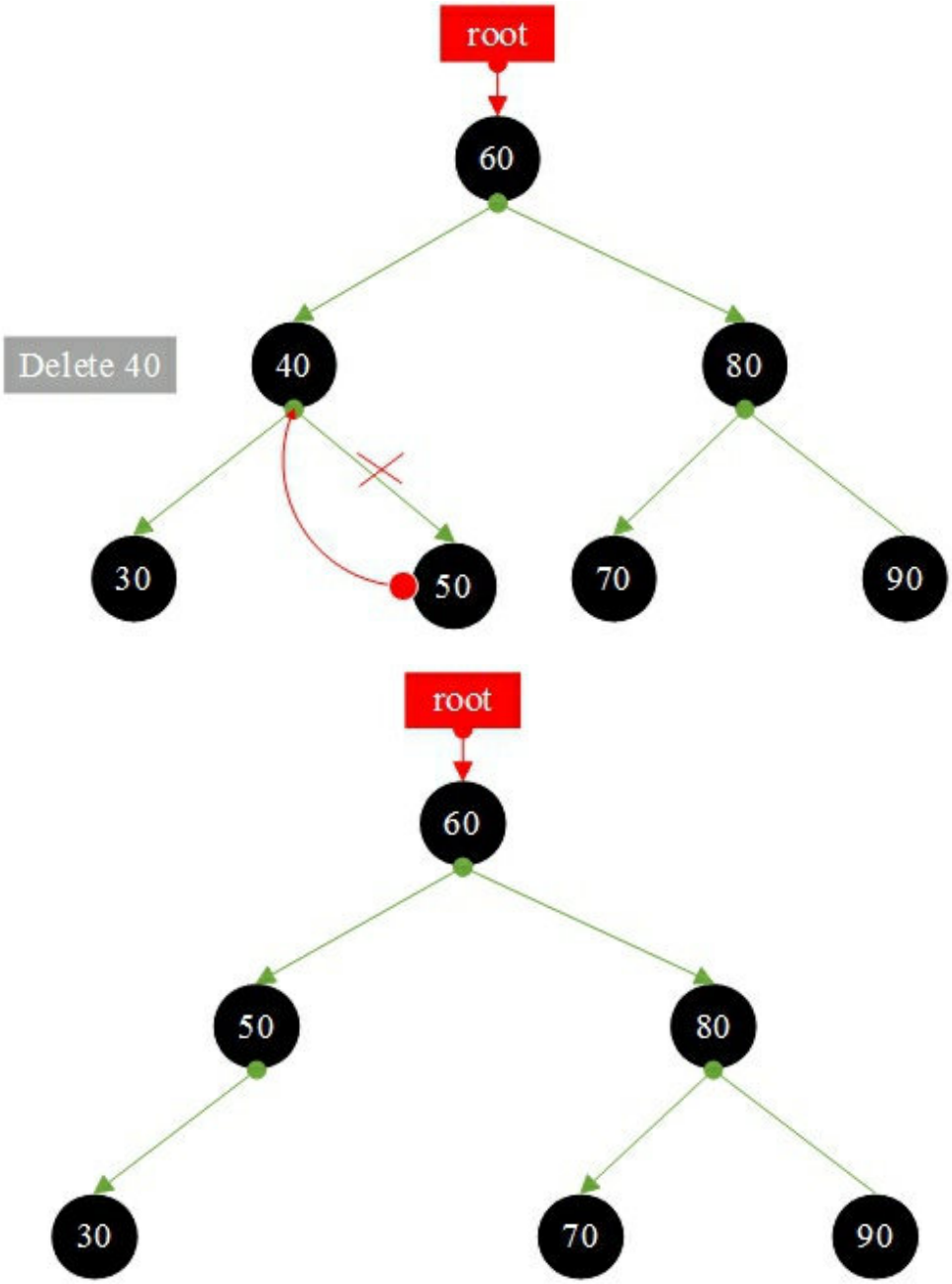



root
60
40
80
20
50
70
90
30

2. If there is only one child node, the child node replaces the current node, and then deletes the current node. Delete node 20



3. If there are two child nodes, replace the current node with the smallest node from the right subtree, Delete node 40



BinaryTree.go

```
package main

import "fmt"

type Node struct {
    data  int
    left  *Node
    right *Node
}

var root *Node = nil

func createNewNode(newData int) *Node {
    var newNode *Node = new(Node)
    newNode.data = newData
    newNode.left = nil
    newNode.right = nil
    return newNode
}

func searchMinValue(node *Node) *Node { //Minimum value
    if node == nil || node.data == 0 {
        return nil
    }
    if node.left == nil {
        return node
    }
    return searchMinValue(node.left) //Recursively find the minimum
from the left subtree
}

// In-order traversal binary search tree
func inOrder(root *Node) {
    if root == nil {
        return
    }
    inOrder(root.left) // Traversing the left subtree
```

```
    fmt.Printf("%d, ", root.data)
    inOrder(root.right) // Traversing the right subtree
}

func removeNode(node *Node, newData int) *Node {
    if node == nil {
        return node
    }
    var compareValue = newData - node.data
    if compareValue > 0 {
        node.right = removeNode(node.right, newData)
    } else if compareValue < 0 {
        node.left = removeNode(node.left, newData)
    } else if node.left != nil && node.right != nil {
        //Find the minimum node of the right subtree to replace the current
node
        node.data = searchMinValue(node.right).data
        node.right = removeNode(node.right, node.data)
    } else {
        if node.left != nil {
            node = node.left
        } else {
            node = node.right
        }
    }
    return node
}

func insert(node *Node, newData int) {
    if root == nil {
        root = &Node{data: newData, left: nil, right: nil}
        return
    }
    var compareValue = newData - node.data
    //Recursive left subtree, continue to find the insertion position
    if compareValue < 0 {
        if node.left == nil {
```

```
            node.left = createNewNode(newData)
        } else {
            insert(node.left, newData)
        }
    } else if compareValue > 0 {//Recursive right subtree
        if node.right == nil {
            node.right = createNewNode(newData)
        } else {
            insert(node.right, newData)
        }
    }
}

func main() {
    //Constructing a binary search tree
    insert(root, 60)
    insert(root, 40)
    insert(root, 20)
    insert(root, 10)
    insert(root, 30)
    insert(root, 50)
    insert(root, 80)
    insert(root, 70)
    insert(root, 90)

    fmt.Printf("\ndelete node is:  10 \n")
    removeNode(root, 10)

    fmt.Printf("\nIn-order traversal binary tree \n")
    inOrder(root)

    fmt.Printf("\n----------------------------------------\n")

    fmt.Printf("\ndelete node is:  20 \n")
    removeNode(root, 20)

    fmt.Printf("\nIn-order traversal binary tree \n")
    inOrder(root)
```

```
    fmt.Printf("\n----------------------------------------\n")

    fmt.Printf("\ndelete node is:  40 \n")
    removeNode(root, 40)

    fmt.Printf("\nIn-order traversal binary tree \n")
    inOrder(root)

}
```

Result:

delete node is: 10

In-order traversal binary tree
20, 30, 40, 50, 60, 70, 80, 90,
---

delete node is: 20

In-order traversal binary tree
30, 40, 50, 60, 70, 80, 90,
---

delete node is: 40

In-order traversal binary tree
30, 50, 60, 70, 80, 90,

Binary Heap Sorting

Binary Heap Sorting:
The value of the non-terminal node in the binary tree is not greater than the value of its left and right child nodes.

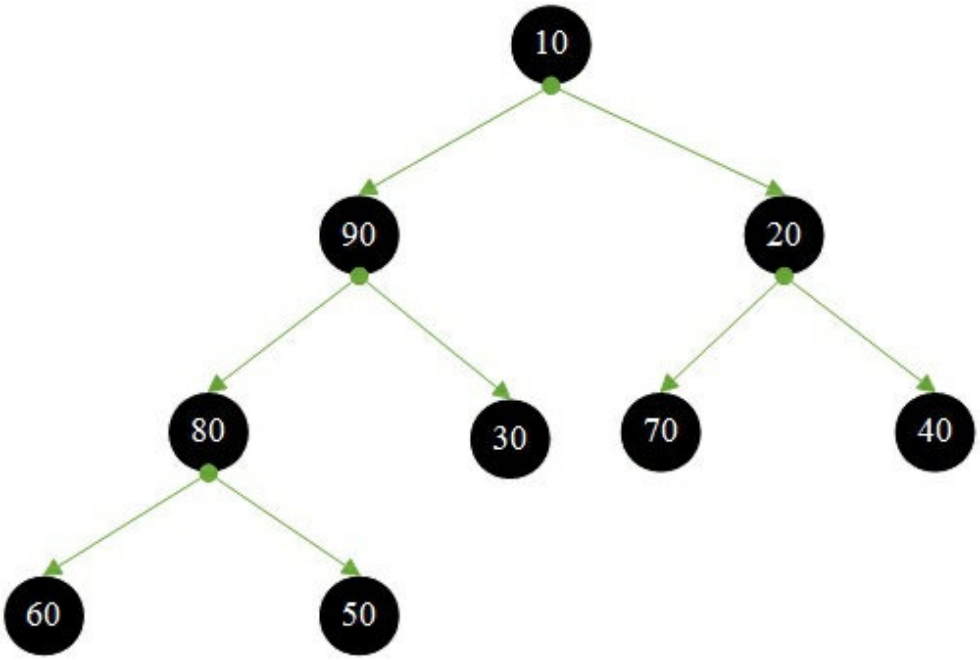
Small top heap : ki <= k2i and ki <= k2i+1
Big top heap :ki >= k2i and ki >= k2i+1

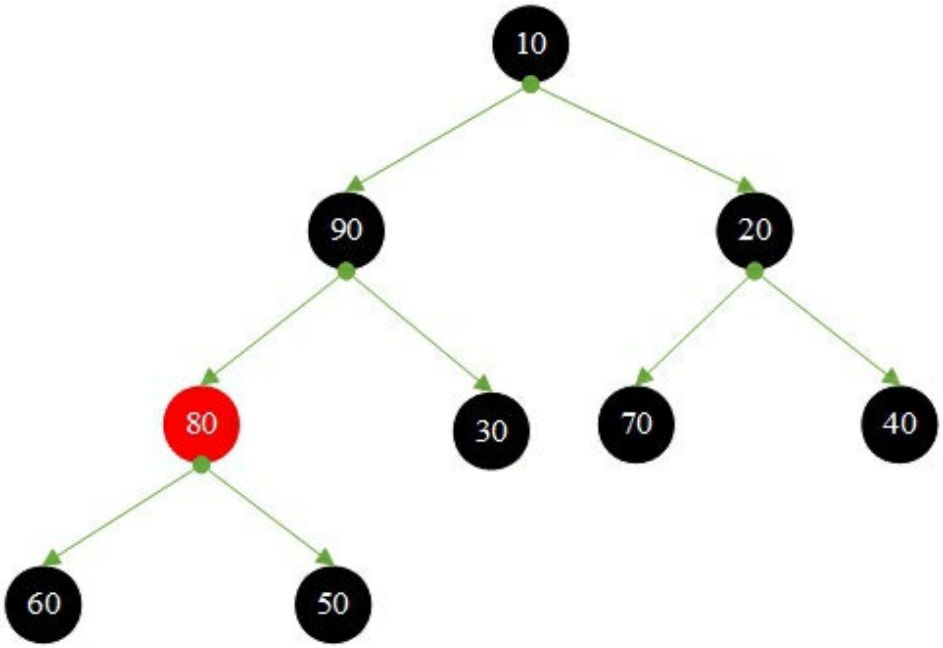
Parent node subscript = (i-1)/2
Left subnode subscript = 2*i+1
Right subnode subscript = 2*i+2

Heap sorting process:
1. Build a heap
2. After outputting the top element of the heap, adjust from top to bottom, compare the top element with the root node of its left and right subtrees, and swap the smallest element to the top of the heap; then adjust continuously until the leaf nodes to get new heap.

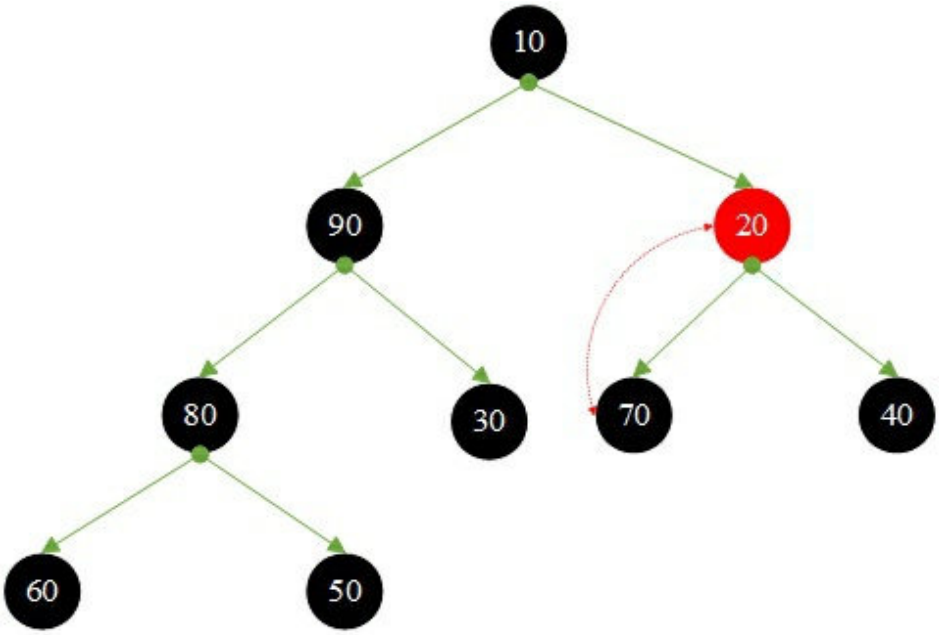
1. {10, 90, 20, 80, 30, 70, 40, 60, 50} build heap and then heap sort output.

Initialize the heap and build the heap

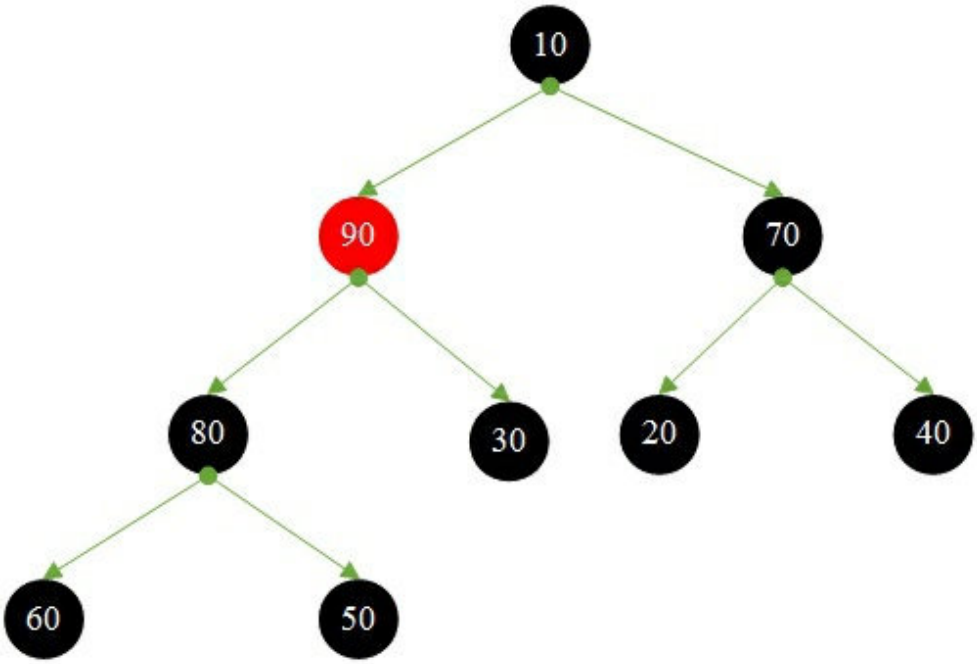




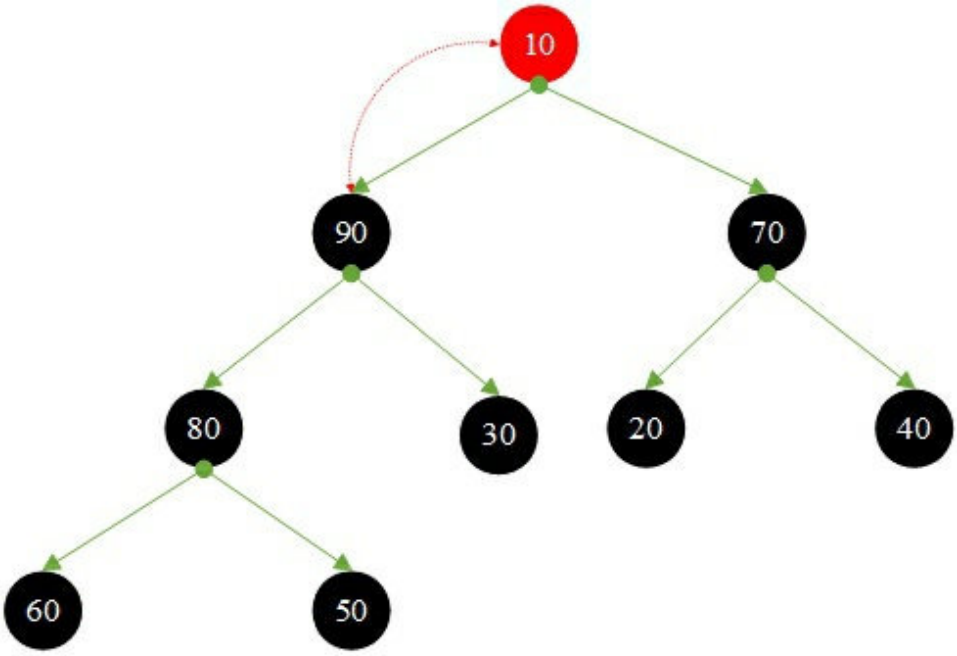
Not Leaf Node = 80 > left = 60 , 80 > right = 50 No need to move



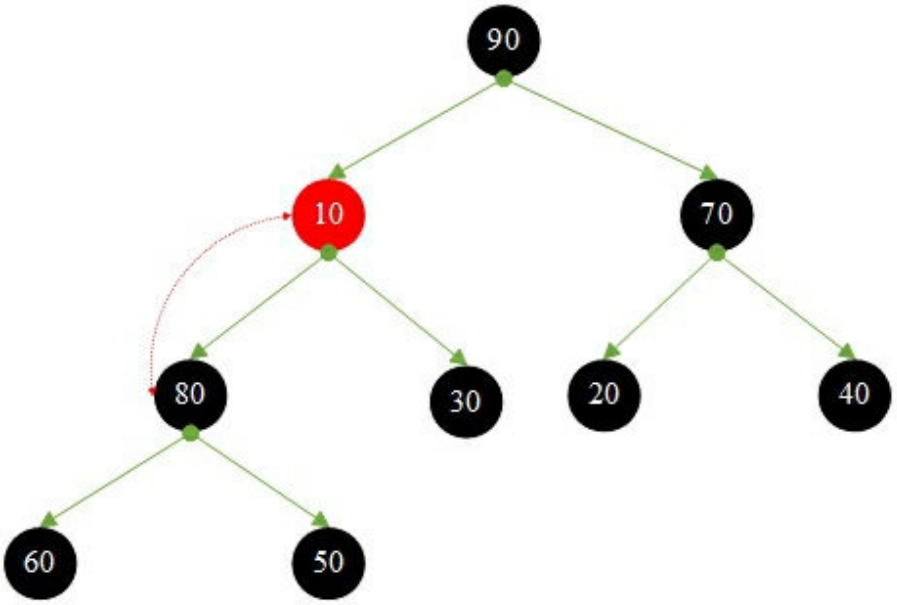
Not Leaf Node = 20 < left = 70 , 70 > right = 40 , 20 swap with 70



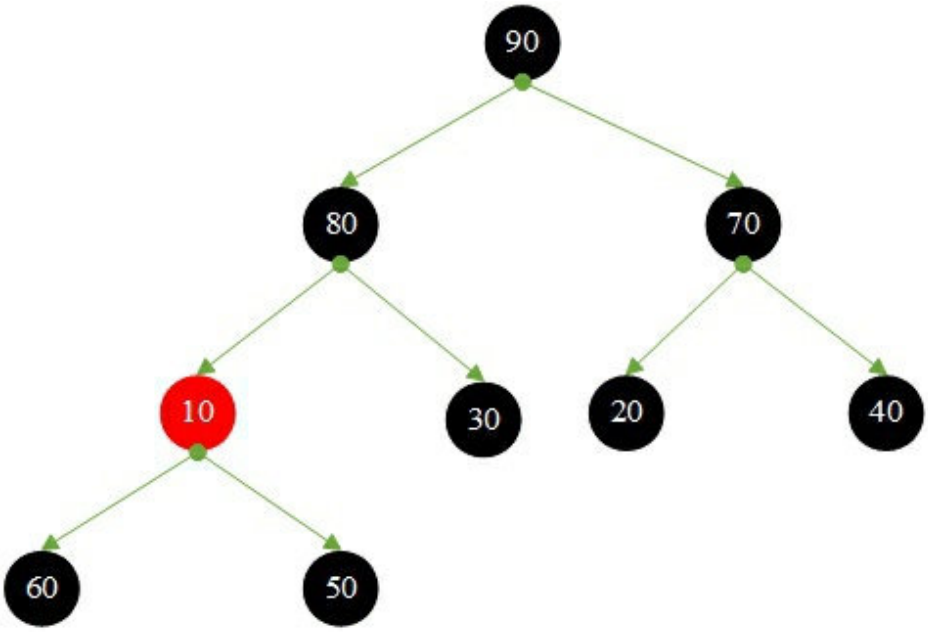
Not Leaf Node = 90 > left = 80 , 80 > right = 30 No need to move



