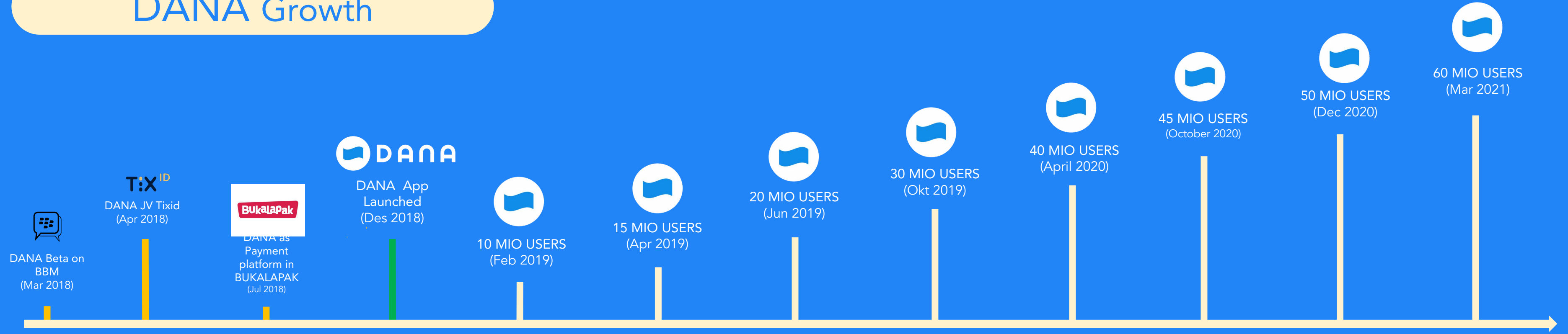
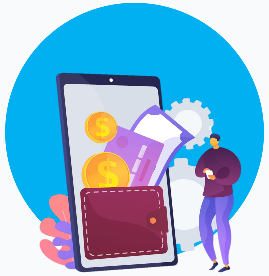
#DaysInDANA

DANA Growth



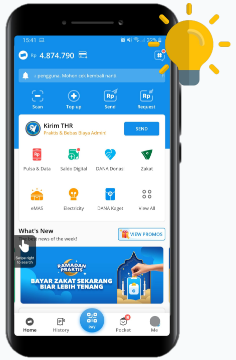
720 DANAM8s, with over 60% *engineers* among us.

More than 60 MILLION users.



3 MILLION transactions per day.

New DANA Features



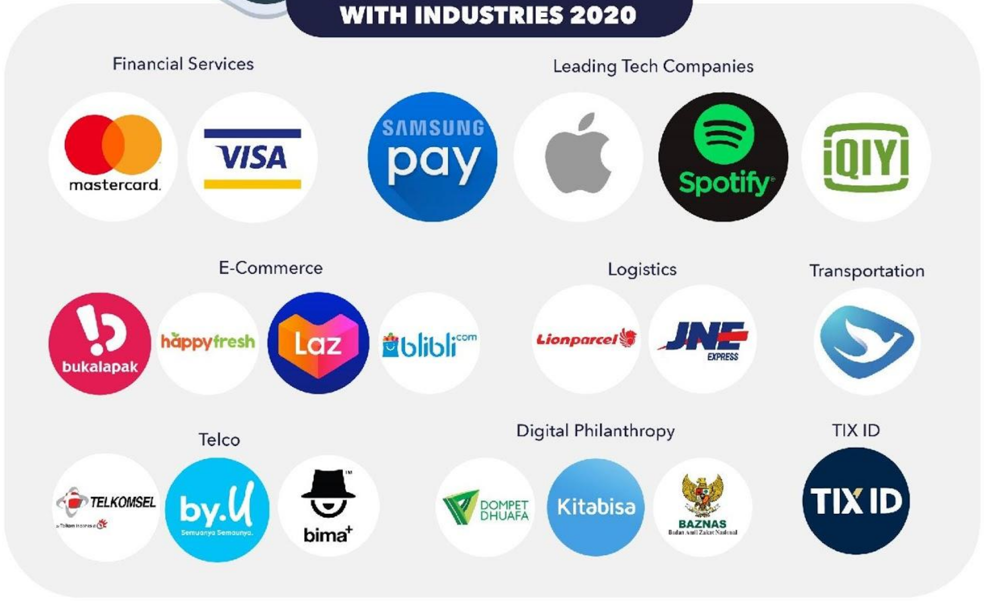
- QRIS 100%
- Integrated with Apple
- Integrated with Lazada
- Integrated with Secure Parking
- DANA Protection
- Card Binding
- P2P Transfer
- Biller Reminder

