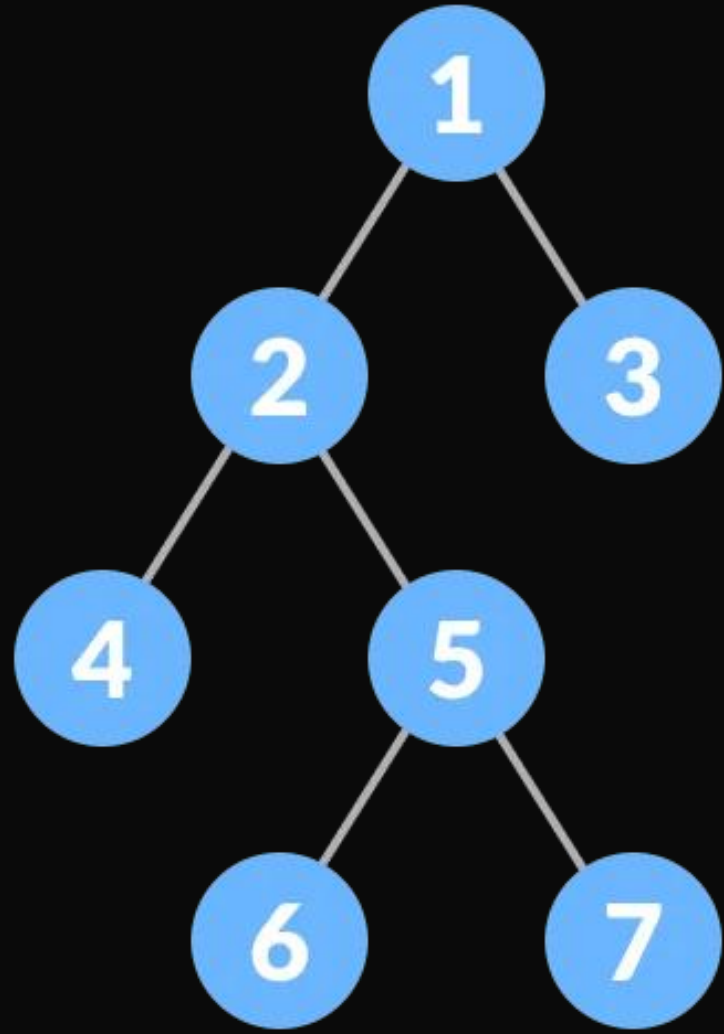
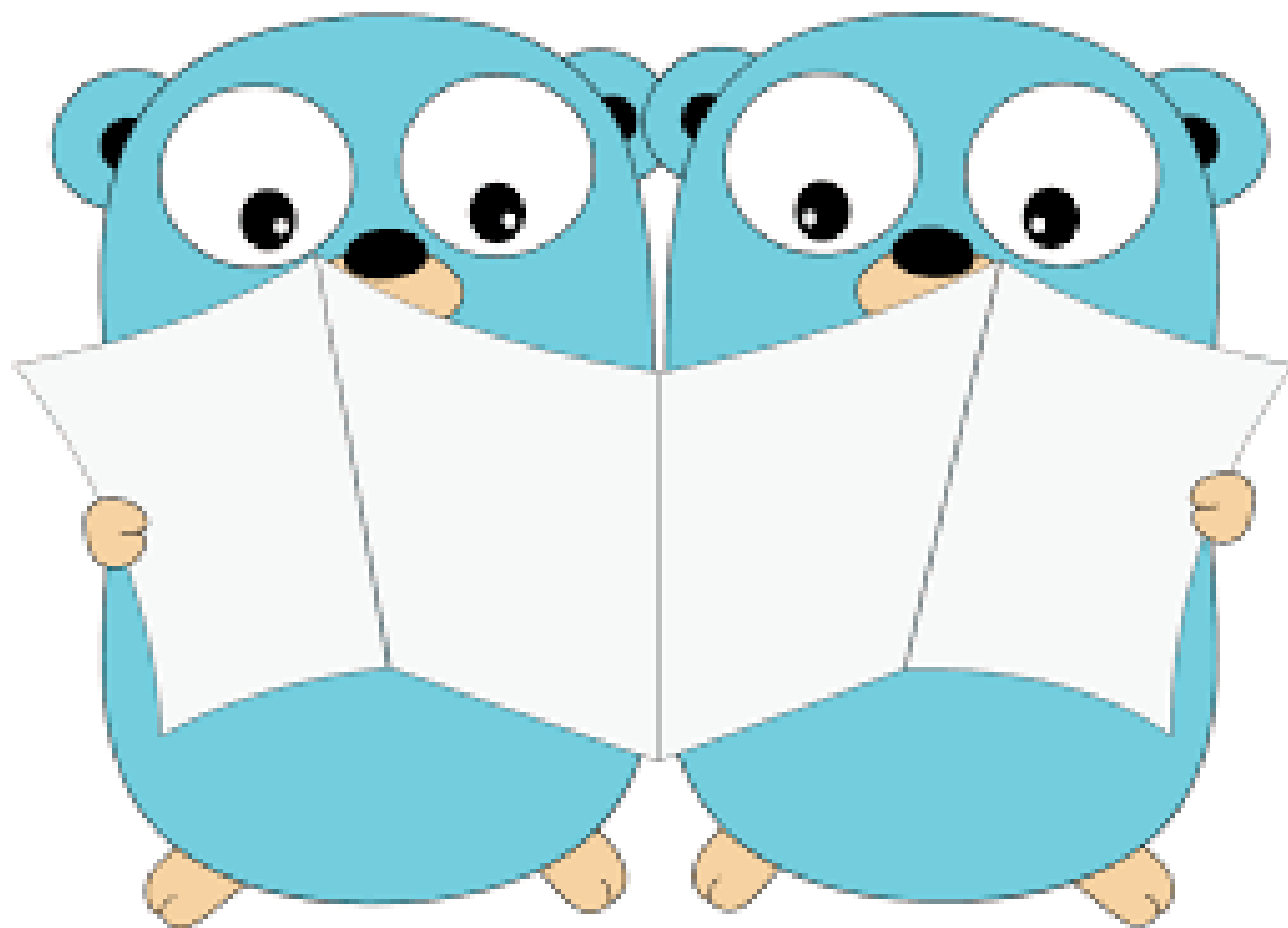