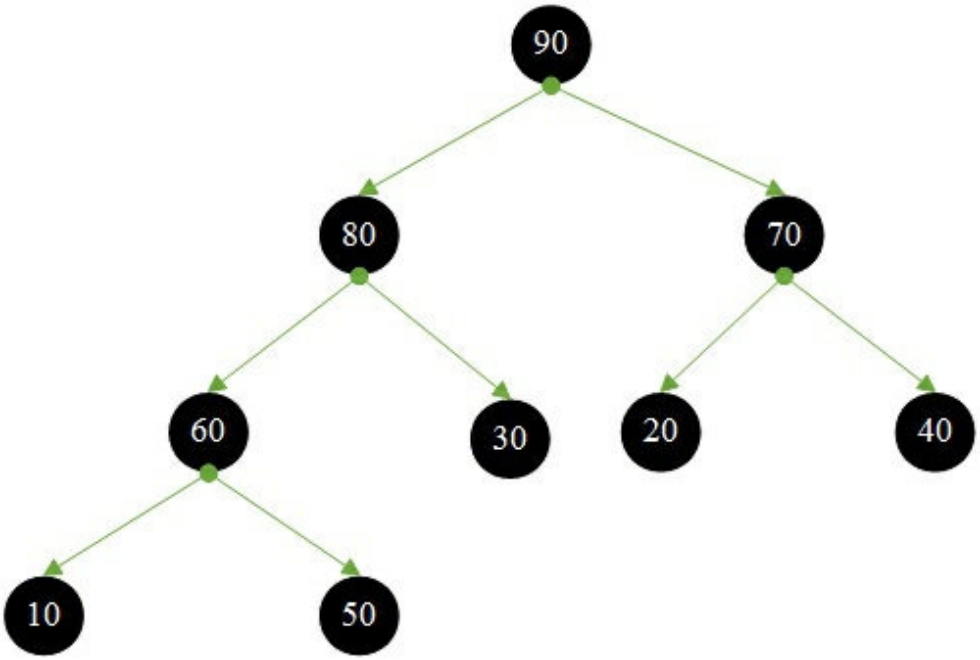
Not Leaf Node = 10 < left = 90 , 90 > right = 70 , 10 swap with 90