DANA ONLINE PARTNERS

DANA has partnered with >1.900 ONLINE MERCHANTS

And more are coming

DANA NEW PARTNERSHIP WITH INDUSTRIES 2020



TOP 5 MOST USED FEATURES

1 QRIS PAYMENT

2 ONLINE MERCHANTS

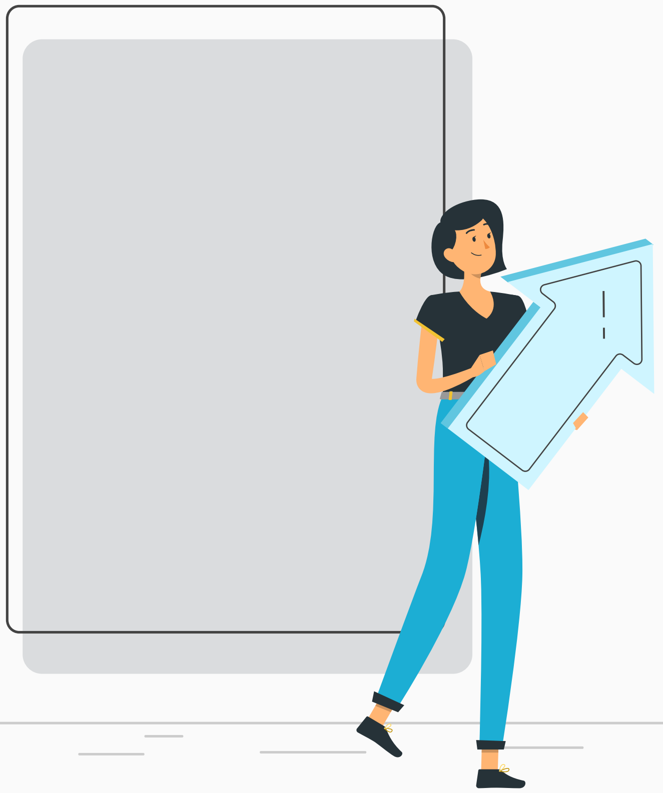
3 SEND MONEY

4 BILLER (Water, Electricity, etc.)

5 DANA BISNIS

DATA STRUCTURE IN GOLANG

#BicaraDANA BINUS University



01 STACK & QUEUE

02 LINKED LIST

03 HEAPS

04 HASH

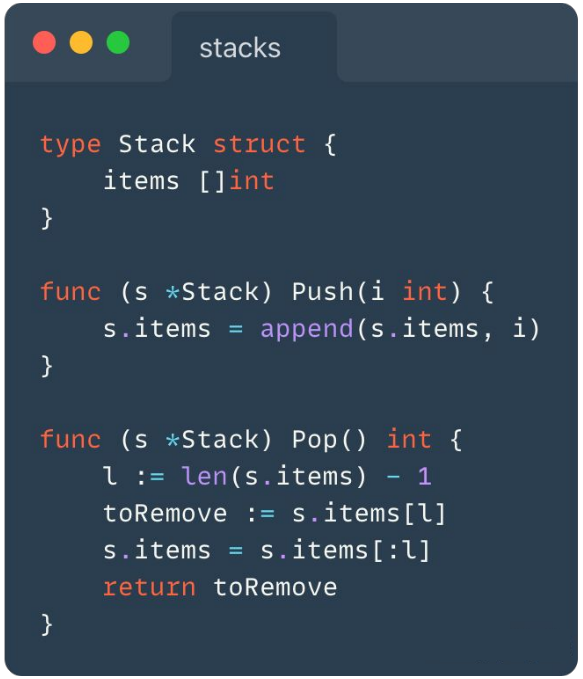
05 GRAPH

06 CASE STUDY

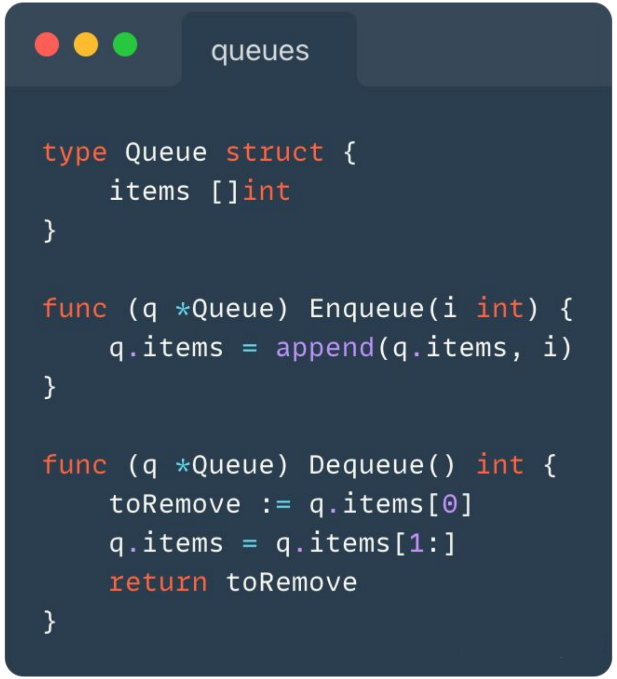
01 STACK & QUEUE

- Linear data structures
- Flexible sizes
- The main difference between Stacks and Queues is the way data is removed:
    - Stacks use LIFO
    - Queues use FIFO

01 STACK & QUEUE


stacks

```go
type Stack struct {
    items []int
}

func (s *Stack) Push(i int) {
    s.items = append(s.items, i)
}

func (s *Stack) Pop() int {
    l := len(s.items) - 1
    toRemove := s.items[l]
    s.items = s.items[:l]
    return toRemove
}
```

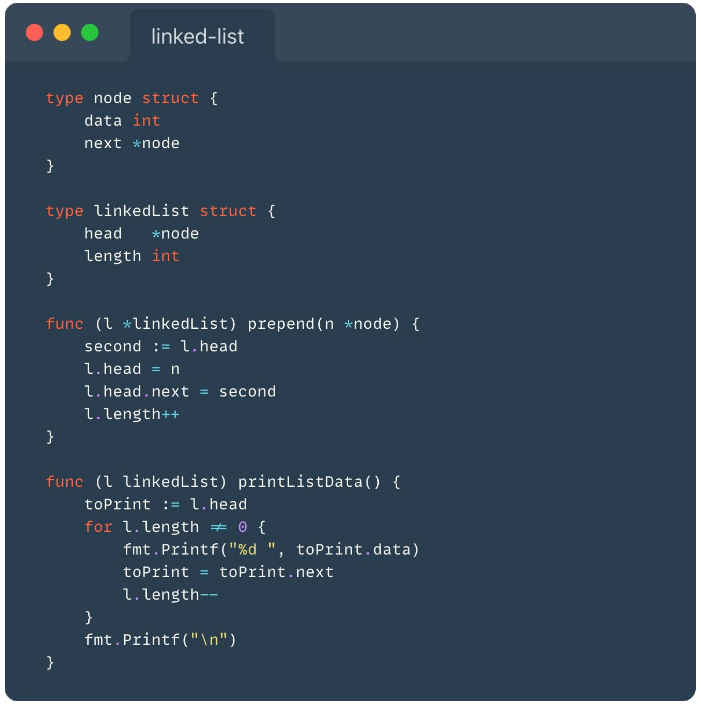

queues

```go
type Queue struct {
    items []int
}

func (q *Queue) Enqueue(i int) {
    q.items = append(q.items, i)
}

func (q *Queue) Dequeue() int {
    toRemove := q.items[0]
    q.items = q.items[1:]
    return toRemove
}
```