Data Structures In



Contents



1 Linked List

2 Stack

3 Queue

4 Tree

5 Graph

6 Sorting

7 Searching

Linked List

Linked lists are a data structure that stores data in a linked list of nodes.

Linked lists are efficient for storing data that needs to be accessed in a non-sequential order.

To initialize create 2 structs as Linked List and Node

there are 4 primary methods on LinkedList :-

- Create
- Insert
- InsertN
- Remove
- Reverse

```
1 // Linked List
2 type LinkedList struct {
3     Head *Node
4     Length int
5 }
6
7 type Node struct {
8     Data int
9     Next *Node
10 }
11
12 func main() {
13
14     myl := LinkedList{Head: nil}
15     myl.Create(1)
16     myl.Create(2)
17
18 }
19
```

Create Node

Create method on LinkedList takes integer as data and creates a new node elem. It checks if there is no head then the elem node will be the first node else it goes to all nodes by node Next and adds the node where the node is nil

```
1 func (l *LinkedList) Create(data int) {
2     elem := &Node{Data: data, Next: nil}
3     if l.Head == nil {
4         l.Head = elem
5     } else {
6         current := l.Head
7         for current.Next != nil {
8             current = current.Next
9         }
10        current.Next = elem
11    }
12 }
13
```

Reverse LinkedList

To reverse a linked list first check for its head.
We need 3 variables of node type as current next and previous.
we need to store the value of current and next and previous before changing any pointer address.
once done then current can move to next node with next variable address and point it to pervious node with previous variable

```
1 func (l *LinkedList) reverse() {
2     if l.Head == nil {
3         fmt.Println("no list")
4     } else {
5         current := l.Head
6         var prev *Node
7         var next *Node
8
9         for current != nil {
10             next = current.Next //first
11             current.Next = prev // set n
12             prev = current      //change
13             current = next      //move t
14         }
15         l.Head = prev //add head pointer
16
17     }
18
19 }
```


Stack

Stacks are a data structure that stores data in a last-in, first-out (LIFO) order.

This means that the last item that is added to the stack is the first item that is removed.

Stacks can be implemented using underlying array or linked list.

they work on last in first out principal.

the main methods on stacks are

Push -> add on top

Pop -> remove from top

Peek -> see value on top

```
1 // Lifo
2 // Stack implementation with slice
3 type Stack struct {
4     Items []int
5 }
6
7 func main() {
8
9     var mys Stack
10    mys.Push(99)
11    mys.Push(1)
12    mys.Push(2)
13    mys.Push(3)
14    mys.Push(4)
15
16 }
```

Stack Push

Stack push method takes argument int and appends it on the last end in case of a array [slice for go]

```
1 func (s *Stack) Push(data int) {  
2     s.Items = append(s.Items, data)  
3 }
```

Stack Pop

On using pop method on stack the last value of underlying array and removes it

```
1 func (s *Stack) Pop() (int, bool) {  
2     if len(s.Items) == 0 {  
3         return 0, false  
4     } else {  
5         lastItem := len(s.Items) - 1  
6         s.Items = s.Items[:len(s.Items)-1]  
7         return lastItem, true  
8     }  
9 }  
10
```


Peek Stack

Peek method on stack looks for the last variable in the array and returns int value

```
1 func (s *Stack) Peek() int {  
2     if len(s.Items) == 0 {  
3         fmt.Println("no stack")  
4         return 0  
5     } else {  
6         lastItem := len(s.Items) - 1  
7         return lastItem  
8     }  
9 }  
10
```

Queue

Queues are a data structure that stores data in a first-in, first-out (FIFO) order.