Still Not Leaf Node = 10 <1eft = 80 , 80 > right =30 , 10 swap with 80

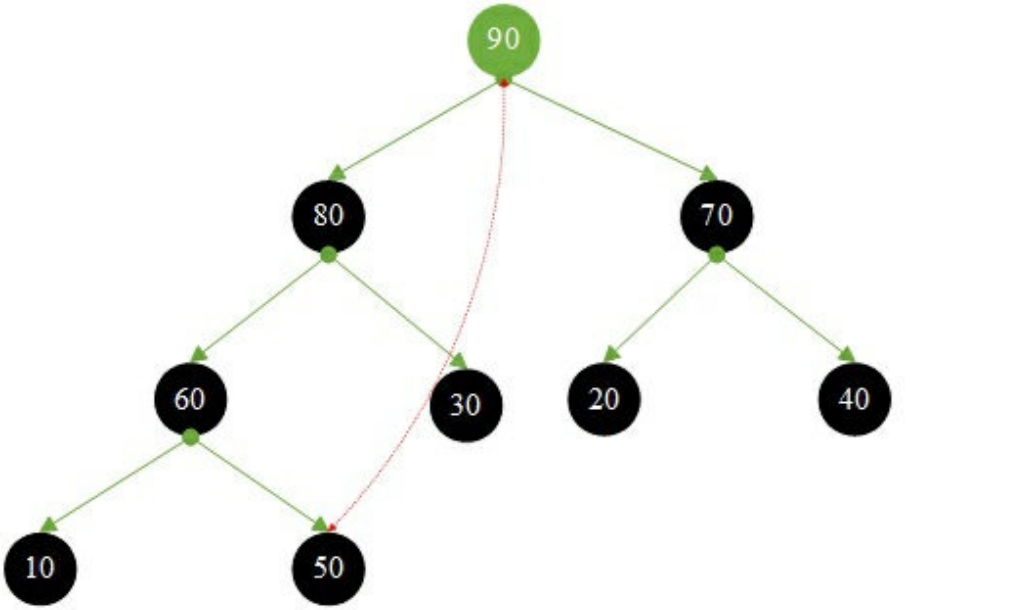


Still Not Leaf Node = 10 < left = 60 , 60 > right =50 , 10 swap with 60

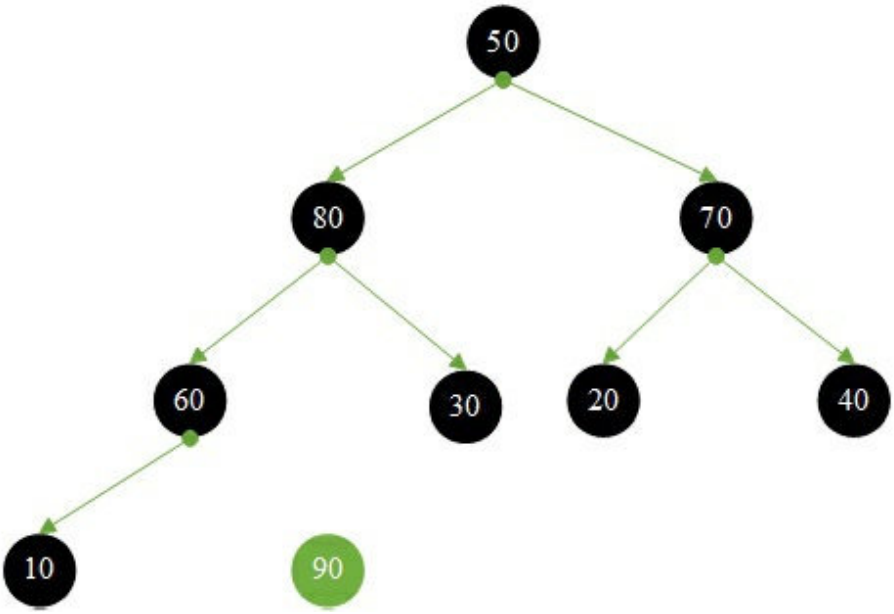


Create the heap finished

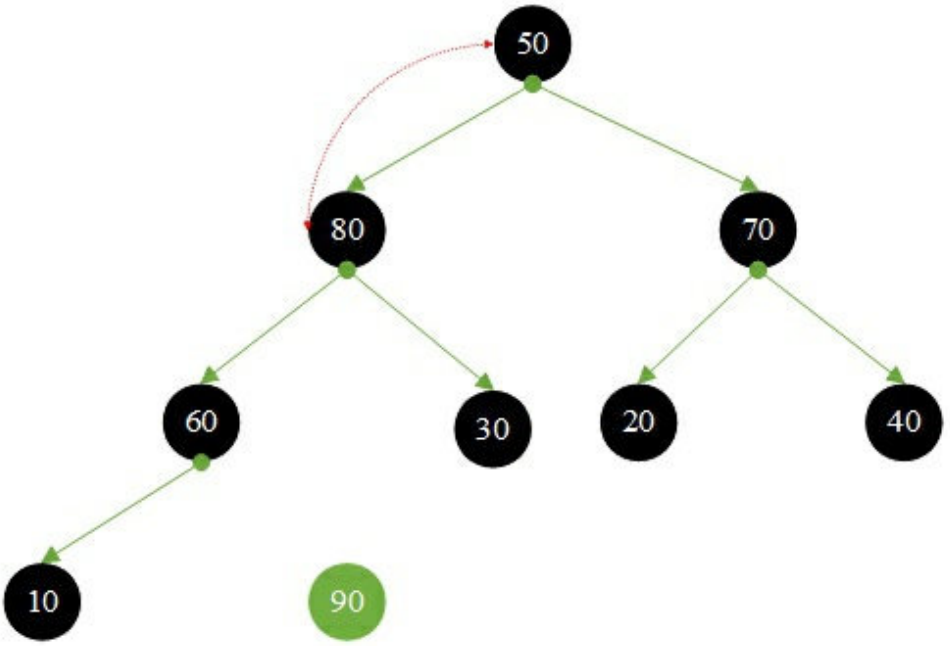
2. Start heap sorting

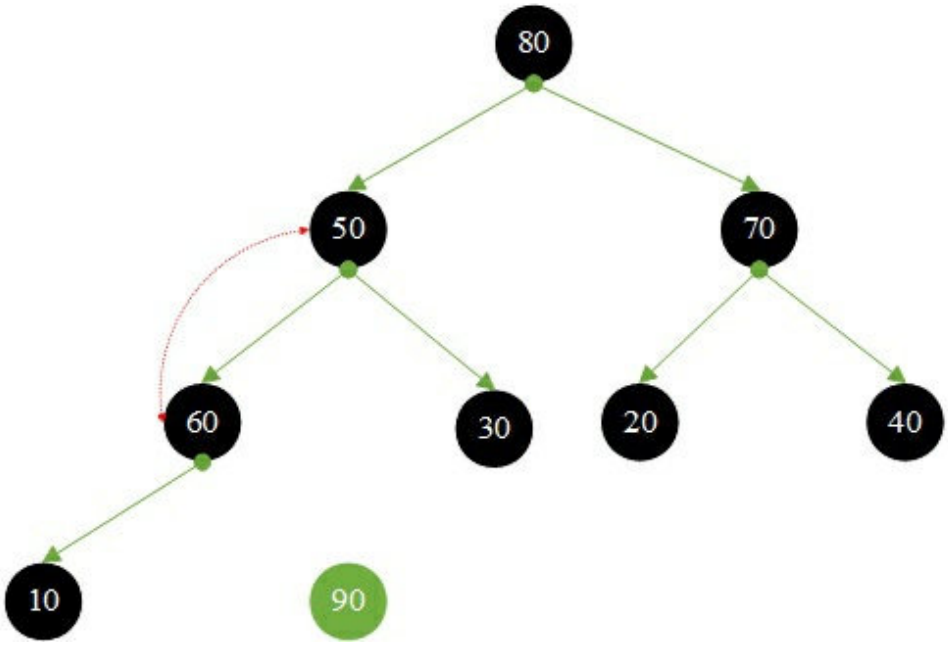


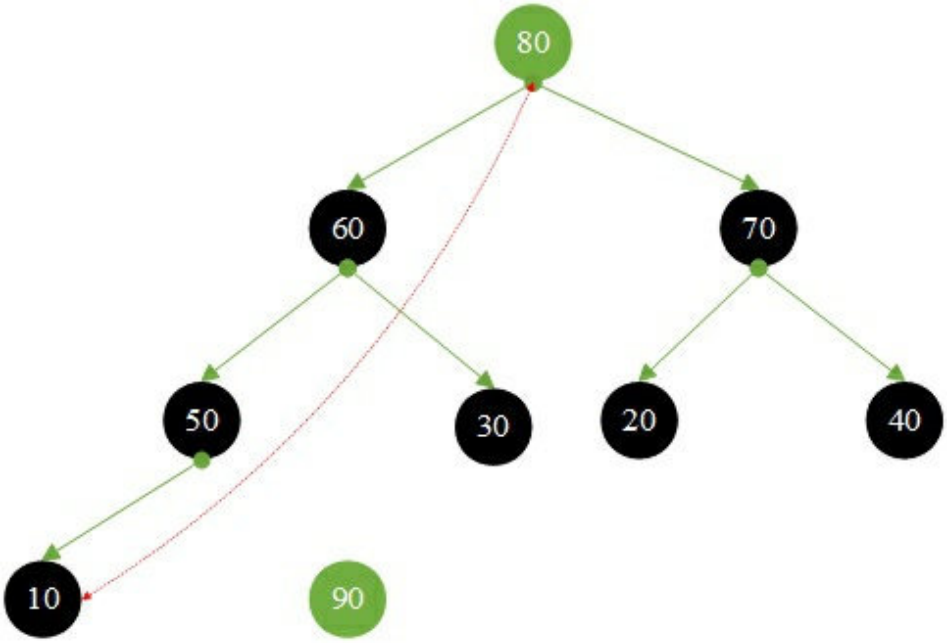
root = 90 and tail = 50 are exchanged



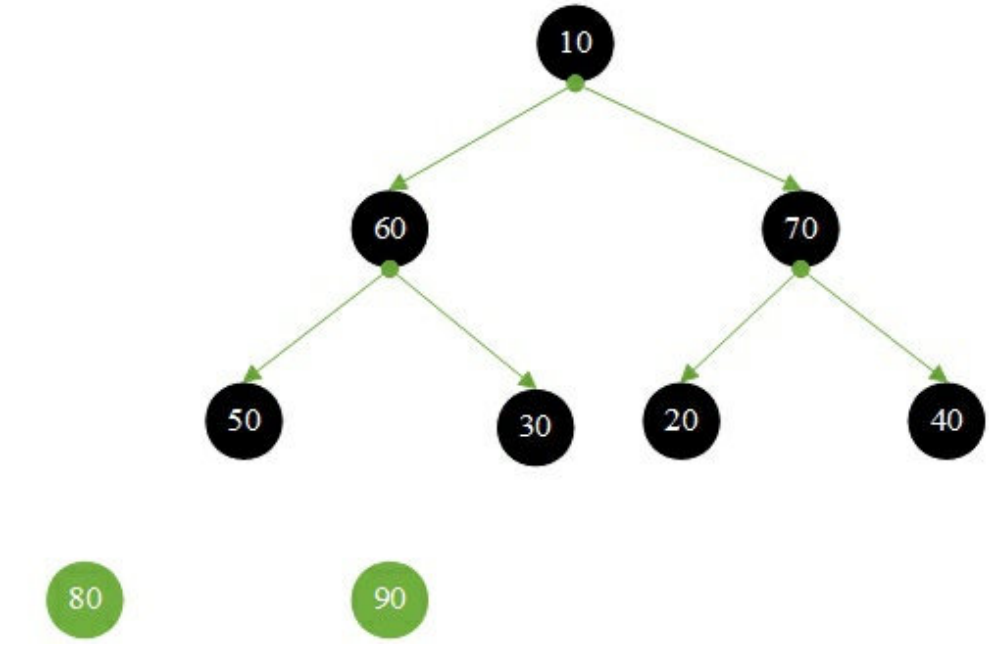
adjust the heap



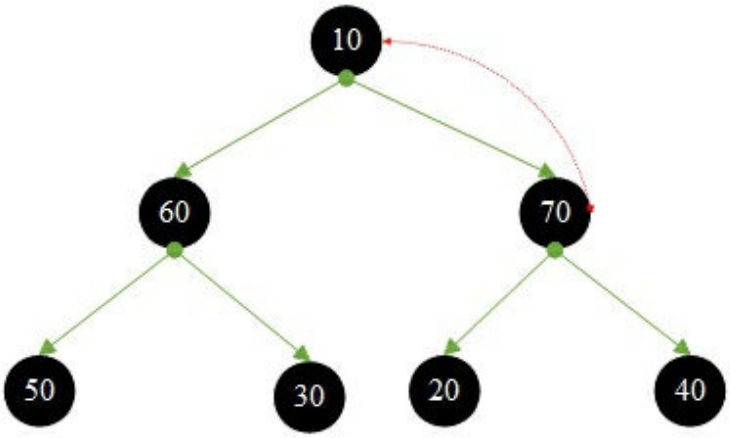




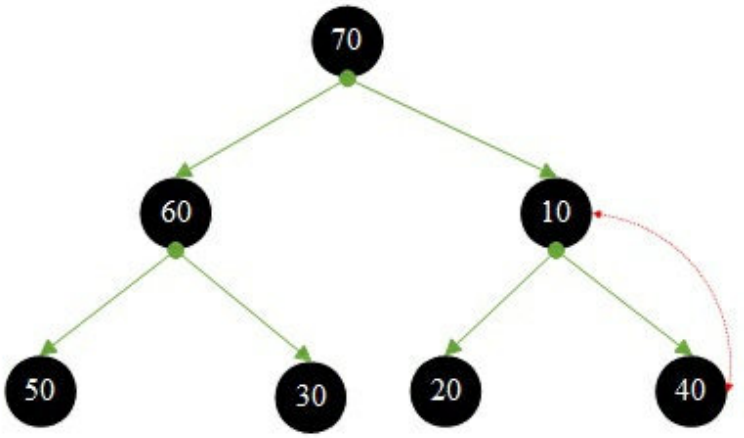
root = 80 and tail = 10 are exchanged



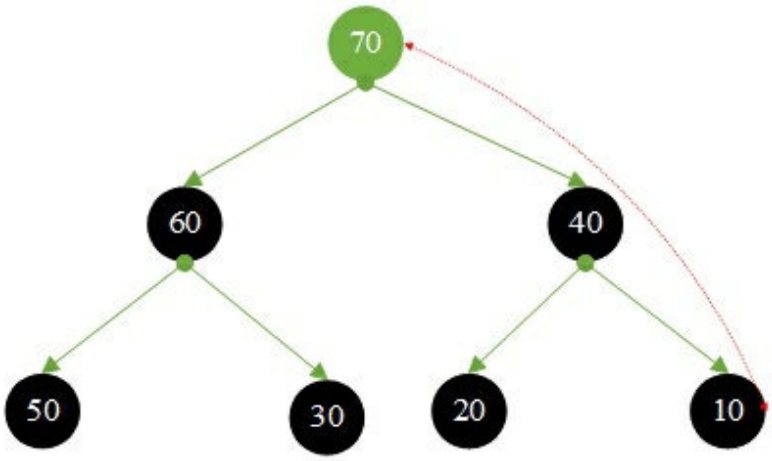
adjust the heap



80 90

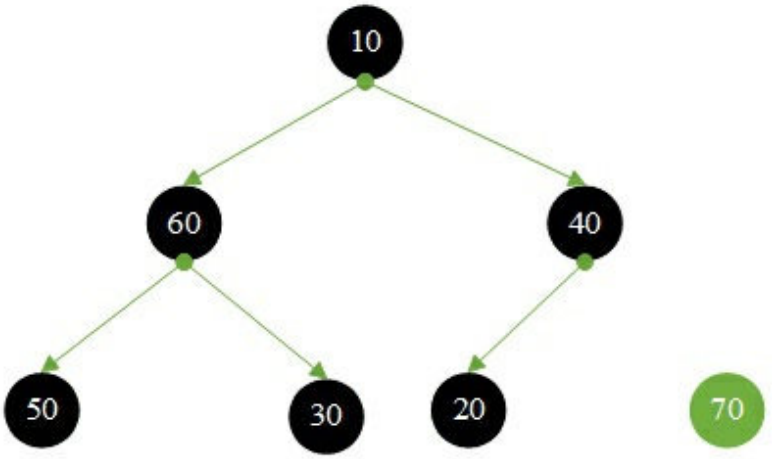


80 90



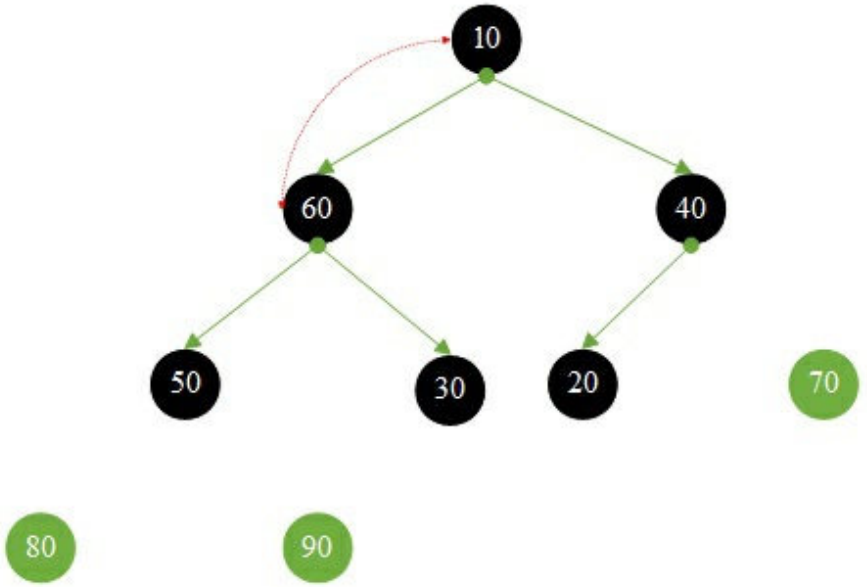
80 90

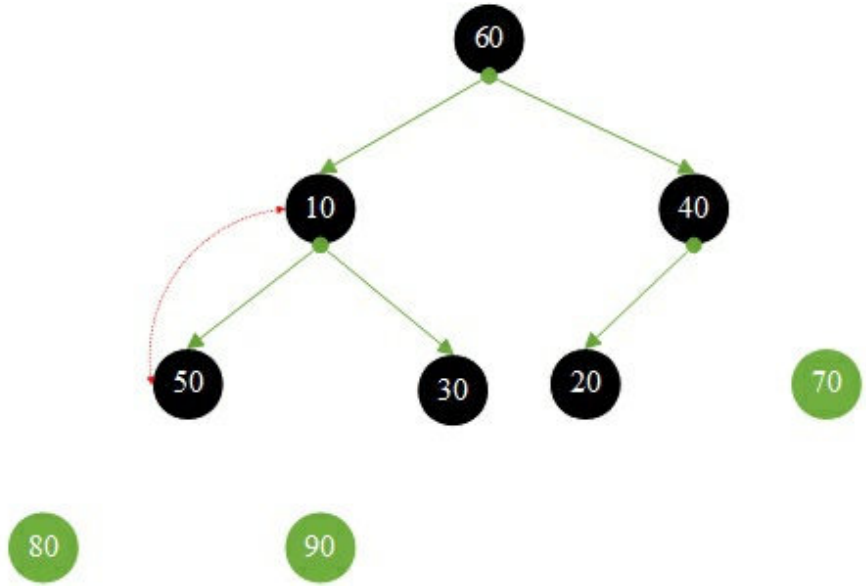
root = 70 and tail = 10 are exchanged

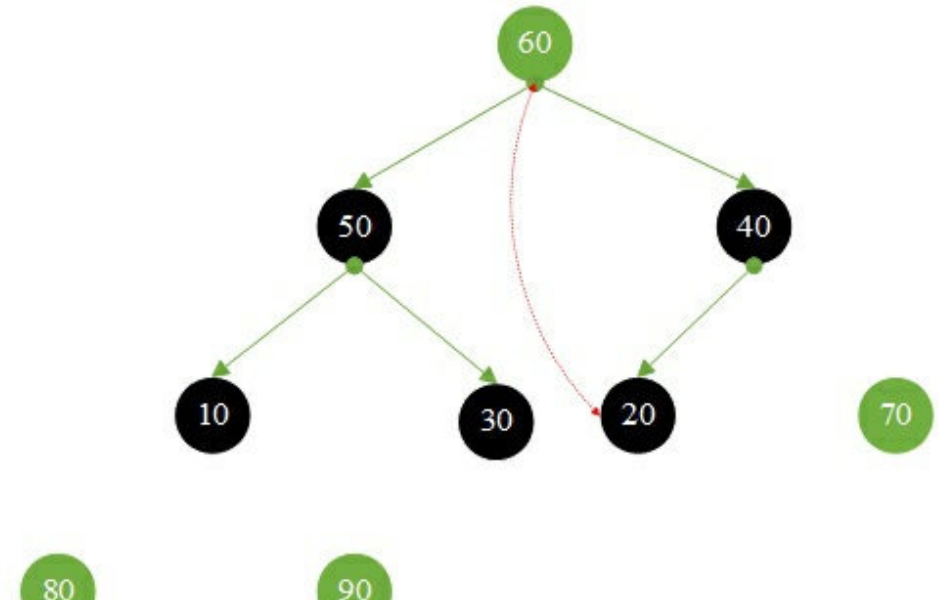


80 90

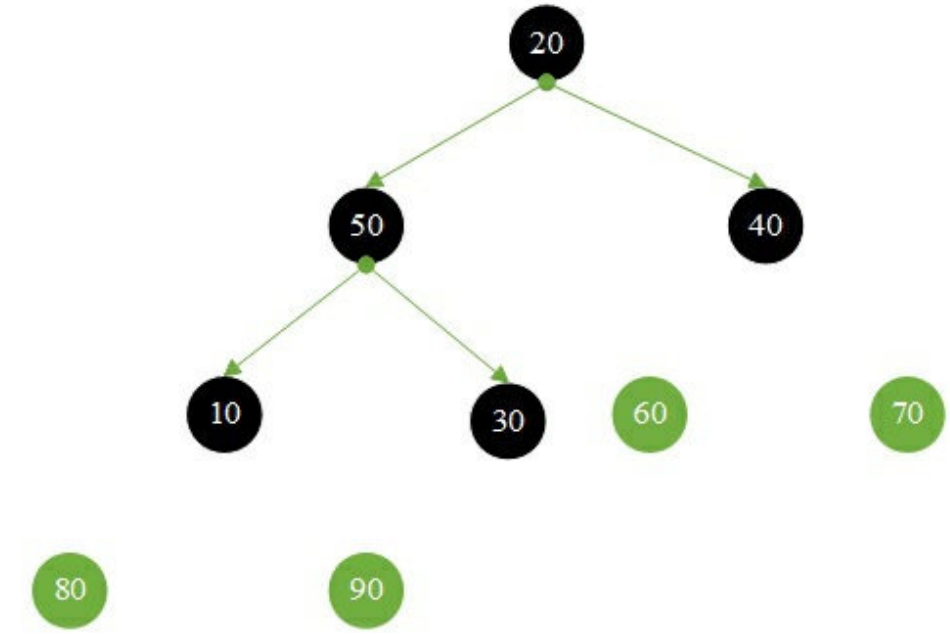
adjust the heap



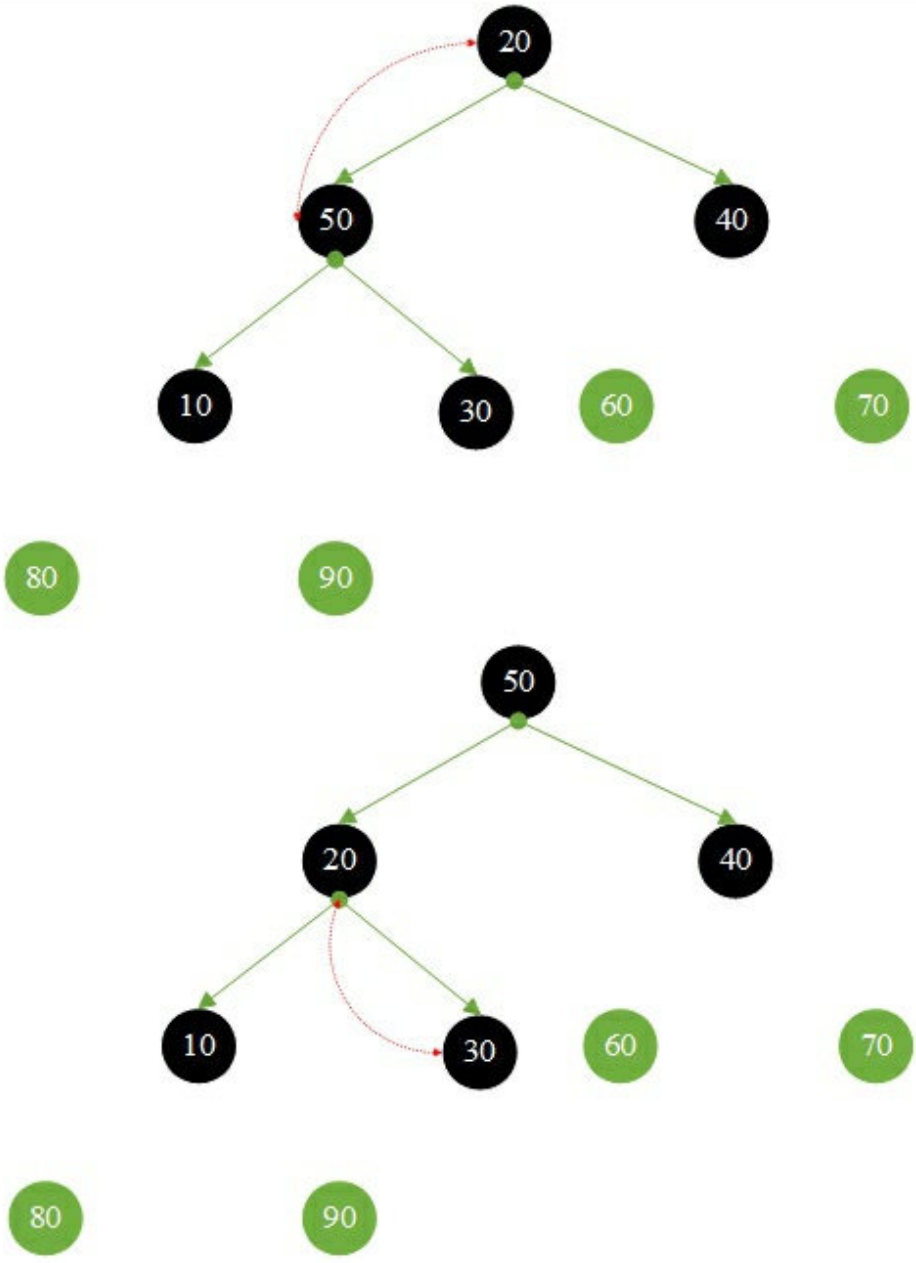


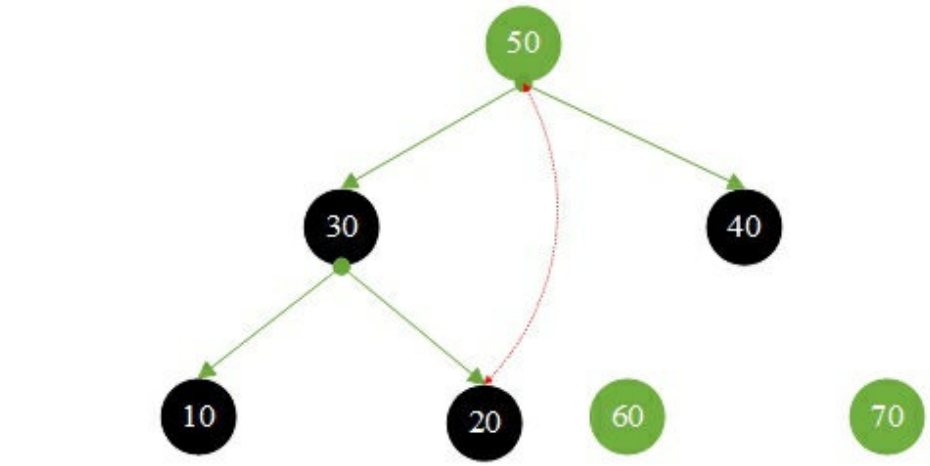


root = 60 and tail = 20 are exchanged



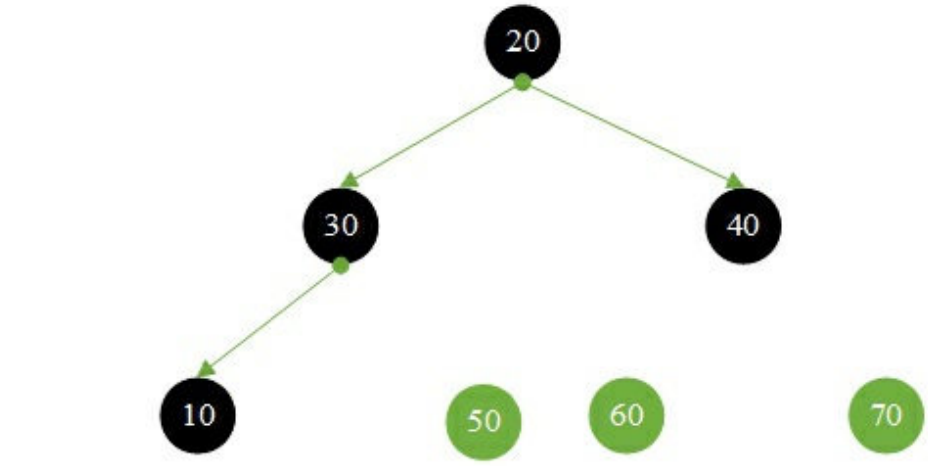
adjust the heap





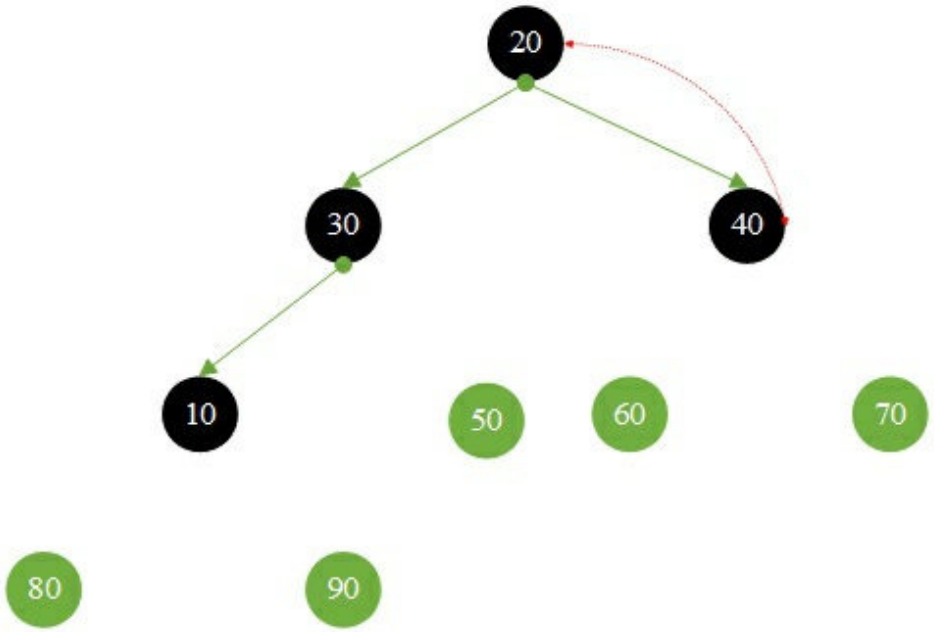
80 90

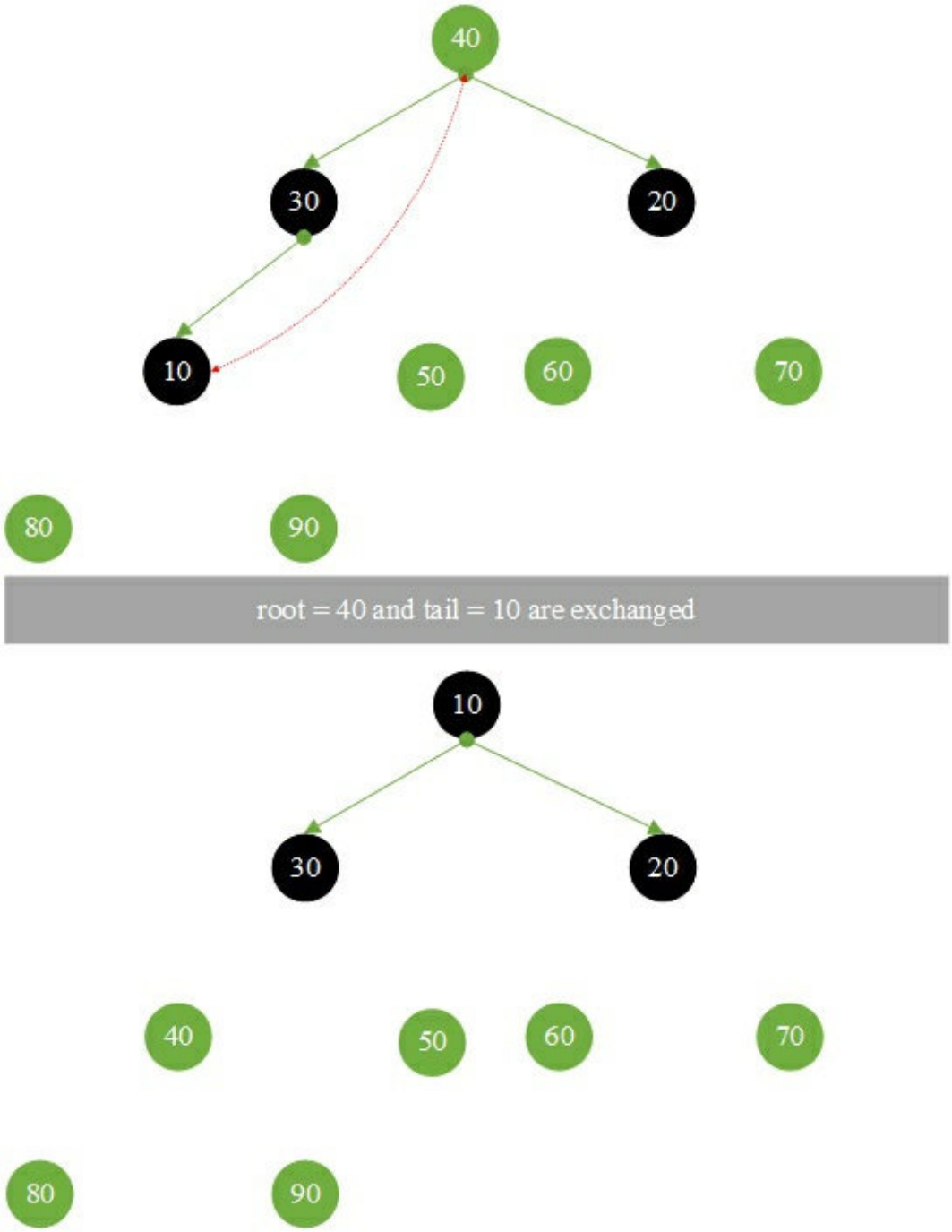
root = 50 and tail = 20 are exchanged

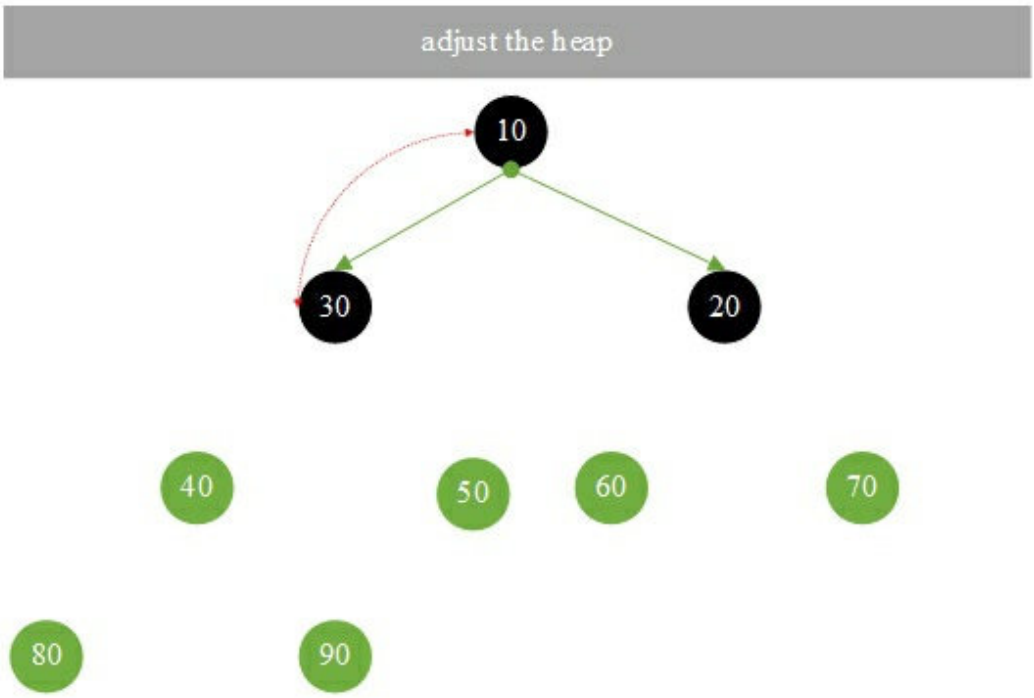


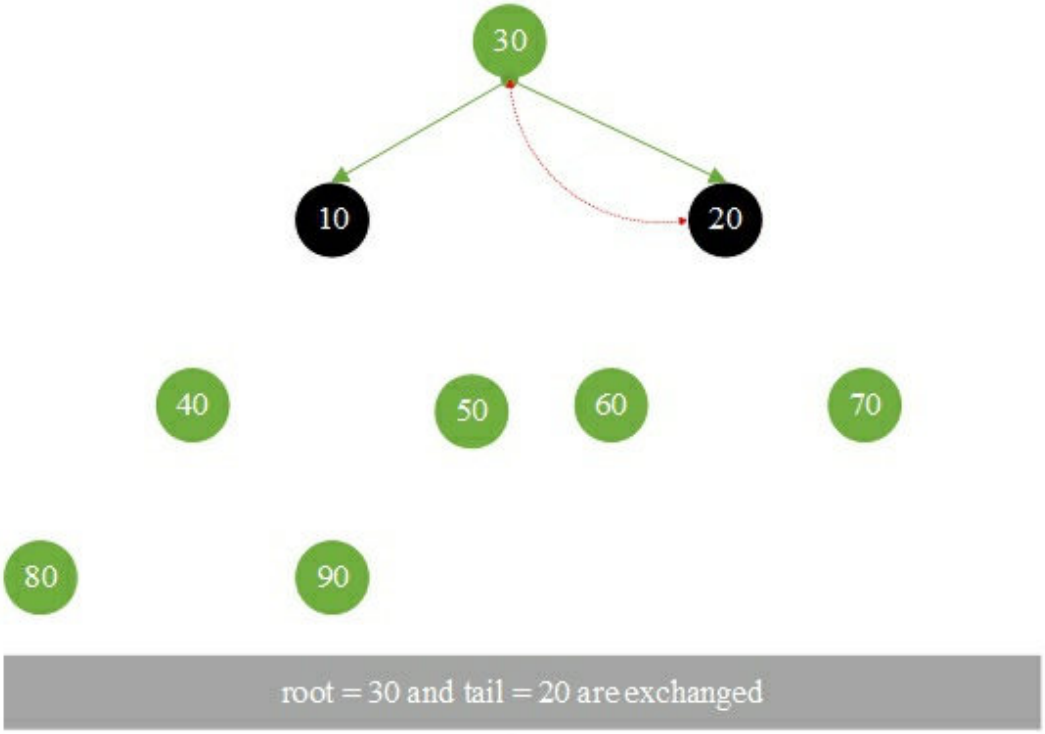
80 90

adjust the heap

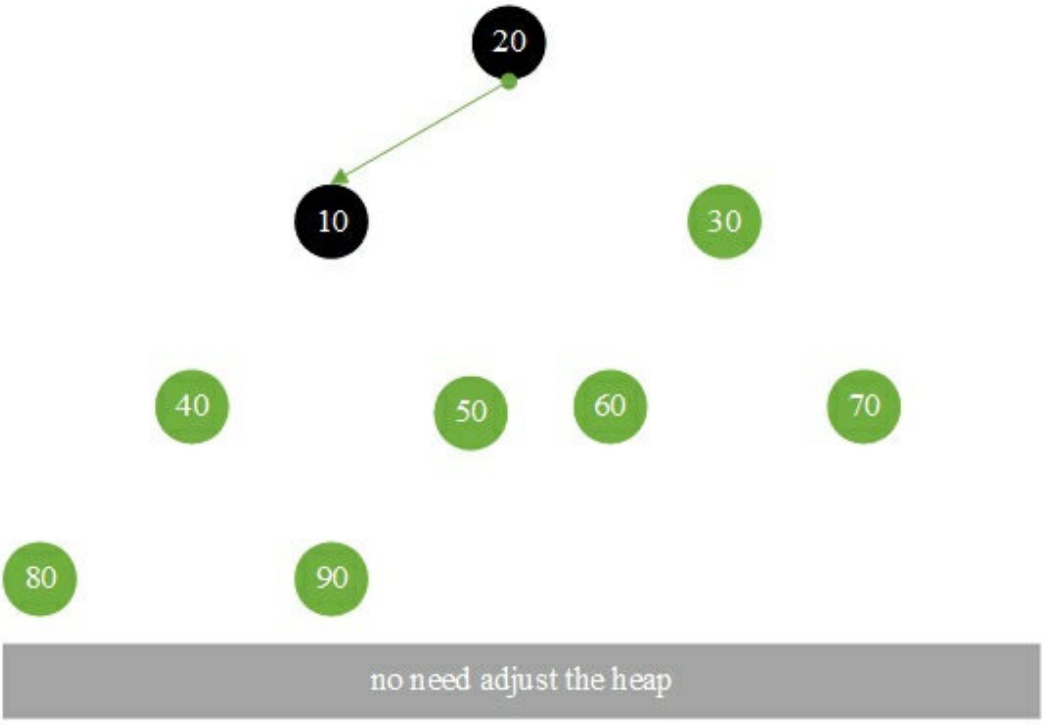



40
30
20
10
50
60
70
80
90
root = 40 and tail = 10 are exchanged
10
30
20
40
50
60
70
80
90