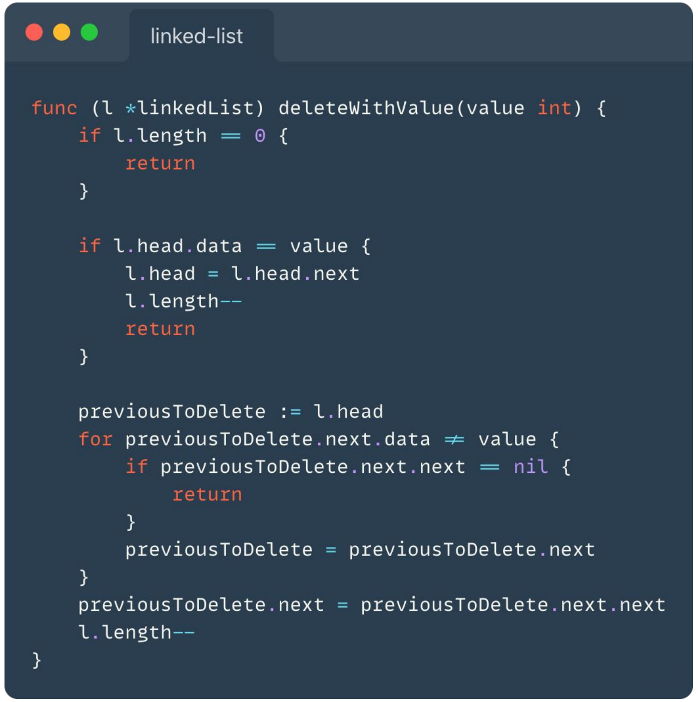
02 LINKED LIST

- Linear data structures
- A linked list is a *sequential access* data structure, where each element can be accessed only in particular order
- Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node
- The last node has a reference to **null**
- The entry point into a linked list is called the **head** of the list
- If the list is empty then the head is a null reference
- Different kinds of linked list:
  - **Singly** linked list
  - **Doubly** linked list

02 LINKED LIST



```go
type node struct {
    data int
    next *node
}

type linkedList struct {
    head   *node
    length int
}

func (l *linkedList) prepend(n *node) {
    second := l.head
    l.head = n
    l.head.next = second
    l.length++
}

func (l linkedList) printListData() {
    toPrint := l.head
    for l.length != 0 {
        fmt.Printf("%d ", toPrint.data)
        toPrint = toPrint.next
        l.length--
    }
    fmt.Printf("\n")
}
```

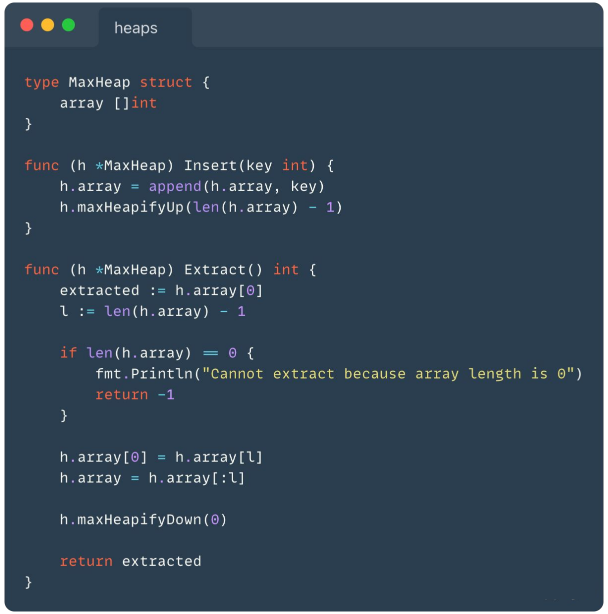


```go
func (l *linkedList) deleteWithValue(value int) {
    if l.length == 0 {
        return
    }

    if l.head.data == value {
        l.head = l.head.next
        l.length--
        return
    }

    previousToDelete := l.head
    for previousToDelete.next.data != value {
        if previousToDelete.next.next == nil {
            return
        }
        previousToDelete = previousToDelete.next
    }
    previousToDelete.next = previousToDelete.next.next
    l.length--
}
```