This means that the first item that is added to the queue is the first item that is removed. Queues are efficient for storing data that needs to be processed in a FIFO order.

Main methods on Queue are


- Enqueue -> adds variable on last
- Dequeue -> takes the first value out

```
1 // first in first out
2 // Queue implementation
3 type Queue struct {
4     Items []int
5 }
6
7 func main() {
8     myq := Queue{}
9     myq.enqueue(100)
10    myq.enqueue(200)
11    fmt.Println(myq.dequeue())
12    fmt.Println(myq.dequeue())
13 }
```

Enqueue & Dequeue

Enqueue takes the data int and appends it to the underlying slice

Dequeue reads the first item in underlying slice and return and removes it



```
1  func (q *Queue) enqueue(data int) int {
2      q.Items = append(q.Items, data)
3      return data
4
5  }
6
7  func (q *Queue) dequeue() (int, bool) {
8      if len(q.Items) == 0 {
9          return 0, false
10     } else {
11         firstItem := q.Items[0]
12         q.Items = append(q.Items[1:])
13         return firstItem, true
14     }
15 }
16
```

Tree

Trees are a data structure that stores data in a hierarchical structure. Trees are efficient for storing data that has a natural hierarchy, such as the file system on a computer.

Trees can be of many types

Binary tree → both sides with max 2 nodes

Tries → one node can have multiple nodes

```
1 // Tree
2 type Tree struct {
3     Data int
4     Left *Tree
5     Right *Tree
6 }
7
8 func main() {
9     t := &Tree{Data: 50, Left: nil, Right: nil}
10    t.Insert(100)
11    t.Insert(20)
12    t.Insert(55)
13    t.Insert(23)
14    t.Insert(11)
15 }
```


Insert Tree Node

To insert data into a tree when we use Insert method. when the data is added then a temp node is created and all the branches are looked such as if the data is small than right node then left node is looked for empty space and vice versa. once a empty node is found then the temp node is added to left if data is small than right.

```
1 func (t *Tree) Insert(data int) *Tree {
2     temp := Tree{Data: data, Left: nil, Right: nil}
3     if t.Data == 0 {
4         t.Left = &temp
5         return &temp
6     } else {
7         current := t
8         for {
9             if data < current.Data {
10                // left
11                if current.Left == nil {
12                    current.Left = &temp
13                    return current
14                }
15                current = current.Left
16            } else {
17                // right
18                if current.Right == nil {
19                    current.Right = &temp
20                    return current
21                }
22                current = current.Right
23            }
24        }
25    }
26 }
27 }
```


Tree Lookup

To search if a values is in the tree we can use BFS, DFS . Our approach is to look for all the nodes in left then to right . if we find a value that matches the lookup value then we return the value else we return nil



```
1  func (t *Tree) Lookup(data int) *Tree {
2      if t == nil {
3          fmt.Println("non tree")
4      } else {
5          current := t
6          for current != nil {
7              if data < current.Data {
8                  current = current.Left
9              } else if data > current.Data {
10                 current = current.Right
11             } else if current.Data == data {
12                 return current
13             }
14         }
15     }
16     return nil
17 }
```


Traverse Tree

Lookup of Binary tree has several methods such as Breadth first search or depth first search.

for this approach we can traverse a tree by checking all the nodes from left to right and add the node data into a array.



```
1  // in order Traverse
2  func Traverse(root *Tree) []int {
3      if root == nil {
4          return nil
5      }
6      output := make([]int, 0)
7      left := Traverse(root.Left)
8      right := Traverse(root.Right)
9
10     output = append(output, left...)
11     output = append(output, root.Data)
12     output = append(output, right...)
13     return output
14 }
15
```

Graph

Graphs are a data structure that stores data in a network of nodes and edges. Graphs are efficient for storing data that can be represented as a network, such as the social network of friends and family.


Graph can be weighted, directed or cyclic.