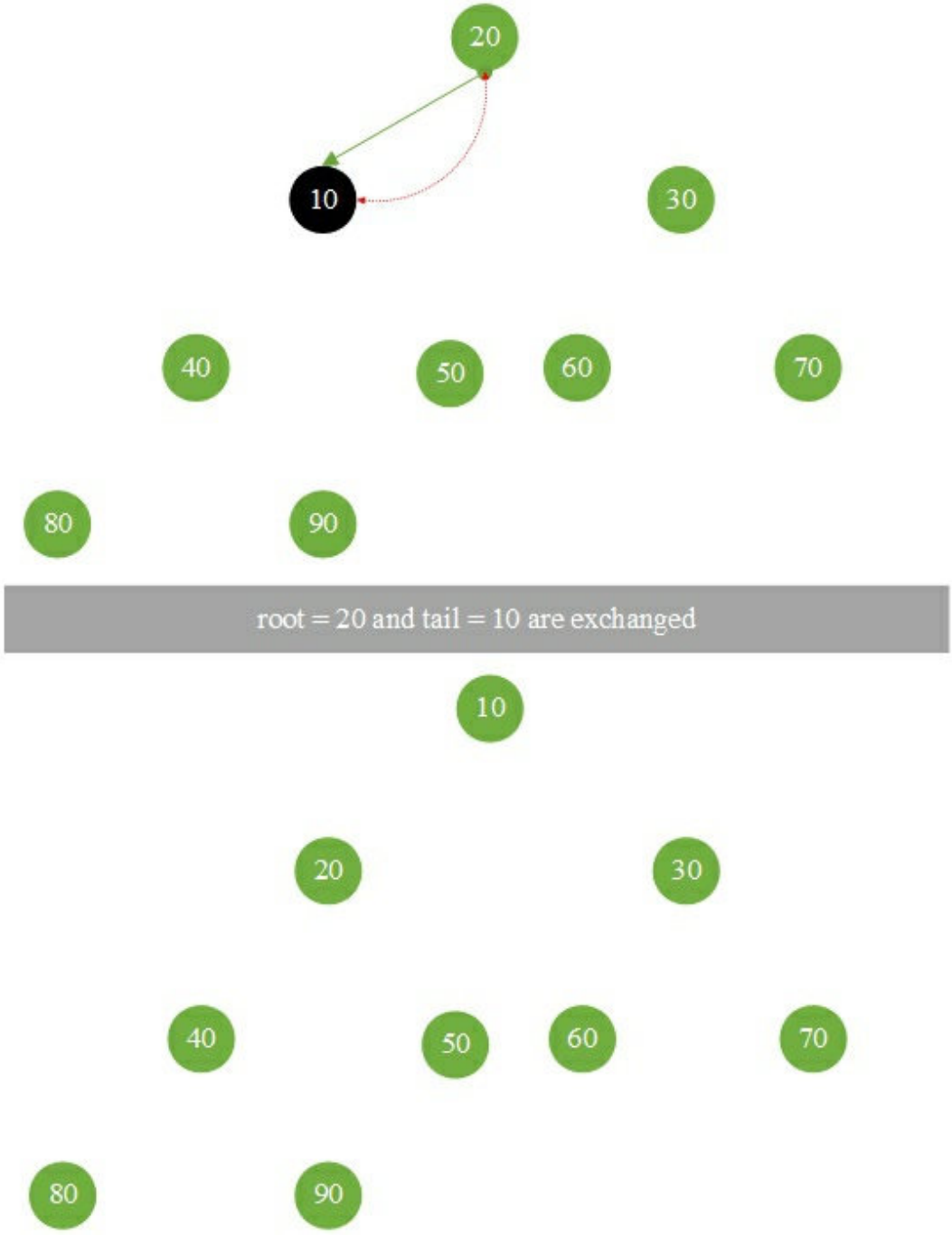

adjust the heap
10
30
20
40
50
60
70
80
90



root = 30 and tail = 20 are exchanged



no need adjust the heap


20
10
30
40
50
60
70
80
90
root = 20 and tail = 10 are exchanged
10
20
30
40
50
60
70
80
90

Heap sort result

HeapSort.go

```go
package main

import "fmt"

//Adjustment heap
func adjustHeap(array []int, currentIndex int, maxLength int) {
    var noLeafValue = array[currentIndex] // Current non-leaf node

    //2 * currentIndex + 1  Current left subtree subscript
    for j := 2*currentIndex + 1; j <= maxLength; j = currentIndex*2 + 1 {
        if j < maxLength && array[j] < array[j+1] {
            j++ // j Large subscript
        }

        if noLeafValue >= array[j] {
            break
        }

        array[currentIndex] = array[j] // Move up to the parent node
        currentIndex = j
    }

    array[currentIndex] = noLeafValue // To put in the position
}

//Initialize the heap
func createHeap(array []int, length int) {
    // Build a heap, (length - 1) / 2 scan half of the nodes with child nodes
    for i := (length - 1) / 2; i >= 0; i-- {
        adjustHeap(array, i, length-1)
    }
}
```

```
func heapSort(array []int, length int) {
    for i := length - 1; i > 0; i-- {
        var temp = array[0]
        array[0] = array[i]
        array[i] = temp
        adjustHeap(array, 0, i-1)
    }
}

func main() {
    var scores = []int{10, 90, 20, 80, 30, 70, 40, 60, 50}
    var length = len(scores)

    fmt.Printf("Before building a heap : \n")
    for i := 0; i < length; i++ {
        fmt.Printf("%d, ", scores[i])
    }
    fmt.Printf("\n\n")

    fmt.Printf("After building a heap : \n")
    createHeap(scores, length)
    for i := 0; i < length; i++ {
        fmt.Printf("%d, ", scores[i])
    }
    fmt.Printf("\n\n")

    fmt.Printf("After heap sorting : \n")
    heapSort(scores, length)
    for i := 0; i < length; i++ {
        fmt.Printf("%d, ", scores[i])
    }
}
```

Result:

Before building a heap :
10, 90, 20, 80, 30, 70, 40, 60, 50,

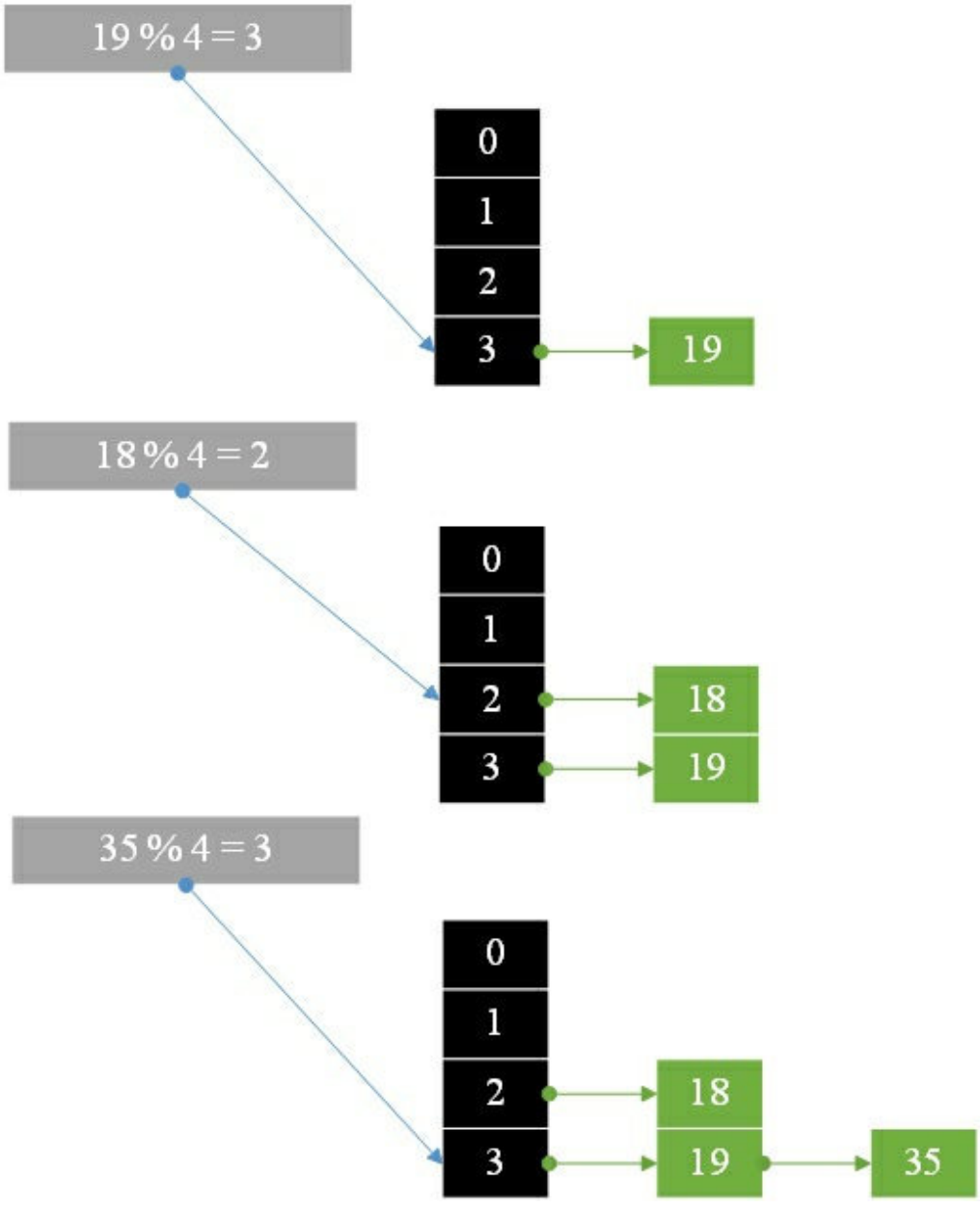
After building a heap :
90, 80, 70, 60, 30, 20, 40, 10, 50,

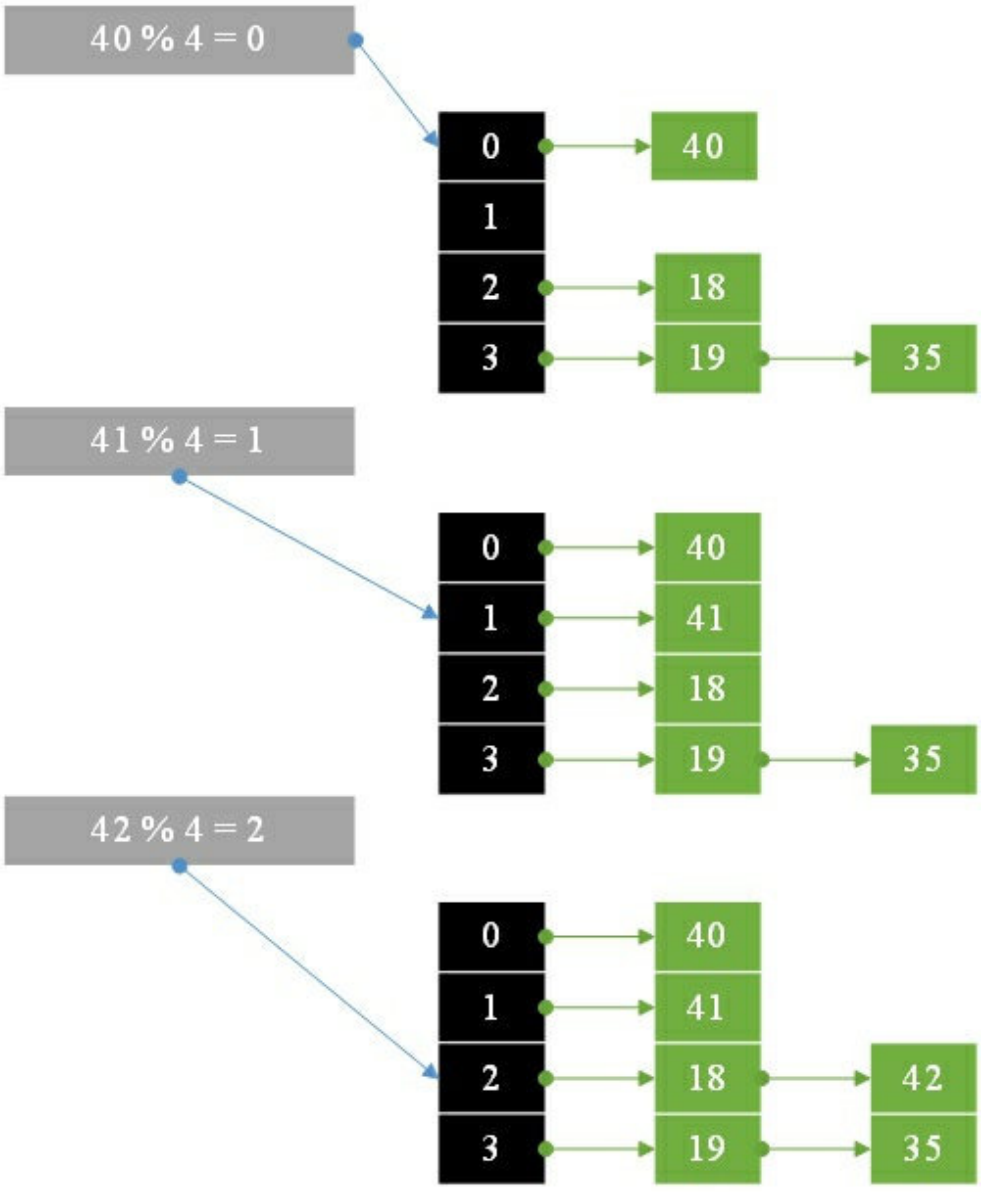
After heap sorting :
10, 20, 30, 40, 50, 60, 70, 80, 90,

Hash Table

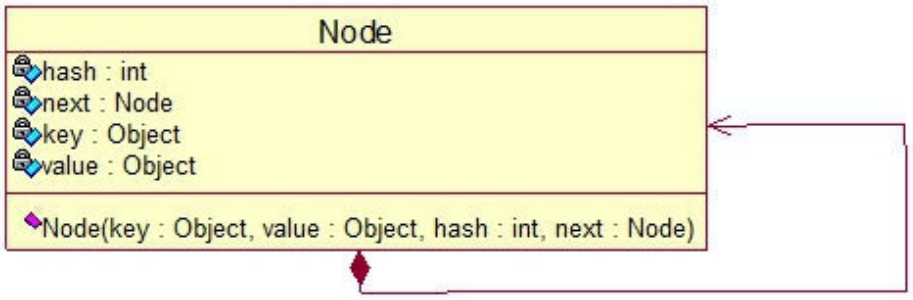
Hash Table:
Access by mapping key => values in the table.

1. Map {19, 18, 35,40,41,42} to the HashTable mapping rule key % 4




40 % 4 = 0
0
1
2
3
40
18
19
35
41 % 4 = 1
0
1
2
3
40
41
18
19
35
42 % 4 = 2
0
1
2
3
40
41
18
19
42
35

2. Implement a Hashtable




```
type Node struct {
    key   string
    value string
    hash  int
    next  *Node
}
```

Hashtable.go

```
package main

import (
    "fmt"
    "math"
    "strings"
)

type Node struct {
    key   string
    value string
    hash  int
    next  *Node
}

const CAPACITY = 16

var table = make([]*Node, CAPACITY)
var size int

func isEmpty() bool {
    if size == 0 {
        return true
    } else {
        return false
    }
}

func hashCode(key string) int {
    var num = 0
    var length = len(key)
    for i := 0; i < length; i++ {
        num += int(key[i])
    }
    //hash strategy is to take the square in the middle
    var avg = num * int((math.Pow(5.0, 0.5) - 1)) / 2
    var numeric = avg - int(math.Floor(float64(avg)))
```

```
    return int(math.Floor(float64(numeric * CAPACITY)))
}


func put(key string, value string) {
    var hash = hashCode(key)
    var newNode *Node = new(Node)
    newNode.key = key
    newNode.value = value
    newNode.hash = hash
    newNode.next = nil

    var node = table[hash]
    for {
        if node == nil {
          break
        }
        if strings.Compare(node.key, key) == 0 {
          node.value = value
          return
        }
        node = node.next
    }
    newNode.next = table[hash]
    table[hash] = newNode
    size++
}

func get(key string) string {
    if key == "" {
        return ""
    }
    var hash = hashCode(key)
    var node = table[hash]
    for {
        if node == nil {
          break
        }
```

```
        if strings.Compare(node.key, key) == 0 {
          return node.value
        }
        node = node.next
    }
    return ""
}


func main() {
    put("david", "Good Boy Keep Going")
    put("grace", "Cute Girl Keep Going")

    fmt.Printf("david => %s \n", get("david"))
    fmt.Printf("grace => %s \n", get("grace"))
}
```

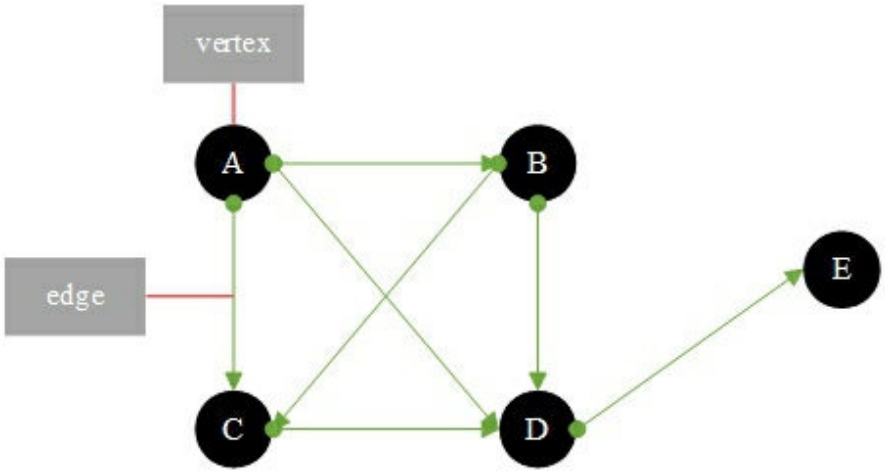
Result:

```
david => Good Boy Keep Going
grace => Cute Girl Keep Going
```

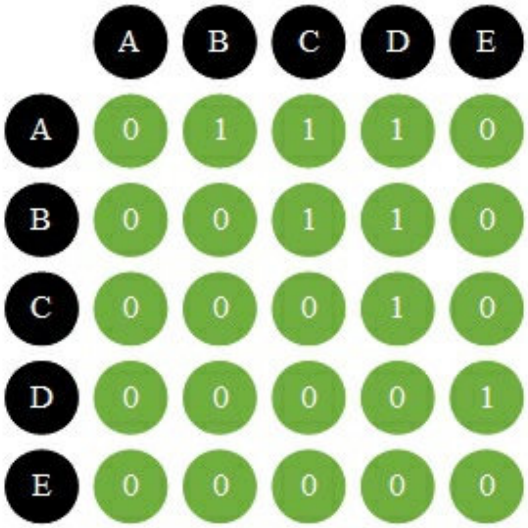
Directed Graph and Depth-First Search

Directed Graph:
The data structure is represented by an adjacency matrix (that is, a two-dimensional array) and an adjacency list. Each node is called a vertex, and two adjacent nodes are called edges.

Directed Graph has direction : A -> B and B -> A are different

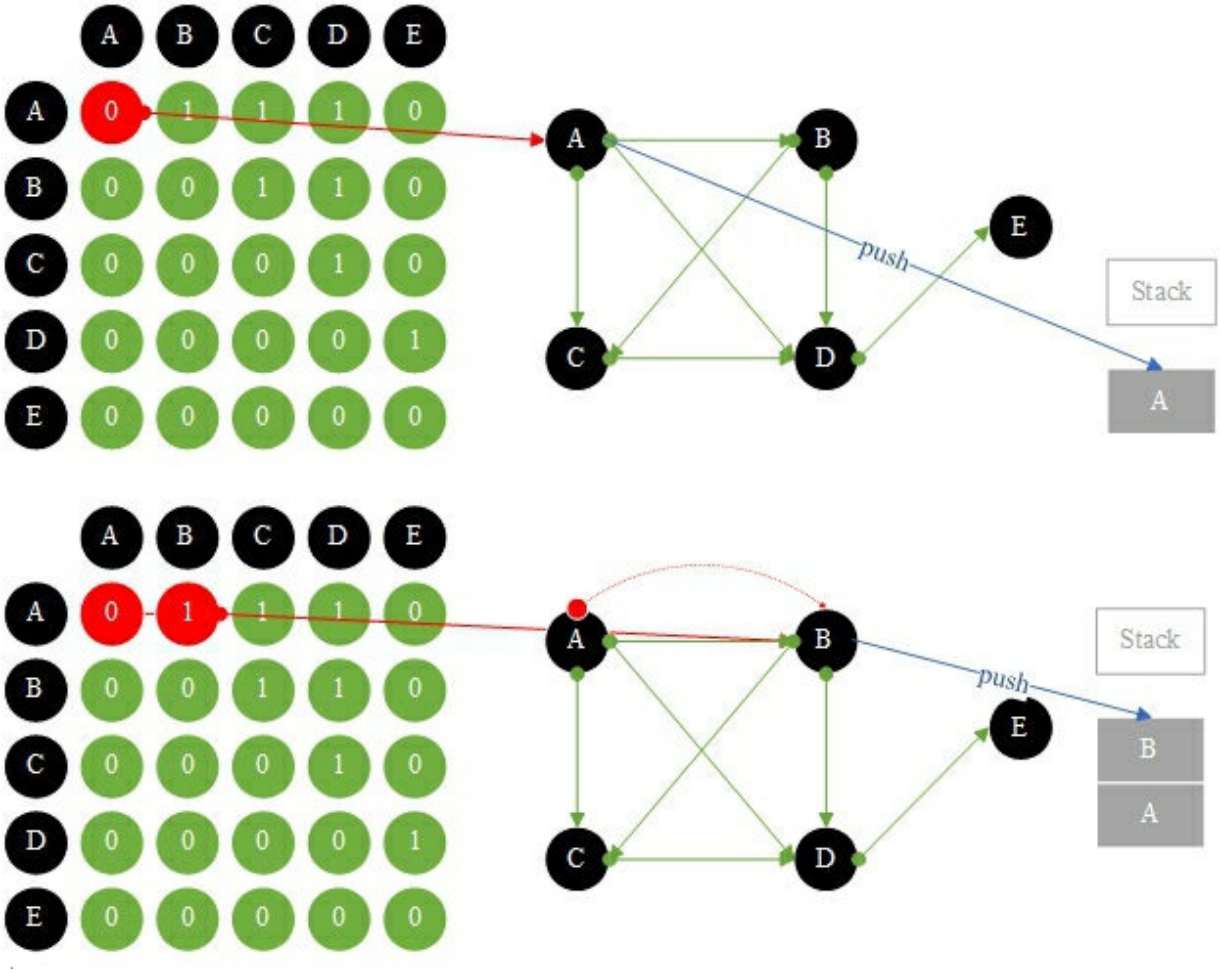


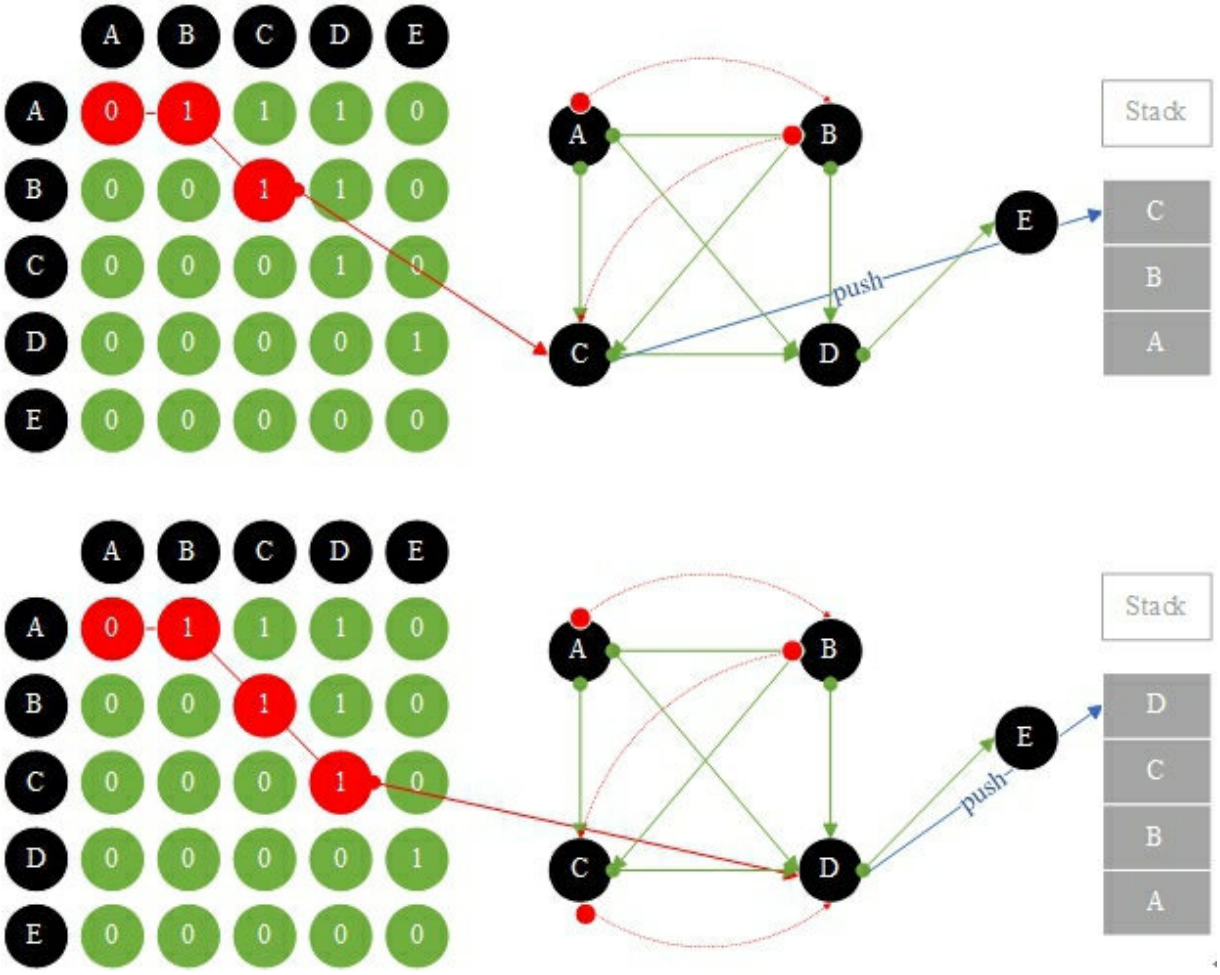
1. The adjacency matrix is described above:
The total number of vertices is a two-dimensional array size, if have value of the edge is 1, otherwise no value of the edge is 0.