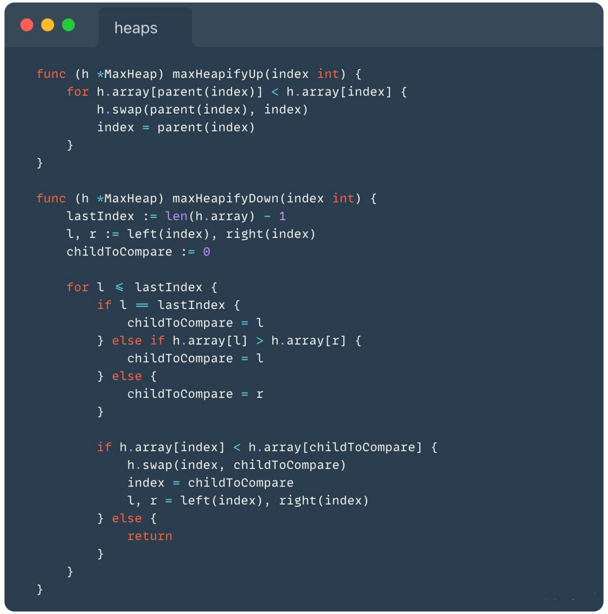
03 HEAPS

- Heap can be expressed as a complete tree which satisfies the heap ordering property, means all the level in the tree are full, except the lowest level
- The ordering can be one of two types:
  - the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root
  - the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root
- In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap”
- Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height
- Can be calculate as:
  - [1] x 2 + 1 = [3] ß get left child index
  - [1] x 2 + 2 = [4] ß get right child index
  - ([1]-1)/2 = ß get parent index

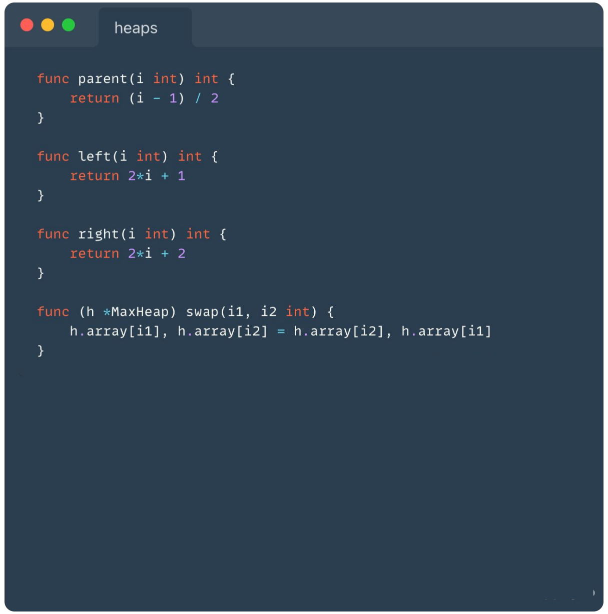
03 HEAPS


heaps

```go
type MaxHeap struct {
    array []int
}

func (h *MaxHeap) Insert(key int) {
    h.array = append(h.array, key)
    h.maxHeapifyUp(len(h.array) - 1)
}

func (h *MaxHeap) Extract() int {
    extracted := h.array[0]
    l := len(h.array) - 1

    if len(h.array) == 0 {
        fmt.Println("Cannot extract because array length is 0")
        return -1
    }

    h.array[0] = h.array[l]
    h.array = h.array[:l]

    h.maxHeapifyDown(0)

    return extracted
}
```