There are ways to represent graphs


- Adjacent Matrix -> 2d matrix
- Adjacent list -> used mostly because of better $O(n)$
- Incidence matrix

```
1  type Vertex struct {
2      Key      int
3      Vertices []*Vertex
4  }
5
6  type Graph struct {
7      vertices []*Vertex
8  }
9
10 func main() {
11     g := Graph{}
12     g.add_vertices(1)
13     g.add_edges(1, 2)
14 }
15
```

Add & Get Vertex



```
1 func (g *Graph) get_vertex(k int) *Vertex {
2     for _, v := range g.vertices {
3         if v.Key == k {
4             return v
5         }
6     }
7     return nil
8 }
```



```
1 func (g *Graph) add_vertices(v int) {
2     temp := &Vertex{Key: v}
3     g.vertices = append(g.vertices, temp)
4 }
5
```

Add Edges



```
1 // add edges to each vertexes
2 func (g *Graph) add_edges(to, from int) {
3     // change int to vertices
4     toVertex := g.get_vertex(to)
5     fromVertex := g.get_vertex(from)
6     // in graph take to and from points
7     for _, v := range g.vertices {
8         if v.Key == toVertex.Key {
9             toVertex.Vertices = append(toVertex.Vertices, fromVertex)
10            // append from.vertices to to
11        }
12    }
13
14 }
15
```

Show Graph



```
1
2 //showGraph prints all vertex and list of related vertexes
3 func (g *Graph) showGraph() {
4     for _, v := range g.vertices {
5         fmt.Printf("%v", v.Key)
6         for _, e := range v.Vertices {
7             fmt.Printf("--> %v", e.Key)
8         }
9         fmt.Println()
10    }
11 }
12
```


Merge Sort

merge sort sorts the array by breaking down the array into sub arrays until each value is individually compared to its right value and interchanged if needed then the values are merged back repeatedly to give out a completed sorted array

```
1 func mergeSort(a []int) []int {  
2     if len(a) < 2 {  
3         return a  
4     }  
5     right := a[:len(a)/2]  
6     left := a[len(a)/2:]  
7  
8     return merge(mergeSort(left), mergeSort(right))  
9  
10 }  
11
```


Merge function of merge sort takes the arrays and then creates a new array. all the elements are compared and added to final array then the final array is appended to the other arrays and returned

```
1 func merge(a []int, b []int) []int {
2     final := []int{}
3     i := 0
4     j := 0
5     // sort here
6     for i < len(a) && j < len(b) {
7         if a[i] < b[j] {
8             final = append(final, a[i])
9             i++
10        } else {
11            final = append(final, b[j])
12            j++
13        }
14    }
15    // merge here
16    for ; i < len(a); i++ {
17        final = append(final, a[i])
18    }
19    for ; j < len(b); j++ {
20        final = append(final, b[j])
21    }
22    return final
23 }
```


Quick Sort

Quicksort algorithm for sorting arrays efficiently. it uses a random pivot and adds all values less to left and more to right and then again sets a new pivot and repeats until the array is sorted. the time complexity is reduced to $O(\log(n))$ for best cases.

```
1 // starter
2 func quicksortStart(a []int) []int {
3     return quicksort(a, 0, len(a)-1)
4 }
5
6 // quicksort
7 func quicksort(a []int, low, high int) []int {
8     if low < high {
9         var p int
10        a, p = partition(a, low, high)
11        a = quicksort(a, low, p-1)
12        a = quicksort(a, high, p+1)
13    }
14    return a
15 }
16
```

Partition

Partition creates a partition of array and sets the high and low. It uses the pivot and changes the values as less than pivot to less and more than pivot to right and returns the array back with index



```
1  // partition
2  func partition(a []int, low, high int) ([]int, int) {
3      pivot := a[high]
4      i := low
5
6      for j := low; j < high; j++ {
7          if a[j] < pivot {
8              a[i], a[j] = a[j], a[i]
9              i++
10         }
11     }
12     a[i], a[high] = a[high], a[i]
13     return a, i
14 }
15
```

Breadth First Search

BFS is user to traverse a tree or graph. It goes to all near nodes then goes to a depth and goes to all the nodes and repeat.

It is used to find the shortest path. or travers a graph or tree

```
1 func BFST(tree *Tree) []int {
2     result := []int{}
3     queue := []*Tree{}
4     queue = append(queue, tree)
5     return bfstUtil(queue, result)
6 }
7
8 func bfstUtil(queue []*Tree, result []int) []int {
9     if len(queue) == 0 {
10         return result
11     }
12     result = append(result, queue[0].Data)
13
14     if queue[0].Right != nil {
15         queue = append(queue, queue[0].Right)
16     }
17     if queue[0].Left != nil {
18         queue = append(queue, queue[0].Left)
19     }
20     return bfstUtil(queue[1:], result)
21 }
22
```

Shubham Panchal