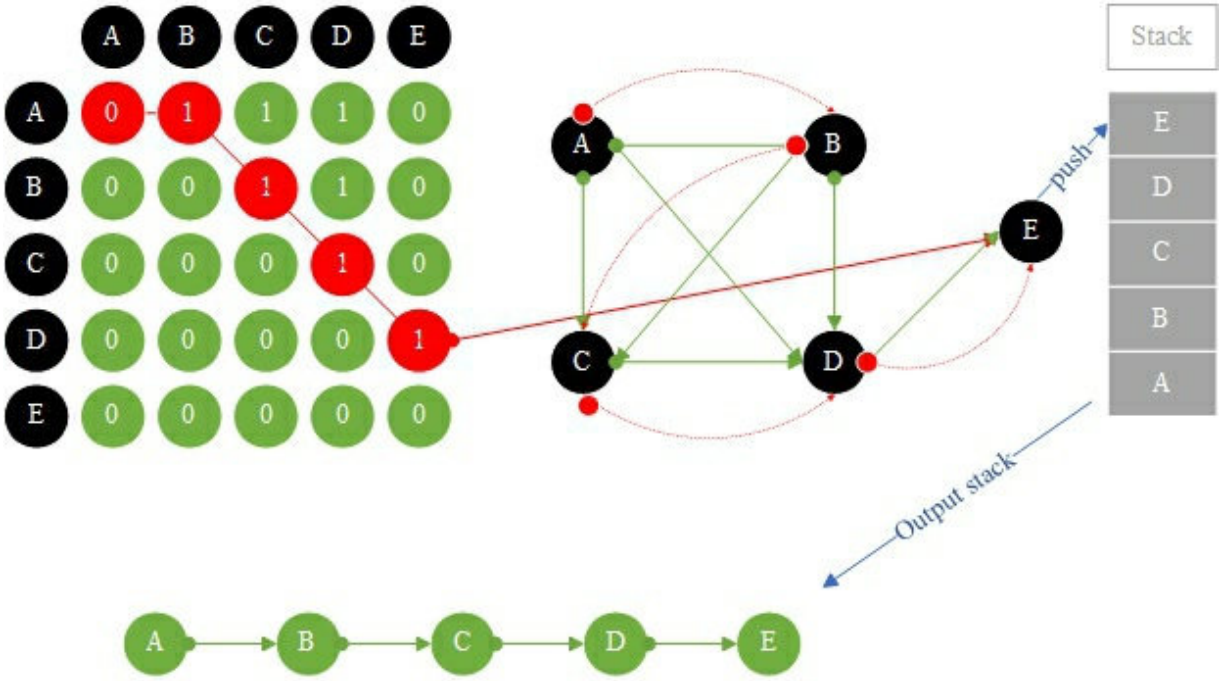


2. Depth-First Search:

Look for the neighboring edge node B from A and then find the neighboring node C from B and so on until all nodes are found A -> B -> C -> D -> E.




A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
C
B
A
A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
push
Stack
D
C
B
A


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
Stack
E
D
C
B
A
push
Output stack
A → B → C → D → E

Graph.go

```
package main
import "fmt"
const MAX_VERTEX_SIZE = 5
const STACKSIZE = 1000
type Vertex struct {
    data    string
    visited bool // Have you visited
}

var top = -1 // Stack saves current vertices
var stacks = make([]int, STACKSIZE)

func push(element int) {
    top++
    stacks[top] = element
}

func pop() int {
    if top == -1 {
        return -1
    }
    var data = stacks[top]
    top--
    return data
}

func peek() int {
    if top == -1 {
        return -1
    }
    var data = stacks[top]
    return data
}

func isEmpty() bool {
    if top <= -1 {
        return true
```

```
    }
    return false
}
////// stack end ////////////////////////


var size = 0 // Current vertex size
var vertexs = make([]Vertex, MAX_VERTEX_SIZE)
var adjacencyMatrix [MAX_VERTEX_SIZE][MAX_VERTEX_SIZE]int

func addVertex(data string) {
    var vertex Vertex
    vertex.data = data
    vertex.visited = false
    vertexs[size] = vertex
    size++
}

func addEdge(from int, to int) {// Add adjacent edges
    adjacencyMatrix[from][to] = 1 // A -> B != B -> A
}

func clear() {
    for i := 0; i < size; i++ {
        vertexs[i].visited = false
    }
}

func depthFirstSearch() {
    vertexs[0].visited = true // Start searching from the first vertex
    fmt.Printf("%s", vertexs[0].data)
    push(0)
    for {
        if isEmpty() {
            break
        }
        var row = peek()
        // Get adjacent vertex positions that have not been visited
        var col = findAdjacencyUnVisitedVertex(row)
```

```
        if col == -1 {
            pop()
        } else {
            vertexs[col].visited = true
            fmt.Printf(" -> %s", vertexs[col].data)
            push(col)
        }
    }
    clear()
}

// Get adjacent vertex positions that have not been visited
func findAdjacencyUnVisitedVertex(row int) int {
    for col := 0; col < size; col++ {
        if adjacencyMatrix[row][col] == 1 && !vertexs[col].visited {
            return col
        }
    }
    return -1
}

func printGraph() {
    fmt.Printf("Two-dimensional array traversal vertex edge and adjacent
array :  \n   ")
    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
    }
    fmt.Printf("\n")

    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
        for j := 0; j < MAX_VERTEX_SIZE; j++ {
            fmt.Printf("%d  ", adjacencyMatrix[i][j])
        }
        fmt.Printf("\n")
    }
}
```

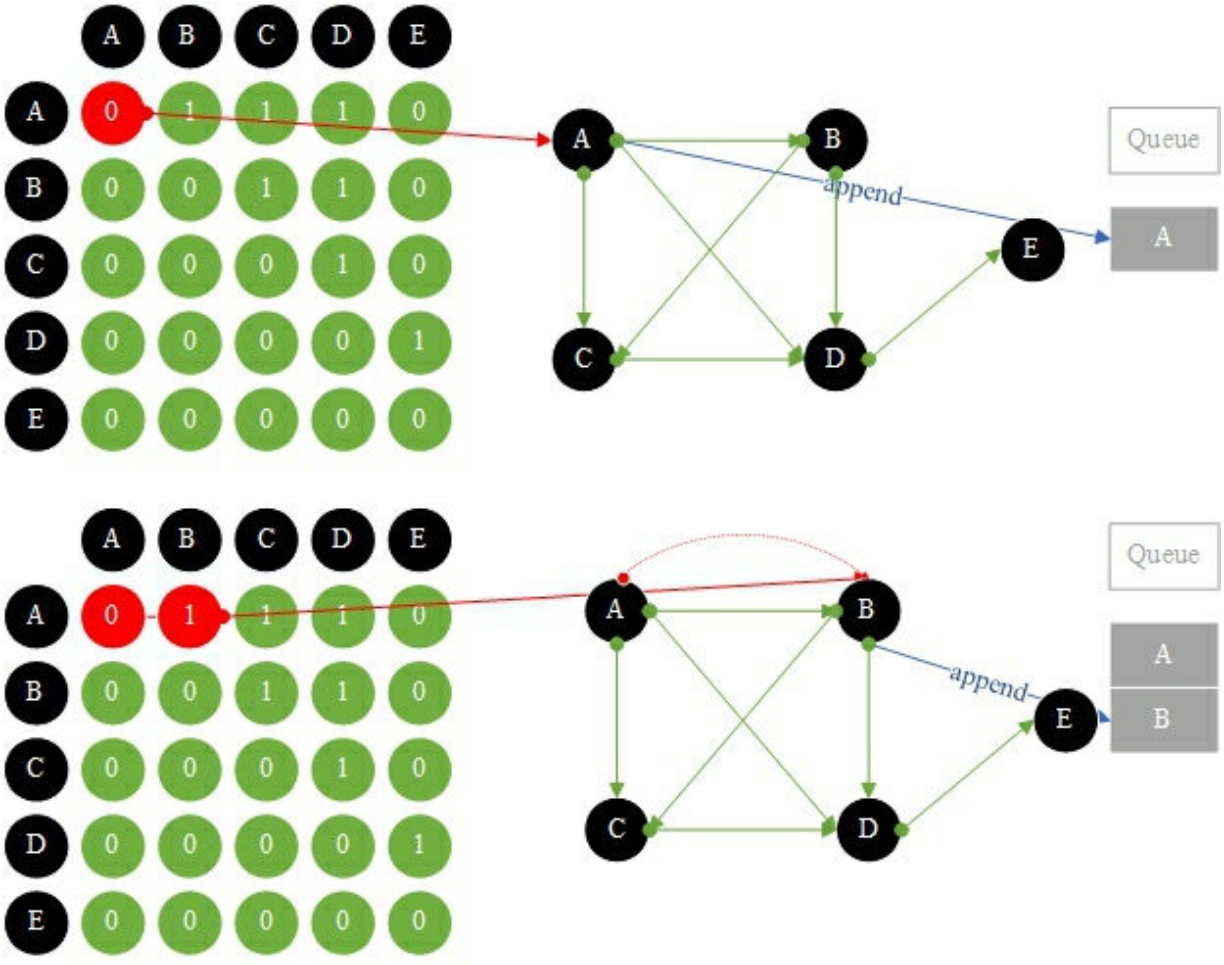
```
func main() {
    addVertex("A")
    addVertex("B")
    addVertex("C")
    addVertex("D")
    addVertex("E")

    addEdge(0, 1)
    addEdge(0, 2)
    addEdge(0, 3)
    addEdge(1, 2)
    addEdge(1, 3)
    addEdge(2, 3)
    addEdge(3, 4)

    // Two-dimensional array traversal output vertex edge and adjacent array
    printGraph()

    fmt.Printf("\nDepth-first search traversal output : \n")
    depthFirstSearch()
}
```

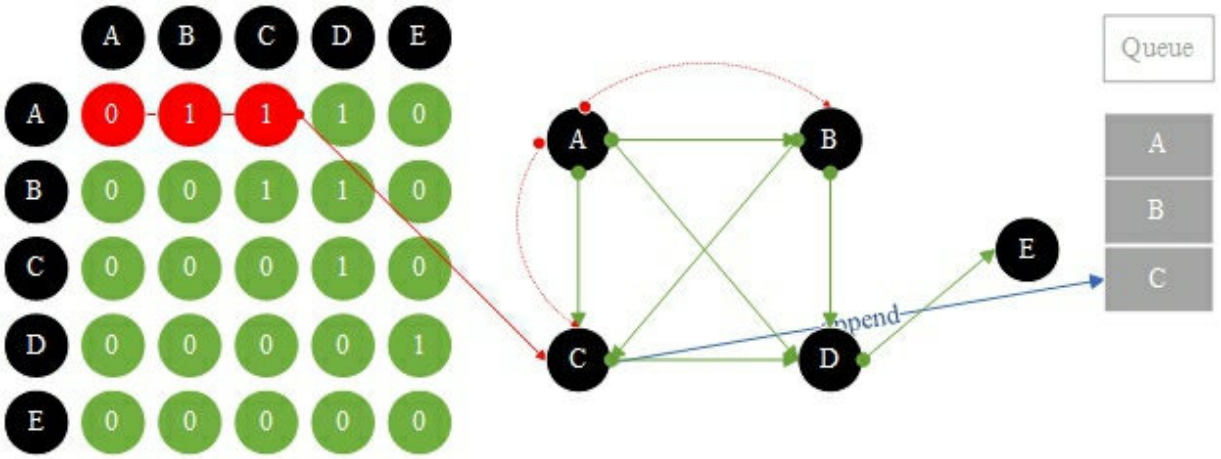
Result:

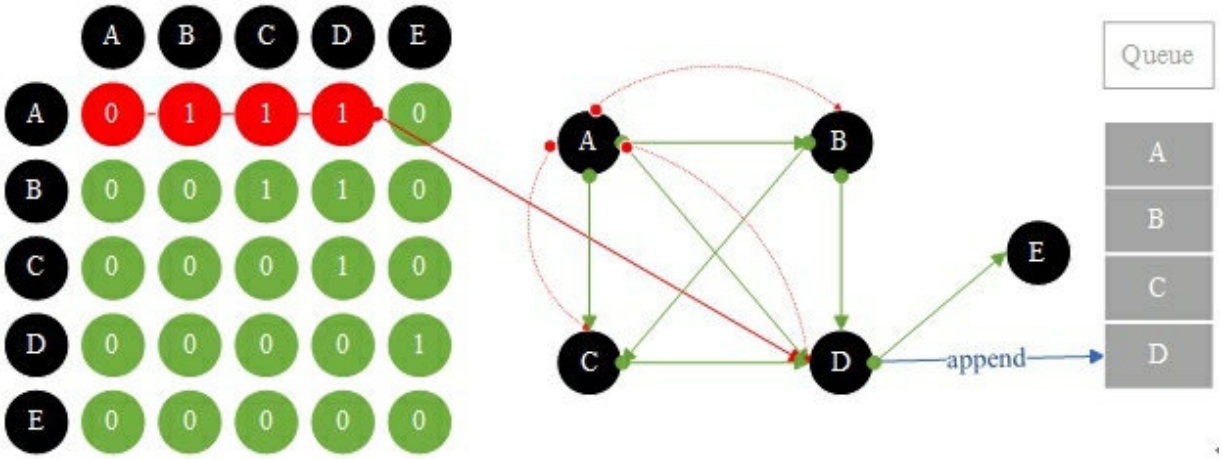
```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0

Depth-first search traversal output :
A -> B -> C -> D -> E
```

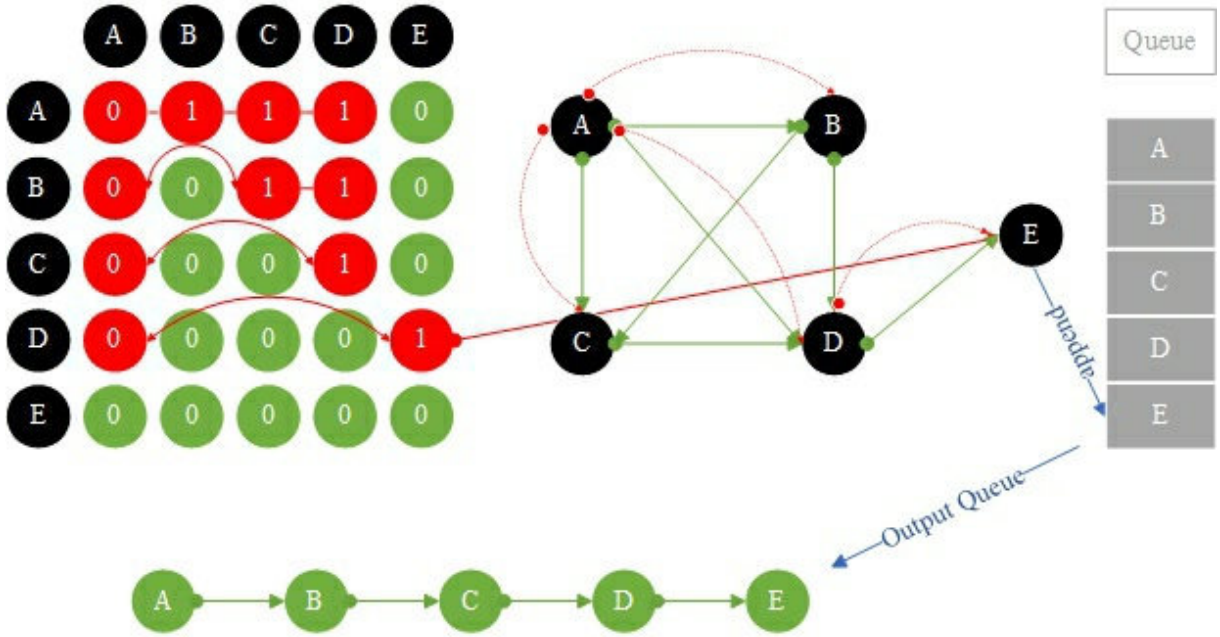
Directed Graph and Breadth-First Search

Breadth-First Search:
Find all neighboring edge nodes B, C, D from A and then find all neighboring nodes A, C, D from B and so on until all nodes are found A -> B -> C -> D -> E.




A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
Queue
A
B
C
D
E
append


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
Queue
A
B
C
D
append


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
Queue
A
B
C
D
E
append
Output Queue
A
B
C
D
E

Graph.go

```
package main

import "fmt"

const MAX_VERTEX_SIZE = 5

// Queue saves current vertices
const QUEUESIZE = 40

type Queue struct {
    queue [QUEUESIZE]int
    head  int
    tail  int
}

var q *Queue = nil

func initQueue() {
    q = new(Queue)
    q.head = 0
    q.tail = 0
}

func isQueueEmpty() bool {
    if q.head == q.tail {
        return true
    } else {
        return false
    }
}

func enQueue(data int) bool {
    if q.tail == QUEUESIZE {
        fmt.Printf("The queue was full and could not join.\n")
        return false
    }
    q.queue[q.tail] = data
```

```
    q.tail++
    return true
}


func deleteQueue() int {
    if q.head == q.tail {
        fmt.Printf("The queue was empty and could not join.\n")
    }
    var data = q.queue[q.head]
    q.head++
    return data
}

////// queue end ////////////////////////

type Vertex struct {
    data    string
    visited bool // Have you visited
}

var size = 0 // Current vertex size
var vertexs [MAX_VERTEX_SIZE]Vertex
var adjacencyMatrix [MAX_VERTEX_SIZE][MAX_VERTEX_SIZE]int

func addVertex(data string) {
    var vertex Vertex
    vertex.data = data
    vertex.visited = false
    vertexs[size] = vertex
    size++
}

// Add adjacent edges
func addEdge(from int, to int) {
    // A -> B != B -> A
    adjacencyMatrix[from][to] = 1
}

// Clear reset
```

```
func clear() {
    for i := 0; i < size; i++ {
        vertexs[i].visited = false
    }
}



func breadthFirstSearch() {
    // Start searching from the first vertex
    vertexs[0].visited = true
    fmt.Printf("%s", vertexs[0].data)
    enQueue(0)

    var col int
    for {
        if isQueueEmpty() {
            break
        }
        var row = deleteQueue()
        // Get adjacent vertex positions that have not been visited
        col = findAdjacencyUnVisitedVertex(row)
        //Loop through all vertices connected to the current vertex
        for {
            if col == -1 {
                break
            }
            vertexs[col].visited = true
            fmt.Printf(" -> %s", vertexs[col].data)
            enQueue(col)
            col = findAdjacencyUnVisitedVertex(row)
        }
    }
    clear()
}

// Get adjacent vertex positions that have not been visited
func findAdjacencyUnVisitedVertex(row int) int {
```

```
    for col := 0; col < size; col++ {
        if adjacencyMatrix[row][col] == 1 && !vertexs[col].visited {
            return col
        }
    }
    return -1
}




func printGraph() {
    fmt.Printf("Two-dimensional array traversal vertex edge and adjacent
array :  \n   ")
    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
    }
    fmt.Printf("\n")

    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
        for j := 0; j < MAX_VERTEX_SIZE; j++ {
            fmt.Printf("%d  ", adjacencyMatrix[i][j])
        }
        fmt.Printf("\n")
    }
}

func main() {
    initQueue()

    addVertex("A")
    addVertex("B")
    addVertex("C")
    addVertex("D")
    addVertex("E")

    addEdge(0, 1)
```

```
    addEdge(0, 2)
    addEdge(0, 3)
    addEdge(1, 2)
    addEdge(1, 3)
    addEdge(2, 3)
    addEdge(3, 4)

    // Two-dimensional array traversal output vertex edge and adjacent
array
    printGraph()

    fmt.Printf("\nBreadth-first search traversal output : \n")
    breadthFirstSearch()
}
```

Result:

```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0
```

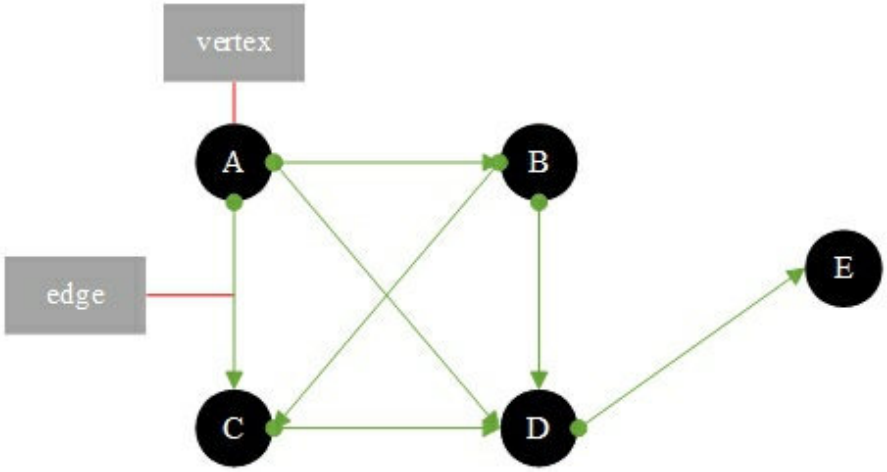
Breadth-first search traversal output :
A -> B -> C -> D -> E

Directed Graph Topological Sorting

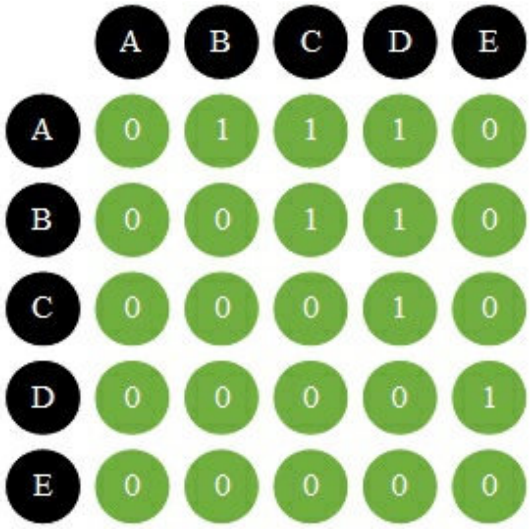
Directed Graph Topological Sorting:
Sort the vertices in the directed graph with order of direction

.