heaps

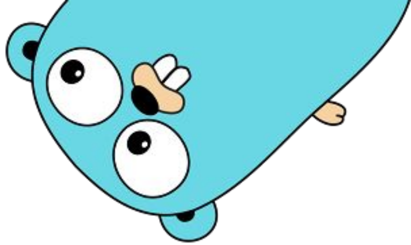
```go
func (h *MaxHeap) maxHeapifyUp(index int) {
    for h.array[parent(index)] < h.array[index] {
        h.swap(parent(index), index)
        index = parent(index)
    }
}

func (h *MaxHeap) maxHeapifyDown(index int) {
    lastIndex := len(h.array) - 1
    l, r := left(index), right(index)
    childToCompare := 0

    for l <= lastIndex {
        if l == lastIndex {
            childToCompare = l
        } else if h.array[l] > h.array[r] {
            childToCompare = l
        } else {
            childToCompare = r
        }

        if h.array[index] < h.array[childToCompare] {
            h.swap(index, childToCompare)
            index = childToCompare
            l, r = left(index), right(index)
        } else {
            return
        }
    }
}
```


heaps

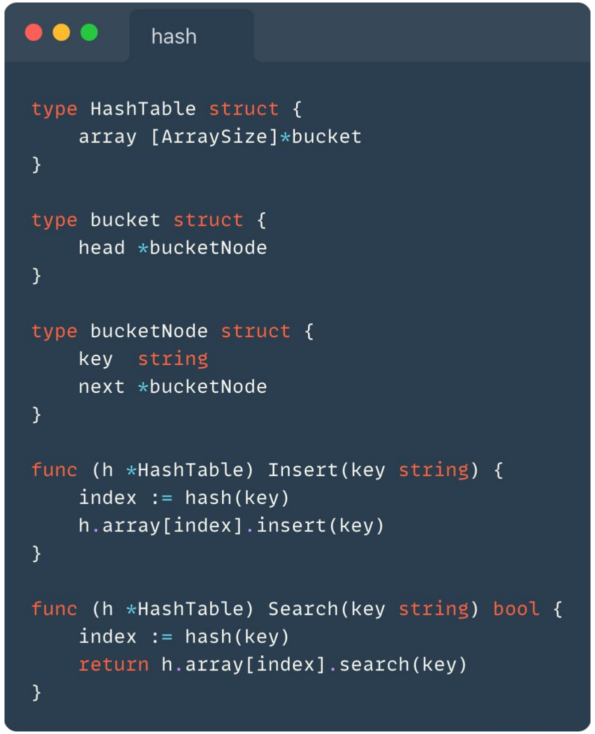
```go
func parent(i int) int {
    return (i - 1) / 2
}

func left(i int) int {
    return 2*i + 1
}

func right(i int) int {
    return 2*i + 2
}

func (h *MaxHeap) swap(i1, i2 int) {
    h.array[i1], h.array[i2] = h.array[i2], h.array[i1]
}
```

04 HASH

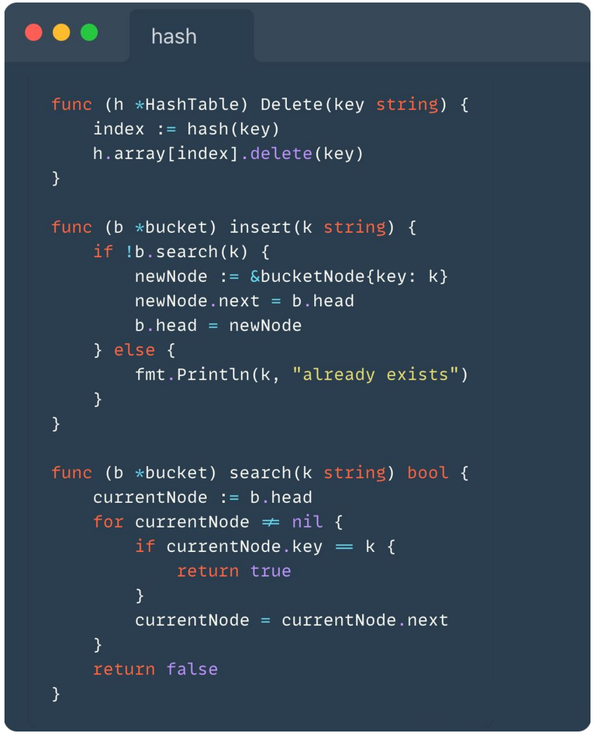
- The problem at hands is to speed up searching
- A hash function that returns a unique hash number is called a **universal hash function**.
- Key => value lookup
- Each letter/character converted to ASCII code
- Collision handling methods:
  - Open addressing: if the index has already taken, then store the name in the next index (bad method!)
  - Separate chaining: storing multiple name in one index by using linked list (good method!)
- Playground: https://www.cs.usfca.edu/~galles/visualization/OpenHash.html



