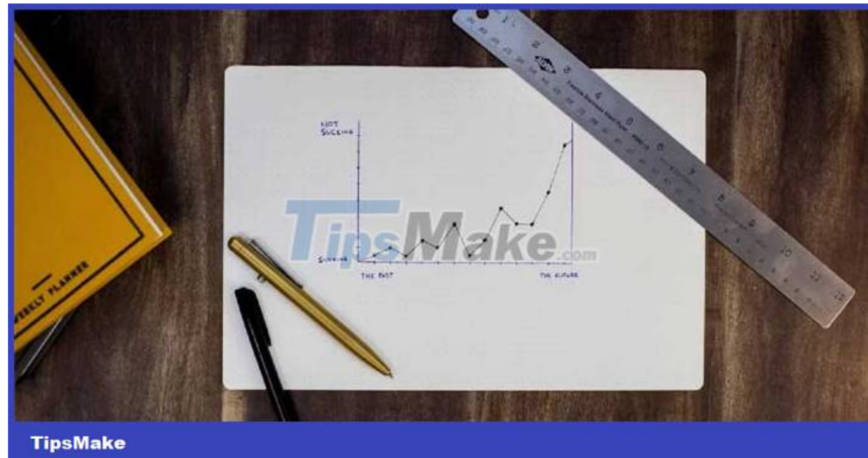
How to implement a graph data structure in Golang

Charts are one of the essential data structures that you must know as a programmer. Let's learn how to create graph/graph data structures in Golang !



Problems related to graphs are common in the field of software engineering, from technical interviews to building applications using graphs.

Graphs are basic data structures used in applications ranging from social networks and transportation systems to network analysis and recommendation engines.

So what is a chart? How can you implement graphs in Go?

What are graphs and charts?

A graph is a non-linear data structure that represents a collection of nodes (or vertices) and the connections between them (edges). Graphs are widely used in 'connection-heavy' software applications such as computer networks, social networks and more.

Graphs are one of the data structures that you should know as a programmer. Charts provide a powerful & flexible way to prototype and analyze a variety of real-world scenes and make them a core & fundamental data structure in computer science.

A variety of problem solving algorithms are now used in the world of graph/graph based software.

Graph implementation in Golang

When you implement a data structure yourself, you almost always need to implement object-oriented programming (OOP) concepts, but creating OOP in Go is not like that because you have it in other languages ?? like Java and C++. .

Go uses structs, types, and interfaces to implement OOP concepts. They are all you need to create a graph data structure and its methods.

A graph contains nodes (or vertices) & edges. Node is an entity or element in a graph. For example, a node is a device on a network or a person on a social network, and an edge is a connection or relationship between two nodes.

To implement a graph in Go, you first need to define a node structure that has the property of being its neighbor. Neighbors of a node are other nodes that are directly connected to that node.

The following code illustrates how Node is structured :

```
type Node struct { Neighbors []*Node }
```

The instructional focus will be on the undirected chart. However, for the sake of clarity, here is a Node construct for directed graphs:

```
type Node struct { OutNeighbors []*Node // outgoing edges InNeighbors []*Node //
```

With this definition, the OutNeighbors slice will contain the nodes with the edges going from the current node, and the InNeighbors slice will contain the node where the edges go to the current node.

You will implement a graph using a map of integers leading to node. This map works like adjacency list . The key will act as the unique ID for the node, and the value will be node.

The following code will show the Graph structure :

```
type Graph struct { nodes map[int]*Node }
```

The integer key can also be visualized as the value of the node it is mapped to. Either way, in real-life contexts, your node could be another data structure, representing an individual's profile or something like that. In such cases, you should have data as one of the properties of the Node.

You need a function that acts as the new graph constructor. This will allocate memory for the contiguous list and also allow you to add nodes to the graph. The code below defines the constructor for the Graph class :

```
func NewGraph() *Graph { return &Graph{ nodes: make(map[int]*Node), } }
```

You can now define methods that perform a variety of operations on the graph, from adding nodes to creating edges between nodes, finding nodes.

Add nodes to the graph

You need the insert function like this:

```
func (g *Graph) AddNode(nodeID int) { if _, exists := g.nodes[nodeID]; !exists {
```

The AddNode function adds a new node to the graph with the passed ID to it as a parameter. This function checks to see if a node with the same ID already exists before adding it to the graph.

Add edge to chart

The next important method of the histogram data structure is the add edge function. Since this graph is undirected, you don't need to worry about the direction when creating the edge.

Here is the function that adds an edge between two nodes on the graph:

```
func (g *Graph) AddNode(nodeID int) { if _, exists := g.nodes[nodeID]; !exists {
```

Adding an edge in an undirected graph is simply the process of creating two adjacent nodes. This function takes both nodes by the ID passed to it and joins the two to each other's Neighbors slice .

Remove edge from chart

To remove a node from the graph, you need to remove it from all related neighbor lists to ensure there are no data inconsistencies.

The process of removing nodes from all neighbors is the same as removing edges (or disconnecting) between nodes, so you must define the edge removal function first, before defining the node removal function.

Here is the implementation of the removeEdge function :

```
func (g *Graph) removeEdge(node, neighbor *Node) { index := -1 for i, n := range
```

The removeEdge function accepts two nodes as parameters and finds the index of the second node in the neighbor list of the primary node. It then proceeds to remove neighbors from node.Neighbors using the name slicing the slice technique .

Elimination works by taking the slice elements (but not including) up to the specific index, slice elements from after the specified index, and concatenating them. Leave the specified element index.

In this case, you have an undirected graph, so its edge is two-dimensional. This is why you must call removeEdge twice in the main RemoveEdge function to remove the neighbor from the node's list and vice versa.

Remove a node from the graph

Here is the function to remove the node from the graph:

```
func (g *Graph) RemoveNode(nodeID int) { node, exists := g.nodes[nodeID] if !exis
```

This function accepts the ID of the node you need to remove. It checks to see if the node exists before continuing to remove all its edges. It then deletes the node from the graph using the delete function available in Go.

Above is how to create a chart data structure in Go. Hope the article is useful to you.

You finished reading the article "**How to implement a graph data structure in Golang**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com