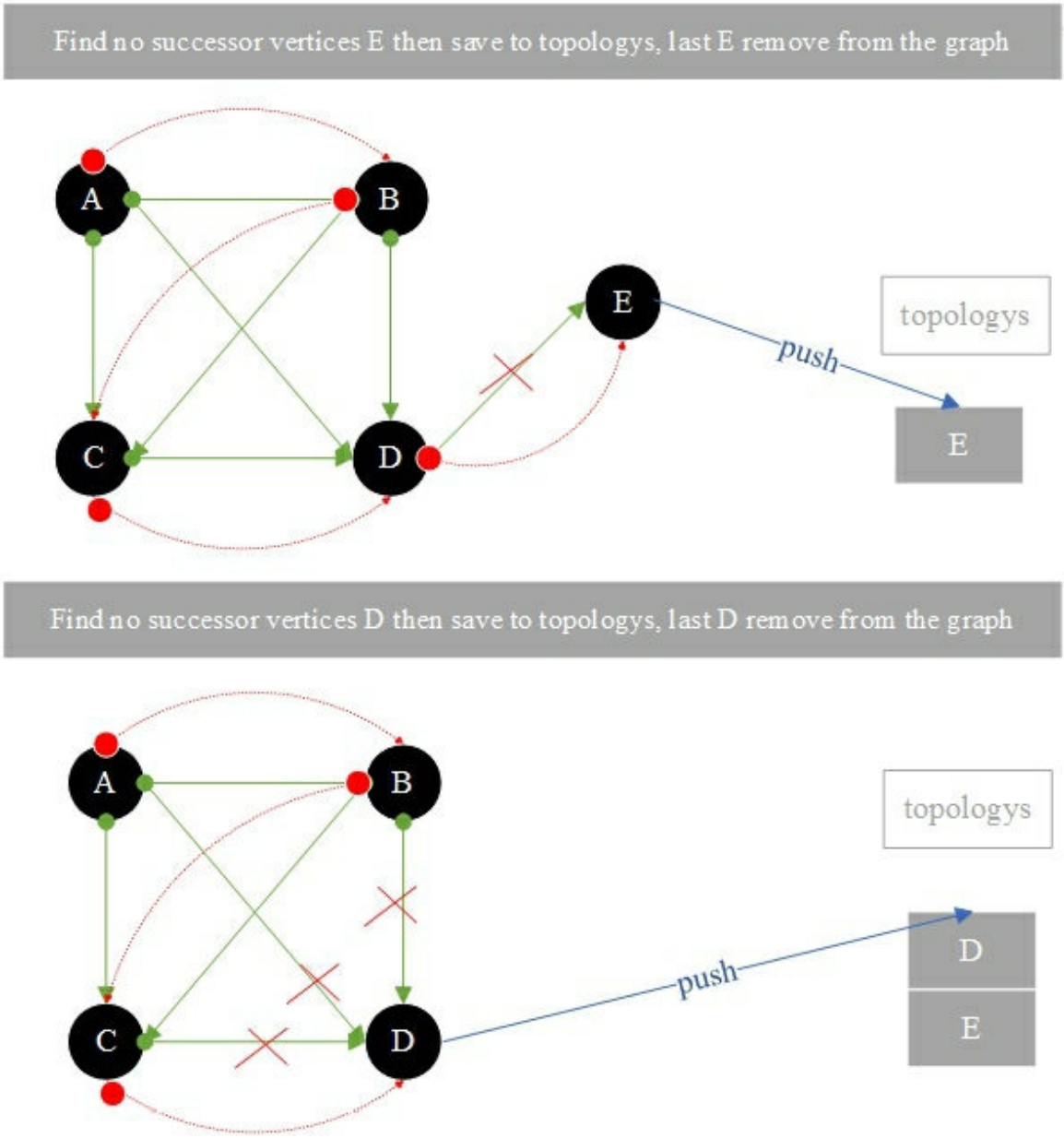
Directed Graph has direction : A -> B and B -> A are different



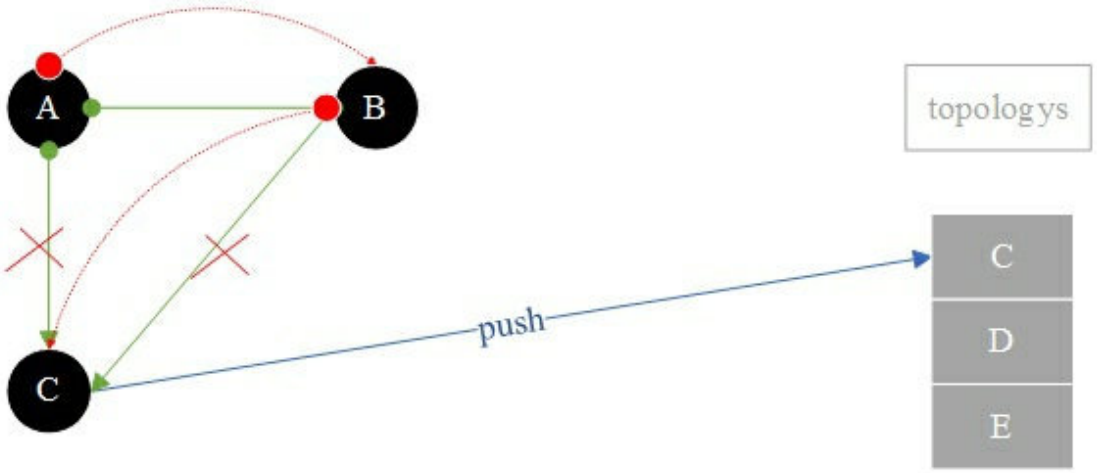
1. The adjacency matrix is described above:
The total number of vertices is a two-dimensional array size, if have value of the edge is 1, otherwise no value of the edge is 0.



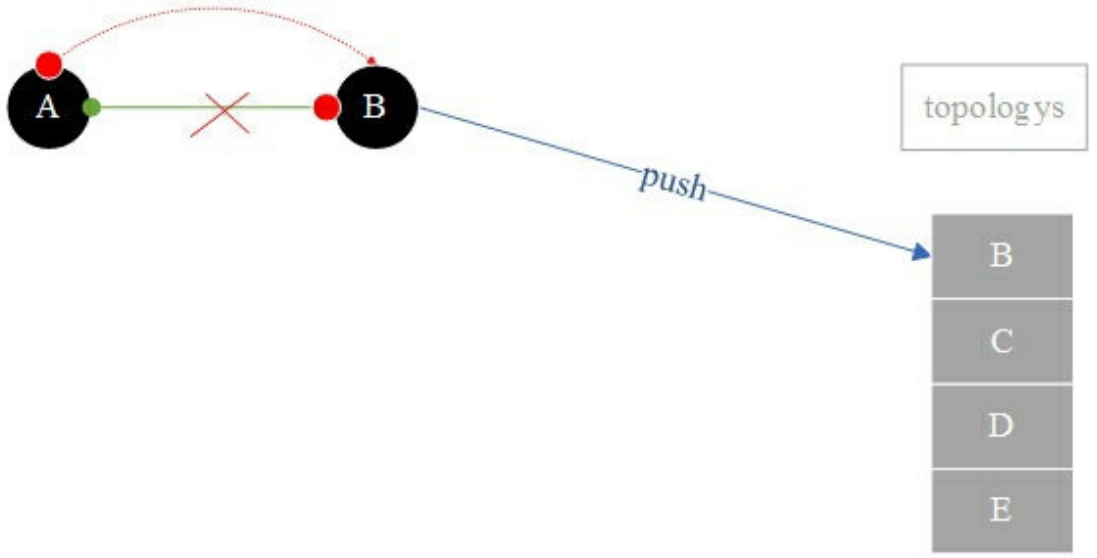
Topological sorting from vertex A : A -> B -> C -> D -> E



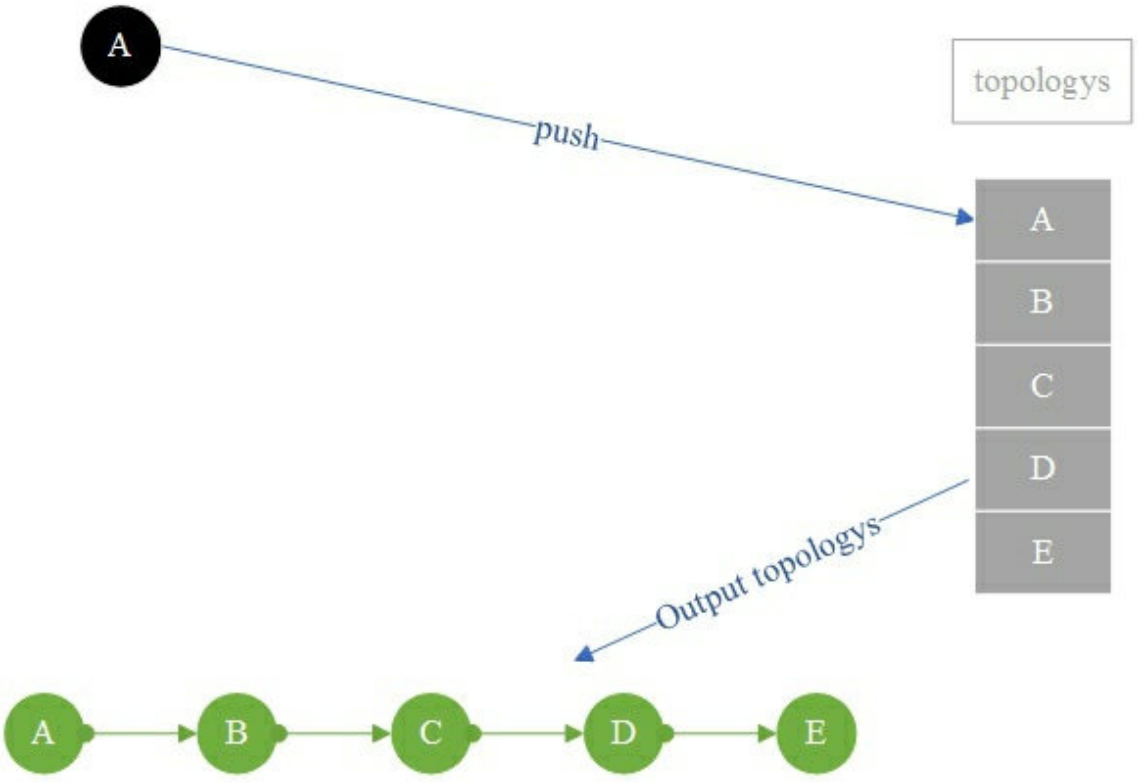
Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



Topology.go

```
package main
import "fmt"
const MAX_VERTEX_SIZE = 5
const STACKSIZE = 1000

type Vertex struct {
    data    string
    visited bool // Have you visited
}

// Stack saves current vertices
var top = -1
var stacks [STACKSIZE]int

func push(element int) {
    top++
    stacks[top] = element
}

func pop() int {
    if top == -1 {
        return -1
    }
    var data = stacks[top]
    top--
    return data
}

func peek() int {
    if top == -1 {
        return -1
    }
    var data = stacks[top]
    return data
}

func isEmpty() bool {
```

```
    if top <= -1 {
        return true
    }
    return false
}

var size = 0 // Current vertex size
var vertexs [MAX_VERTEX_SIZE]Vertex
// An array of topological sort results, recording the sorted sequence
number of each node.
var topologys [MAX_VERTEX_SIZE]Vertex
var adjacencyMatrix [MAX_VERTEX_SIZE][MAX_VERTEX_SIZE]int

func addVertex(data string) {
    var vertex Vertex
    vertex.data = data
    vertex.visited = false
    vertexs[size] = vertex
    size++
}

// Add adjacent edges
func addEdge(from int, to int) {
    // A -> B = B -> A
    adjacencyMatrix[from][to] = 1
}

func removeVertex(vertex int) {
    if vertex != size-1 {
        //If the vertex is the last element, the end
        for i := vertex; i < size-1; i++ { // The vertices are removed from the
vertex array
            vertexs[i] = vertexs[i+1]
        }

        for row := vertex; row < size-1; row++ {
            // move up a row
            for col := 0; col < size-1; col++ {
                adjacencyMatrix[row][col] = adjacencyMatrix[row+1][col]
```

```
            }
        }

        for col := vertex; col < size-1; col++ { // move left a row
            for row := 0; row < size-1; row++ {
                adjacencyMatrix[row][col] = adjacencyMatrix[row][col+1]
            }
        }
    }
    size-- // Decrease the number of vertices
}


func topologySort() {
    for {
        if size <= 0 {
            break
        }
        var noSuccessorVertex = getNoSuccessorVertex() // Get a no
successor node
        if noSuccessorVertex == -1 {
            fmt.Printf("There is ring in Graph \n")
            return
        }
        topologys[size-1] = vertexs[noSuccessorVertex] // Copy the deleted
node to the sorted array
        removeVertex(noSuccessorVertex)             // Delete no successor
node
    }
}

func getNoSuccessorVertex() int {
    var existSuccessor = false
    for row := 0; row < size; row++ {
        // For each vertex
        existSuccessor = false
        //If the node has a fixed row, each column has a 1, indicating that
the node has a successor, terminating the loop
```

```
        for col := 0; col < size; col++ {
            if adjacencyMatrix[row][col] == 1 {
                existSuccessor = true
                break
            }
        }

        if !existSuccessor {
            // If the node has no successor, return its subscript
            return row
        }
    }
    return -1
}




func printGraph() {
    fmt.Printf("Two-dimensional array traversal vertex edge and adjacent
array :  \n   ")
    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
    }
    fmt.Printf("\n")

    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s  ", vertexs[i].data)
        for j := 0; j < MAX_VERTEX_SIZE; j++ {
            fmt.Printf("%d  ", adjacencyMatrix[i][j])
        }
        fmt.Printf("\n")
    }
}

func main() {
    addVertex("A")
```

```
    addVertex("B")
    addVertex("C")
    addVertex("D")
    addVertex("E")

    addEdge(0, 1)
    addEdge(0, 2)
    addEdge(0, 3)
    addEdge(1, 2)
    addEdge(1, 3)
    addEdge(2, 3)
    addEdge(3, 4)

    // Two-dimensional array traversal output vertex edge and adjacent
array
    printGraph()

    fmt.Printf("\nDepth-First Search traversal output : \n")
    fmt.Printf("Directed Graph Topological Sorting: \n")
    topologySort()
    for i := 0; i < MAX_VERTEX_SIZE; i++ {
        fmt.Printf("%s -> ", topologys[i].data)
    }
}
```

Result:

Two-dimensional array traversal output vertex edge and adjacent array :

```
   A  B  C  D  E
A  0  1  1  1  0
B  0  0  1  1  0
C  0  0  0  1  0
D  0  0  0  0  1
E  0  0  0  0  0
```

Depth-First Search traversal output :
Directed Graph Topological Sorting:
A -> B -> C -> D -> E ->

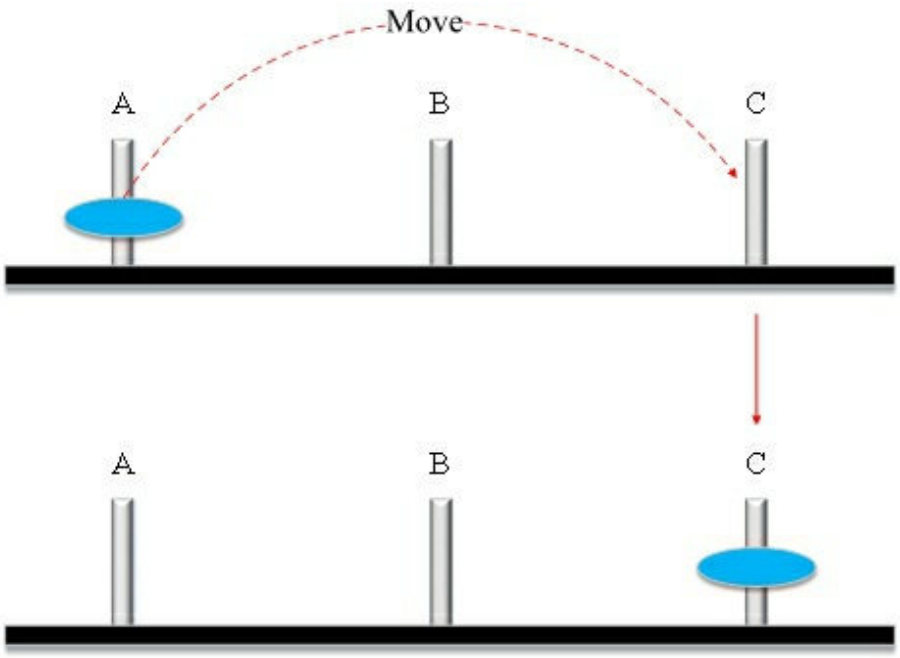
Towers of Hanoi

The Hanoi Tower that a Frenchman M. Claus (Lucas) from Thailand to France in 1883. Hanoi Tower which is supported by three diamond Pillars. At the beginning, God placed 64 gold discs from top to bottom on the first Pillar. God ordered the monks to move all gold discs from the first Pillar to the third Pillar. The principle of large plates under small plates during the handling process. If only one plate is moved daily, the tower will be destroyed. when all the discs are moved that is the end of the world.

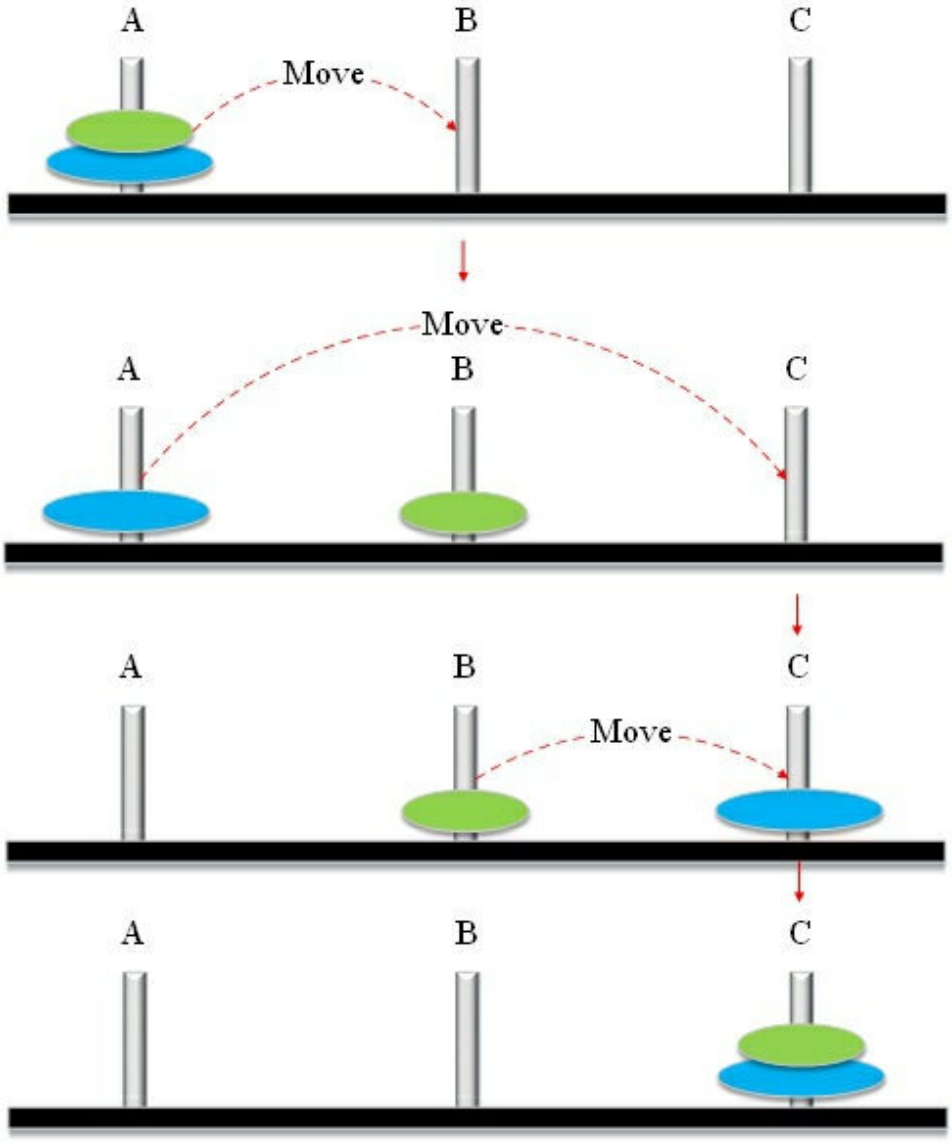
Let's turn this story into an algorithm:

Mark the three columns as ABC.
1. If there is only one disc, move it directly to C (A->C).
2. When there are two discs, use B as an auxiliary (A->B, A->C,B->C).
3. If there are more than two discs, use B as an auxiliary(A->B, A->C,B->C), and continue to recursive process.