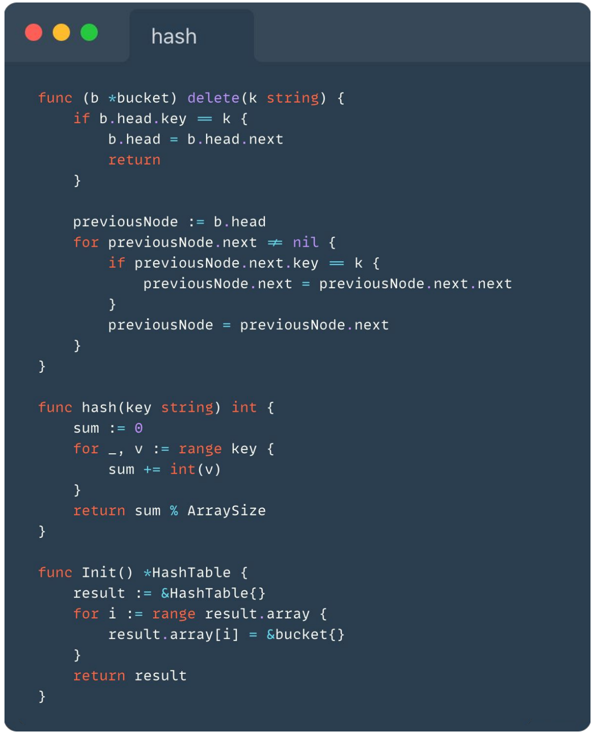
04 HASH



```go
type HashTable struct {
    array [ArraySize]*bucket
}

type bucket struct {
    head *bucketNode
}

type bucketNode struct {
    key  string
    next *bucketNode
}

func (h *HashTable) Insert(key string) {
    index := hash(key)
    h.array[index].insert(key)
}

func (h *HashTable) Search(key string) bool {
    index := hash(key)
    return h.array[index].search(key)
}
```



```go
func (h *HashTable) Delete(key string) {
    index := hash(key)
    h.array[index].delete(key)
}

func (b *bucket) insert(k string) {
    if !b.search(k) {
        newNode := &bucketNode{key: k}
        newNode.next = b.head
        b.head = newNode
    } else {
        fmt.Println(k, "already exists")
    }
}

func (b *bucket) search(k string) bool {
    currentNode := b.head
    for currentNode != nil {
        if currentNode.key == k {
            return true
        }
        currentNode = currentNode.next
    }
    return false
}
```

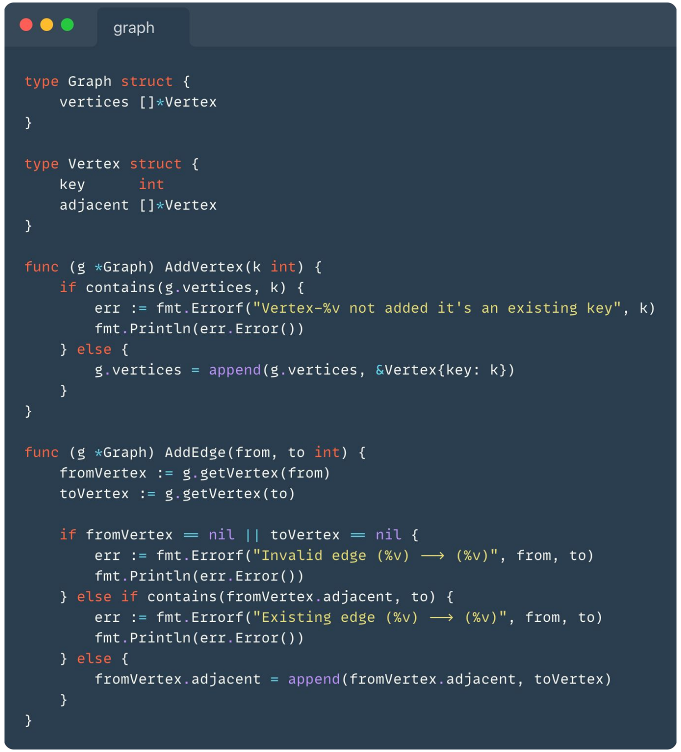


```go
func (b *bucket) delete(k string) {
    if b.head.key == k {
        b.head = b.head.next
        return
    }

    previousNode := b.head
    for previousNode.next != nil {
        if previousNode.next.key == k {
            previousNode.next = previousNode.next.next
        }
        previousNode = previousNode.next
    }
}

func hash(key string) int {
    sum := 0
    for _, v := range key {
        sum += int(v)
    }
    return sum % ArraySize
}

func Init() *HashTable {
    result := &HashTable{}
    for i := range result.array {
        result.array[i] = &bucket{}
    }
    return result
}
```