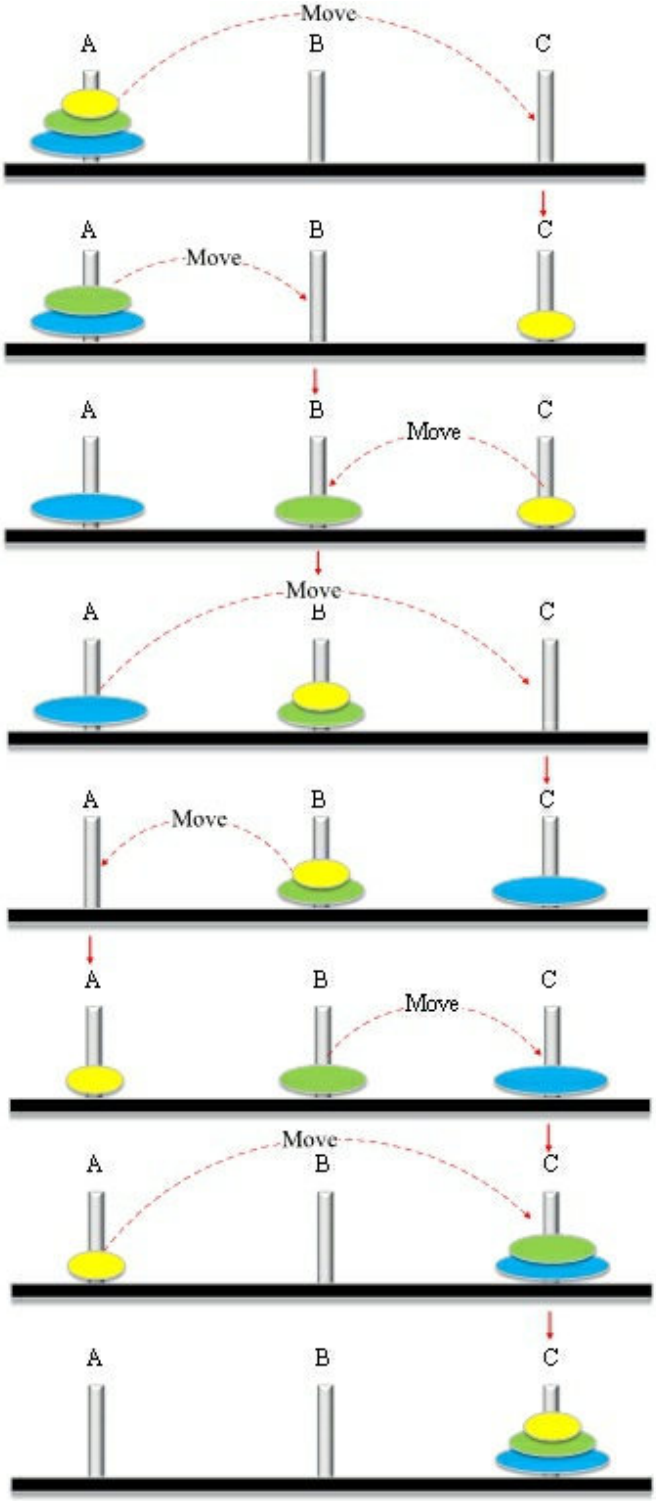
1. If there is only one disc, move it directly to C (A->C).



2. When there are two discs, use B as an auxiliary (A->B, A->C,B->C).



3. If more than two discs, use B as an auxiliary, and continue to recursive process.



TowersOfHanoi.go

```
package main
import "fmt"

func hanoi(n int, A string, B string, C string) {
    if n == 1 {
        fmt.Printf("Move %d %s to %s \n", n, A, C)
    } else {
        hanoi(n-1, A, C, B) // Move the n-1th disc on the A  through C to  B
        fmt.Printf("Move %d from %s to %s \n", n, A, C)
        hanoi(n-1, B, A, C) //Move the n-1th disc on the B  through  A to  C
    }
}

func main() {
    fmt.Printf("Please enter the number of discs : \n")
    var n int
    fmt.Scanf("%d", &n)
    hanoi(n, "A", "B", "C")
}
```

Result:

```
Please enter the number of discs :
1
Move 1 A  to C

Please enter the number of discs :
2
Move 1 A  to B
Move 2 from A  to C
Move 1 B  to C

Please enter the number of discs :
3
Move 1 A  to C
Move 2 from A  to B
Move 1 C  to B
```

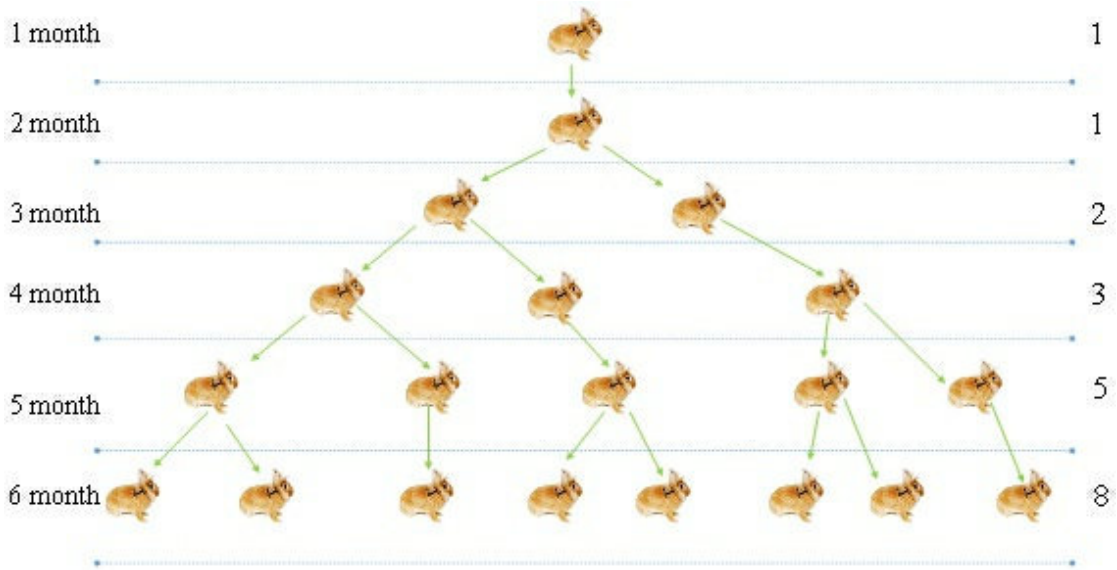
Move 3 from A to C
Move 1 B to A
Move 2 from B to C
Move 1 A to C

Fibonacci

Fibonacci : a European mathematician in the 1200s, in his writings: "If there is a rabbit
After a month can birth to a newborn rabbit. At first there was only 1 rabbit, after one month still 1 rabbit. after two month 2 rabbit, and after three months there are 3 rabbit .....
for example: 1, 1, 2, 3, 5, 8, 13 ...

Fibonacci definition:

if n = 0, 1
    fn = n
if n > 1
    fn = fn-1 + fn-2



Fibonacci.go

```
package main

import "fmt"

func fibonacci(n int) int {
    if n == 1 || n == 2 {
        return 1
    } else {
        return fibonacci(n-1) + fibonacci(n-2)
    }
}

func main() {
    fmt.Printf("Please enter the number of month : \n")
    var number int
    fmt.Scanf("%d", &number)

    for i := 1; i <= number; i++ {
        fmt.Printf("%d month: %d \n", i, fibonacci(i))
    }
}
```

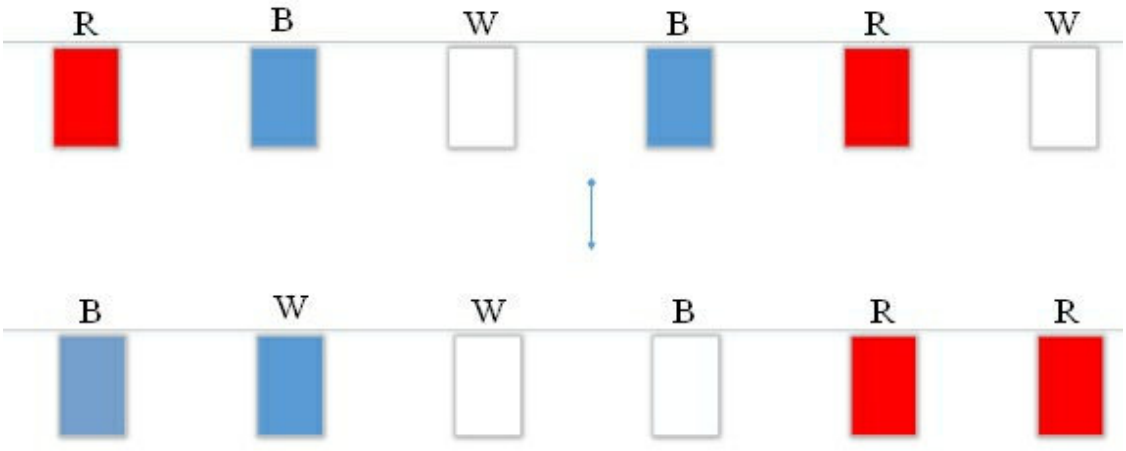
Result:

```
Please enter the number of month :
7
1 month : 1
2 month : 1
3 month : 2
4 month : 3
5 month : 5
6 month : 8
7 month : 13
```

Dijkstra

The tricolor flag was originally raised by E.W. Dijkstra, who used the Dutch national flag (Dijkstra is Dutch).

Suppose there is a rope with red, white, and blue flags. At first all the flags on the rope are not in order. You need to arrange them in the order of blue -> white -> red. How to move them with the least times. you just only do this on the rope, and only swap two flags at a time.



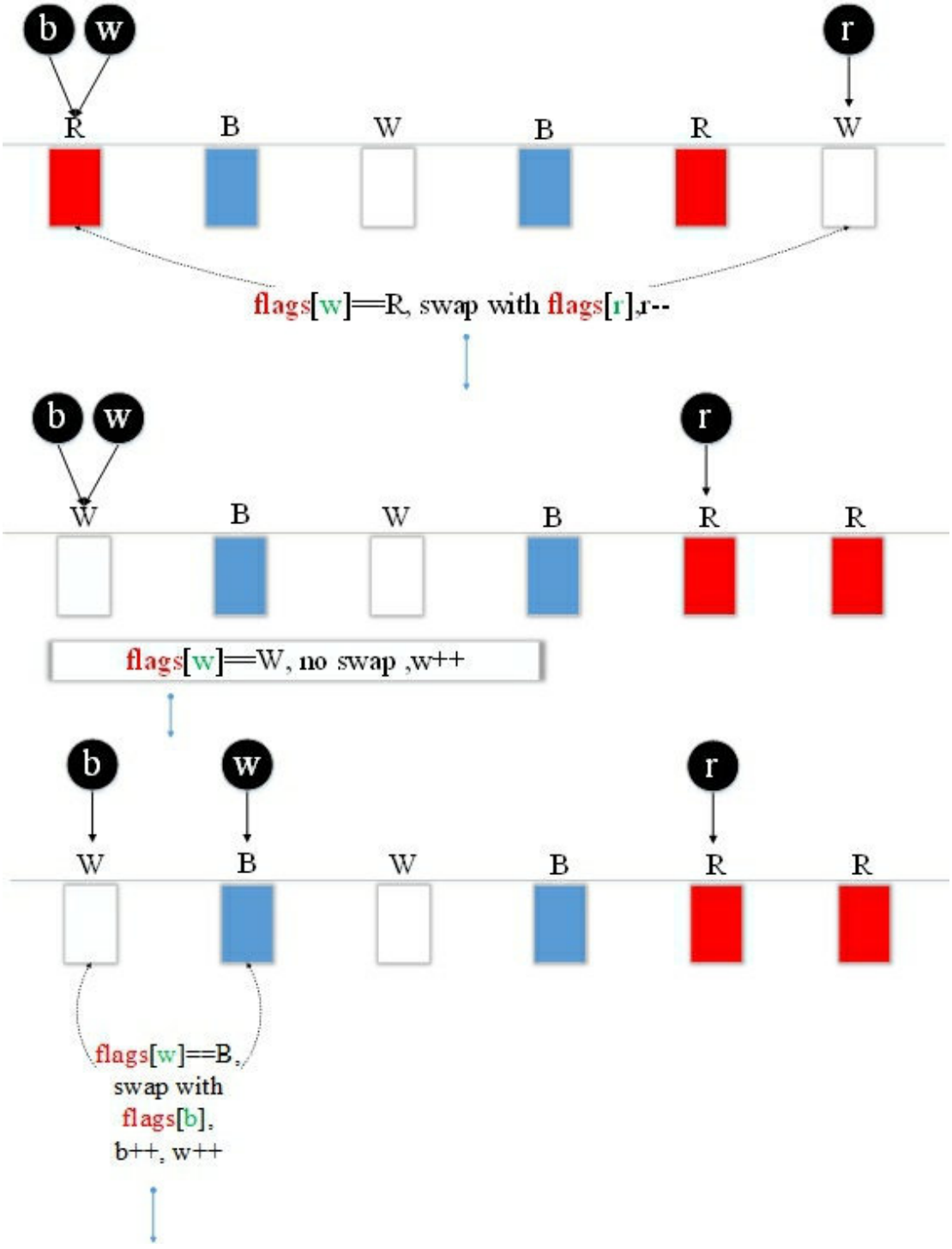
Solution:

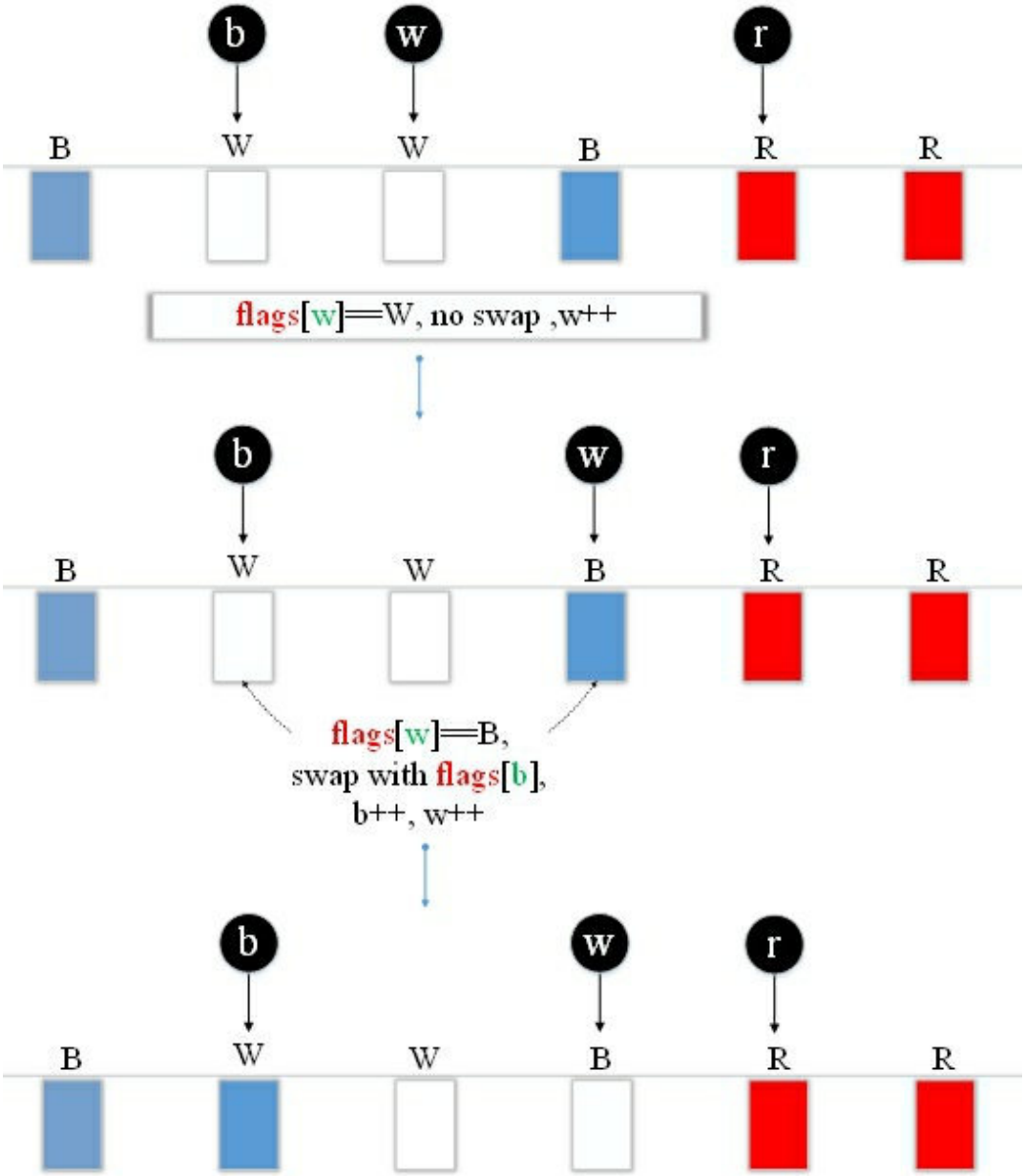
Use the char arrays to store the flags. For example, b, w, and r indicate the position of **blue**, **white** and **red** flags. The beginning of b and w is 0 of the array, and r is at the end of the array.

(1) If the position of w is a blue flag, flags[w] exchange with flags[b]. And whiteIndex and b is moved backward by 1.
(2) If the position of w is a white flag, w moves backward by 1.
(3) If the position of w is a red flag, flags[w] exchange with flags[r]. r moves forward by 1.

In the end, the flags in front of b are all blue, and the flags behind r are all red.

Graphic Solution




b
w
r
B
W
W
B
R
R
flags[w]==W, no swap ,w++
b
w
r
B
W
W
B
R
R
flags[w]==B,
swap with flags[b],
b++, w++
b
w
r
B
W
W
B
R
R

Dijkstra.go

```
package main
import "fmt"

func main() {
    var flags = []string{"R", "B", "W", "B", "R", "W"}
    var length = len(flags)
    var b = 0
    var w = 0
    var r = length - 1
    var count = 0
    for {
        if w > r {
         break
        }
        if flags[w] == "W" {
         w++
        } else if flags[w] == "B" {
         var temp = flags[w]
         flags[w] = flags[b]
         flags[b] = temp
         w++
         b++
         count++
        } else if flags[w] == "R" {
         var m = flags[w]
         flags[w] = flags[r]
         flags[r] = m
         r--
         count++
        }
    }

    for i := 0; i < length; i++ {
        fmt.Printf("%s", flags[i])
    }
    fmt.Printf("\nThe total exchange count : %d", count)
```

```
}
```

Result:
BBWWRR
The total exchange count : 4

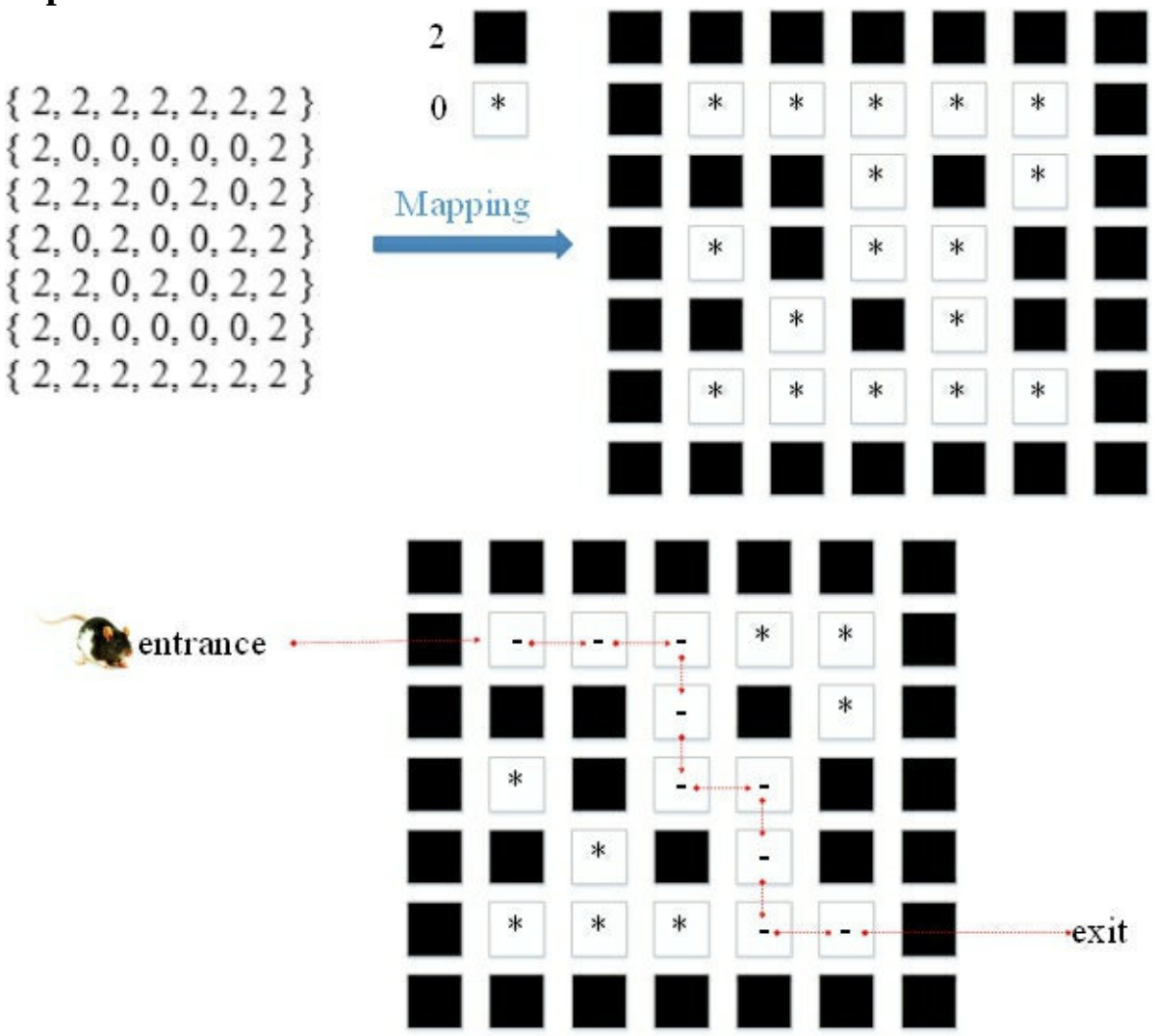
Mouse Walking Maze

Mouse Walking Maze is a basic type of recursive solution. We use 2 to represent the wall in a two-dimensional array, and use 1 to represent the path of the mouse, and try to find the path from the entrance to the exit.

Solution:

The mouse moves in four directions: up, left, down, and right. if hit the wall go back and select the next forward direction, so test the four directions in the array until mouse reach the exit.

Graphic Solution



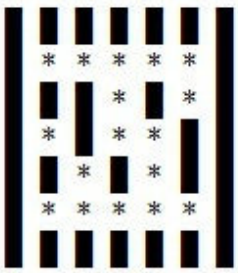
MouseWalkingMaze.go

```
package main
import "fmt"
var maze = [7][7]int{
    {2, 2, 2, 2, 2, 2, 2},
    {2, 0, 0, 0, 0, 0, 2},
    {2, 2, 2, 0, 2, 0, 2},
    {2, 0, 2, 0, 0, 2, 2},
    {2, 2, 0, 2, 0, 2, 2},
    {2, 0, 0, 0, 0, 0, 2},
    {2, 2, 2, 2, 2, 2, 2},
}

var startI = 1
var startJ = 1
var endI = 5
var endJ = 5
var success = 0

//The mouse moves in four directions: up, left, down, and right. if hit the
wall go back and select the next forward direction
func visit(i int, j int) int {
    maze[i][j] = 1
    if i == endI && j == endJ {
        success = 1
    }
    if success != 1 && maze[i][j+1] == 0 {
        visit(i, j+1)
    }
    if success != 1 && maze[i+1][j] == 0 {
        visit(i+1, j)
    }
    if success != 1 && maze[i][j-1] == 0 {
        visit(i, j-1)
    }
    if success != 1 && maze[i-1][j] == 0 {
        visit(i-1, j)
```

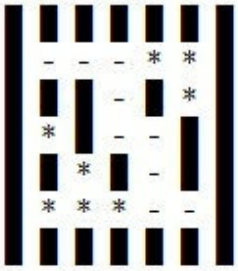
```
    }
    if success != 1 {
        maze[i][j] = 0
    }
    return success
}

func main() {
    fmt.Printf("Maze : \n")
    for i := 0; i < 7; i++ {
        for j := 0; j < 7; j++ {
            if maze[i][j] == 2 {
                fmt.Printf("  ")
            } else {
                fmt.Printf("*  ")
            }
        }
        fmt.Printf("\n")
    }

    if visit(startI, startJ) == 0 {
        fmt.Printf("No exit found \n")
    } else {
        fmt.Printf("Maze Path : \n")
        for i := 0; i < 7; i++ {
            for j := 0; j < 7; j++ {
                if maze[i][j] == 2 {
                    fmt.Printf("  ")
                } else if maze[i][j] == 1 {
                    fmt.Printf("-  ")
                } else {
                    fmt.Printf("*  ")
                }
            }
            fmt.Printf("\n")
        }
    }
}
```

Result:

Maze:



Maze Path :



Eight Coins

There are eight coins with the same appearance, one is a counterfeit coin, and the weight of counterfeit coin is different from the real coin, but it is unknown whether the counterfeit coin is lighter or heavier than the real coin. Please design a efficient algorithm to detect this counterfeit coin.

Solution:

Take six a, b, c, d, e, f from eight coins, and put three to the balance for comparison. Suppose a, b, c are placed on one side, and d, e, f are placed on the other side.

1.a + b + c> d + e + f
2.a + b + c = d + e + f
3.a + b + c <d + e + f

If a + b + c> d + e + f, there is a counterfeit coin in one of the six coins, and g, h are real coins. At this time, one coin can be removed from both side. Suppose that c and f are removed. At the same time, one coin at each side is replaced. Suppose the coins b and e are interchanged, and then the second comparison. There are also three possibilities:

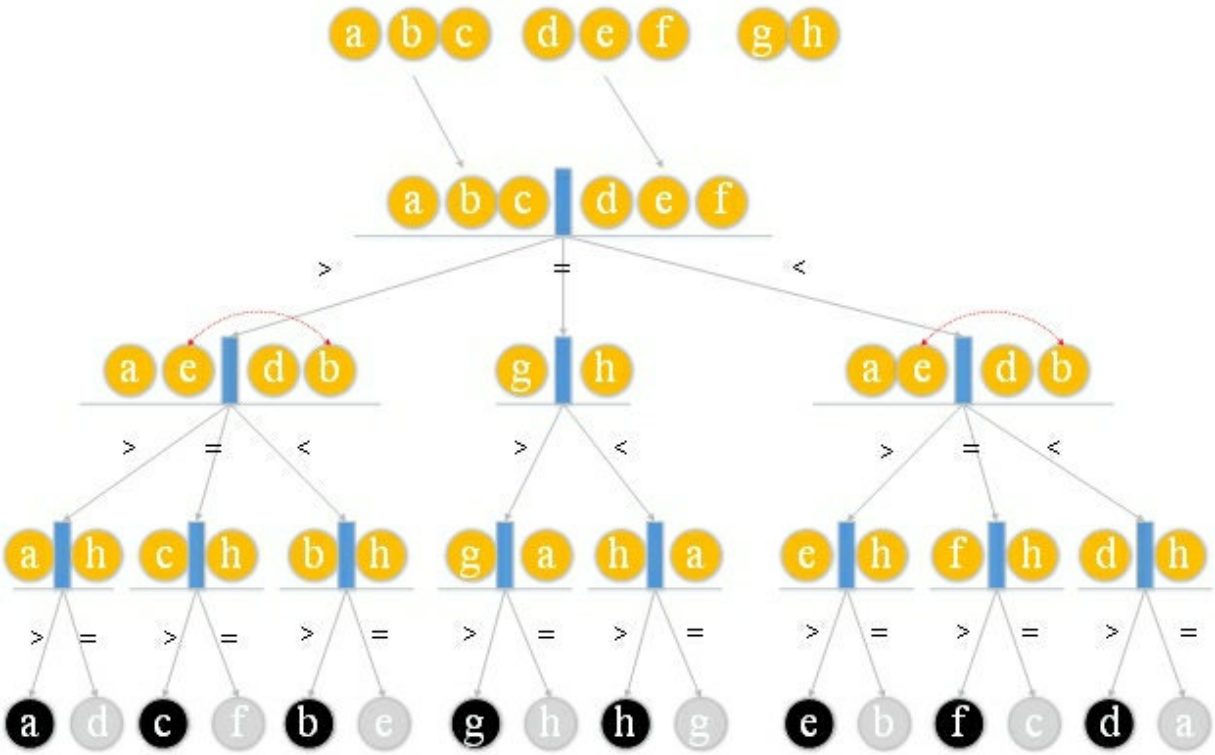
1. a + e> d + b: the counterfeit currency must be one of a, d. as long as we compare a real currency h with a, we can find the counterfeit currency. If a> h, a is a heavier counterfeit currency; if a = h, d is a lighter counterfeit currency.

2. a + e = d + b: the counterfeit currency must be one of c, f, and the real coin h is compared with c. If c> h, c is a heavier counterfeit currency; if c = h, then f is a lighter counterfeit currency.

3. a + e <d + b: one of b or e is a counterfeit coin , Also use the real coin h to compare with b, if b> h, then b is a heavier counterfeit currency; if b = h, then e is a lighter counterfeit currency;

Graphic Solution

Heavy **Light**


a b c d e f g h
a b c d e f
> = <
a e d b
g h
a e d b
> = <
> <
> = <
a h
c h
b h
g a
h a
e h
f h
d h
> = > = > = > = > = > = > = > =
a d c f b e g h h g e b f c d a

EightCoins.go

```
package main
import (
    "fmt"
    "math/rand"
)

func compare(coins []int, i int, j int, k int) { //coin[k] true,coin[i]>coin[j]
    if coins[i] > coins[k] { //coin[i]>coin[j]&&coin[i]>coin[k] ----->coin[i]
is a heavy counterfeit coin
        fmt.Printf("\nCounterfeit currency %d  is heavier ", (i + 1))
    } else { //coin[j] is a light counterfeit coin
        fmt.Printf("\nCounterfeit currency %d  is lighter ", (j + 1))
    }
}

func eightcoins(coins []int) {
    if coins[0]+coins[1]+coins[2] == coins[3]+coins[4]+coins[5] {
//(a+b+c)==(d+e+f)
        if coins[6] > coins[7] { //g>h?(g>a?g:a):(h>a?h:a)
            compare(coins, 6, 7, 0)
        } else { //h>g?(h>a?h:a):(g>a?g:a)
            compare(coins, 7, 6, 0)
        }
    } else if coins[0]+coins[1]+coins[2] > coins[3]+coins[4]+coins[5] {
//(a+b+c)>(d+e+f)
        if coins[0]+coins[3] == coins[1]+coins[4] { //(a+e)==(d+b)
            compare(coins, 2, 5, 0)
        } else if coins[0]+coins[3] > coins[1]+coins[4] { //(a+e)>(d+b)
            compare(coins, 0, 4, 1)
        }
        if coins[0]+coins[3] < coins[1]+coins[4] { //(a+e)<(d+b)
            compare(coins, 1, 3, 0)
        }
    } else if coins[0]+coins[1]+coins[2] < coins[3]+coins[4]+coins[5] {
//(a+b+c)<(d+e+f)
        if coins[0]+coins[3] == coins[1]+coins[4] { //(a+e)>(d+b)
```

```
            compare(coins, 5, 2, 0)
        } else if coins[0]+coins[3] > coins[1]+coins[4] { //(a+e)>(d+b)
            compare(coins, 3, 1, 0)
        }
        if coins[0]+coins[3] < coins[1]+coins[4] { //(a+e)<(d+b)
            compare(coins, 4, 0, 1)
        }
    }
}

func main() {
    var coins = make([]int, 8)
    // Initial coin weight is 10
    for i := 0; i < 8; i++ {
        coins[i] = 10
    }

    fmt.Printf("Enter weight of counterfeit currency  (larger or smaller than
10) :")
    var coin int
    fmt.Scanf("%d", &coin)
    var index = rand.Intn(8)
    coins[index] = coin

    eightcoins(coins)

    for i := 0; i < 8; i++ {
        fmt.Printf("%d , ", coins[i])
    }
}
```

Result:

First run:
Enter weight of counterfeit currency (larger or smaller than 10) :
2

Counterfeit currency 2 is lighter
10 , 2 , 10 , 10 , 10 , 10 , 10 , 10 ,

Run again:
Enter weight of counterfeit currency (larger or smaller than 10) :
13

Counterfeit currency 4 is heavier
10 , 10 , 10 , 13 , 10 , 10 , 10 , 10 ,

Knapsack Problem

Suppose you have a backpack with a weight of up to 8 kg, and you want to the backpack with a total price of Products, suppose the fruit (ID, Name,Price and Weight)

ID	Name	Price	Weight
0	Plum	4kg	4500
1	Apple	5kg	5700
2	Orange	2kg	2250
3	Strawberry	1kg	1100
4	Melon	6kg	6700

Solution:

To solve the optimization problem we can use Dynamic Programming. In the beginning there is a empty set, every time add an element, find the best solution at this stage, until all elements are added. After entering the set finally can get the best solution.

There are two arrays, value and item
value: the total price of the current best solution.
item : the last fruit in the backpack.
there are 8 backpacks with a weight of 1 to 8, and find the best solution for each backpack.

Gradually put the fruit in the backpack and find the best solution:

1. **Put in plums:**

Weight of Backpack	1kg	2 kg	3 kg	4 kg	5 kg	6 kg	7 kg	8 kg
Value	0	0	0	4500	4500	4500	4500	9000
Item	-	-	-	0	0	0	0	0 , 0
Name	-	-	-	Plum	Plum	Plum	Plum	2* Plum

2. **Put in apples:**

Weight of	1kg	2	3	4 kg	5 kg	6 kg	7 kg	8 kg

Backpack		kg	kg					
Value	0	0	0	4500	5700	5700	5700	9000
Item	-	-	-	0	1	1	1	0 , 0
Name	-	-	-	Plum	Apple	Apple	Apple	2* Plum

3. Put in oranges:

Weight of Backpack	1kg	2 kg	3 kg	4 kg	5 kg	6 kg	7 kg	8 kg
Value	0	2250	2250	4500	5700	6750	7950	9000
Item	-	2	2	0	1	0 , 2	1 , 2	0 , 0
Name	-	Orange	Orange	Plum	Apple	Orange Plum	Orange Apple	2* Plum

4. Put in strawberrys:

Weight of Backpack	1kg	2 kg	3 kg	4 kg	5 kg	6 kg	7 kg	8 kg
Value	1100	2250	3350	4500	5700	6800	7950	9050
Item	3	2	2 , 3	0	1	0 , 3	1 , 2	1 , 2, 3
Name	Strawberry	Orange	Orange Strawberry	Plum	Apple	Plum Strawberry	Orange Apple	Strawberry Orange Apple

5. Put in melons:

Weight of Backpack	1kg	2 kg	3 kg	4 kg	5 kg	6 kg	7 kg	8 kg
Value	1100	2250	3350	4500	5700	6800	7950	9050
Item	3	2	2 , 3	0	1	0 , 3	1 , 2	1 , 2, 3
Name	Strawberry	Orange	Orange Strawberry	Plum	Apple	Plum Strawberry	Apple Orange	Apple Orange Strawberry

From the last table that when the backpack weighs 8 kg, a maximum of 9050, so the best solution is to put Strawberries, Oranges and Apples, and the total price is 9050.

Knapsack.go

```
package main
import "fmt"

const MAXSIZE = 8
const MINSIZE = 1

type Fruit struct {
    name  string
    size  int
    price int
}

func main() {
    var item = [MAXSIZE + 1]int{0}
    var value = [MAXSIZE + 1]int{0}
    var fruits = [5]Fruit{
        {"Plum", 4, 4500},
        {"Apple", 5, 5700},
        {"Orange", 2, 2250},
        {"Strawberry", 1, 1100},
        {"Melon", 6, 6700},
    }

    var length = len(fruits)
    for i := 0; i < length; i++ {
        for j := fruits[i].size; j <= MAXSIZE; j++ {
            var p = j - fruits[i].size
            var newValue = value[p] + fruits[i].price
            if newValue > value[j] { // Find the best solution
                value[j] = newValue
                item[j] = i
            }
        }
    }

    fmt.Printf("Item \t Price \n")
    for i := MAXSIZE; i >= MINSIZE; i = i - fruits[item[i]].size {
```

```
        fmt.Printf("%s\t %d \n", fruits[item[i]].name, fruits[item[i]].price)
    }
    fmt.Printf("Total \t %d", value[MAXSIZE])
}
```

Result:

```
Item        Price
Strawberry     1100
Orange       2250
Apple        5700
Total        9050
```

Josephus Problem

There are 9 Jewish hid in a hole with Josephus and his friends . The 9 Jews decided to die rather than be caught by the enemy, so they decided In a suicide method, 11 people are arranged in a circle, and the first person reports the number. After each number is reported to the third person, the person must commit suicide. Then count again from the next one until everyone commits suicide.But Josephus and his friends did not want to obey. Josephus asked his friends to pretend to obey, and he arranged the friends with himself. In the 2th and 7st positions, they escaped this death game.

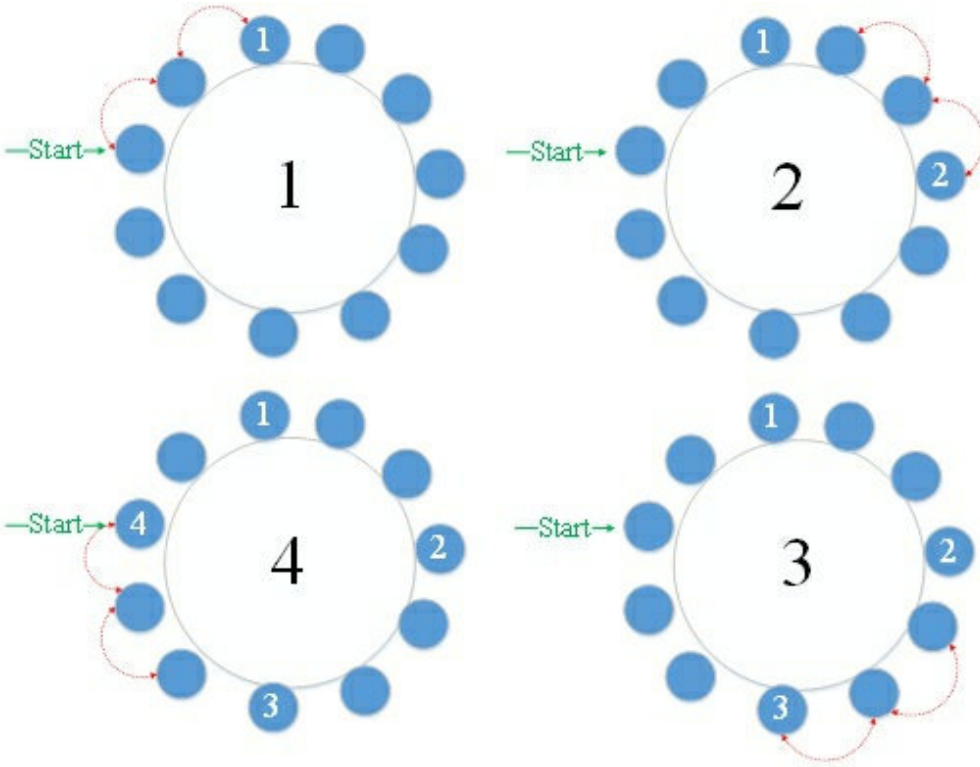
Solution:

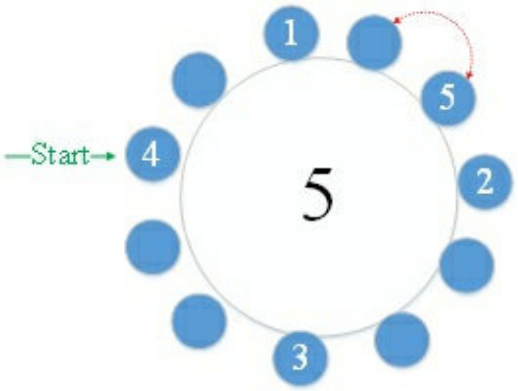
As long as the array is treated as a ring. Fill in a count for each dataless area, until the count reaches 11, and then list the array from index 1, you can know that each suicide order in this position is the Joseph's position. The 11-person position is as follows:

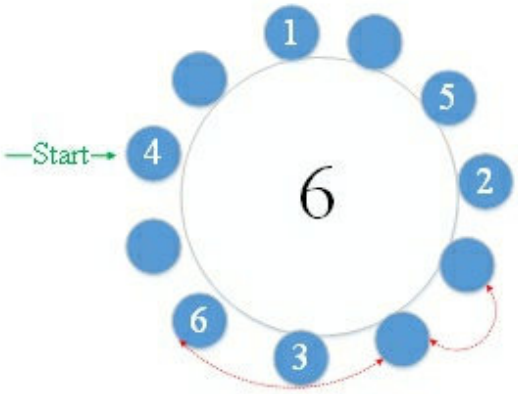
4 10 1 7 5 2 11 9 3 6 8

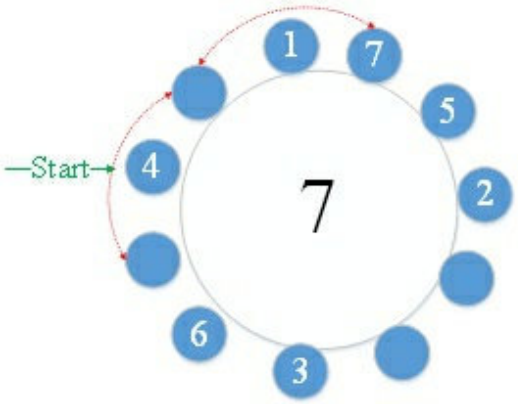
From the above, the last two suicide was in the 31st and 16th position. The previous one

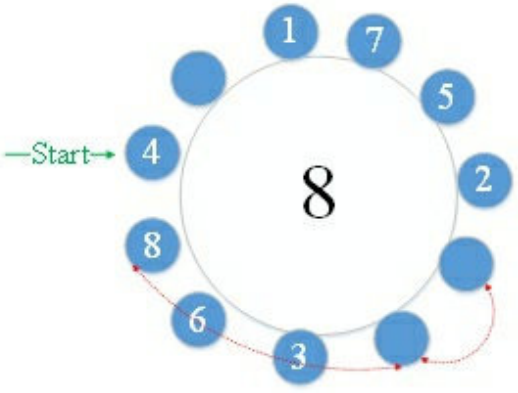
Everyone died, so they didn't know that Joseph and his friends didn't follow the rules of the game.

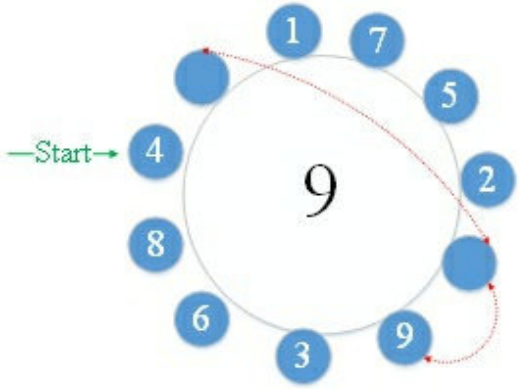

Start
1
Start
2
Start
4
Start
3

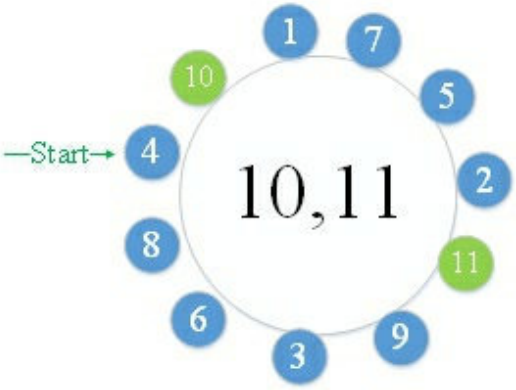

Start
1
5
2
3
4
5


Start
1
5
2
3
6
4
6


Start
1
7
5
2
3
6
4
7


Start
1
7
5
2
3
6
8
4
8


Start
1
7
5
2
3
9
6
8
4
9


Start
1
7
10
5
2
11
3
9
6
8
4
10,11

Joseph.go

```
package main

import "fmt"

const N = 11
const M = 3

func main() {
    var man = [N]int{0}
    var count = 1
    var i = 0
    var pos = -1

    for {
        if count > N {
      break
        }
        for {
      pos = (pos + 1) % N // Ring
      if man[pos] == 0 {
        i++
      }
      if i == M {
        i = 0
        break
      }
        }
        man[pos] = count
        count++
    }
    fmt.Printf("\nJoseph sequence : ")
    for i := 0; i < N; i++ {
        fmt.Printf("%d , ", man[i])
    }
}
```

Result:

Joseph sequence：
4 , 10 , 1 , 7 , 5 , 2 , 11 , 9 , 3 , 6 , 8 ,

If you enjoyed this book and found some benefit in reading this, I'd like to hear from you and hope that you could take some time to post a review on Amazon. Your feedback and support will help us to greatly improve in future and make this book even better.

You can follow this link now.

http://www.amazon.com/review/create-review?&asin=B08GS8B5HQ

Different country reviews only need to modify the amazon domain name in the link:

www.amazon.co.uk
www.amazon.de
www.amazon.fr
www.amazon.es
www.amazon.it
www.amazon.ca
www.amazon.nl
www.amazon.in
www.amazon.co.jp
www.amazon.com.br
www.amazon.com.mx
www.amazon.com.au

I wish you all the best in your future success!