05 GRAPH

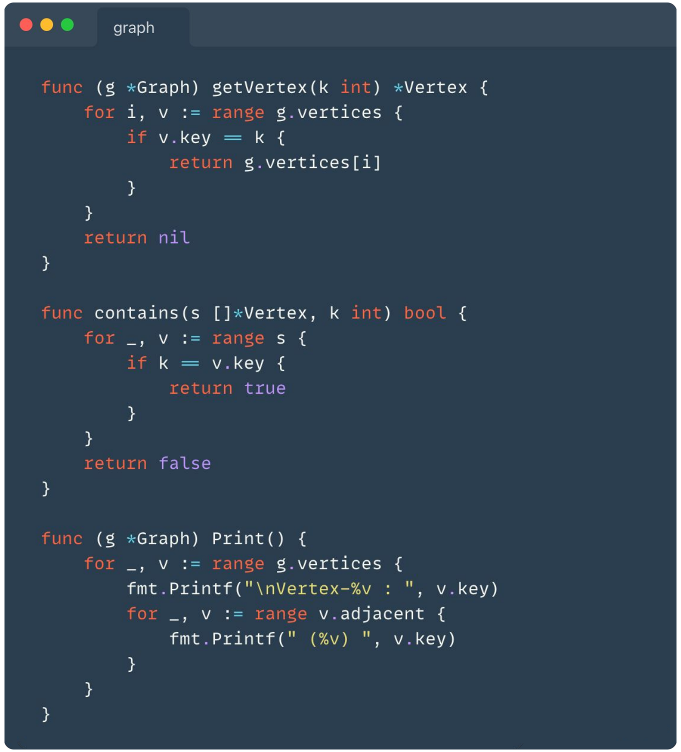
- Set of objects where some pairs of objects are connected by links
- The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**
- Different kinds of way to store a graph:
  - Adjacency List
  - Adjacency Matrix
- Different kinds of graph representations:
  - Undirected graph
  - Directed graph
    - Cyclic or Acyclic
    - Weighted or Unweighted

05 GRAPH



graph

```go
type Graph struct {
    vertices []*Vertex
}

type Vertex struct {
    key      int
    adjacent []*Vertex
}

func (g *Graph) AddVertex(k int) {
    if contains(g.vertices, k) {
        err := fmt.Errorf("Vertex-%v not added it's an existing key", k)
        fmt.Println(err.Error())
    } else {
        g.vertices = append(g.vertices, &Vertex{key: k})
    }
}

func (g *Graph) AddEdge(from, to int) {
    fromVertex := g.getVertex(from)
    toVertex := g.getVertex(to)

    if fromVertex == nil || toVertex == nil {
        err := fmt.Errorf("Invalid edge (%v) --> (%v)", from, to)
        fmt.Println(err.Error())
    } else if contains(fromVertex.adjacent, to) {
        err := fmt.Errorf("Existing edge (%v) --> (%v)", from, to)
        fmt.Println(err.Error())
    } else {
        fromVertex.adjacent = append(fromVertex.adjacent, toVertex)
    }
}
```



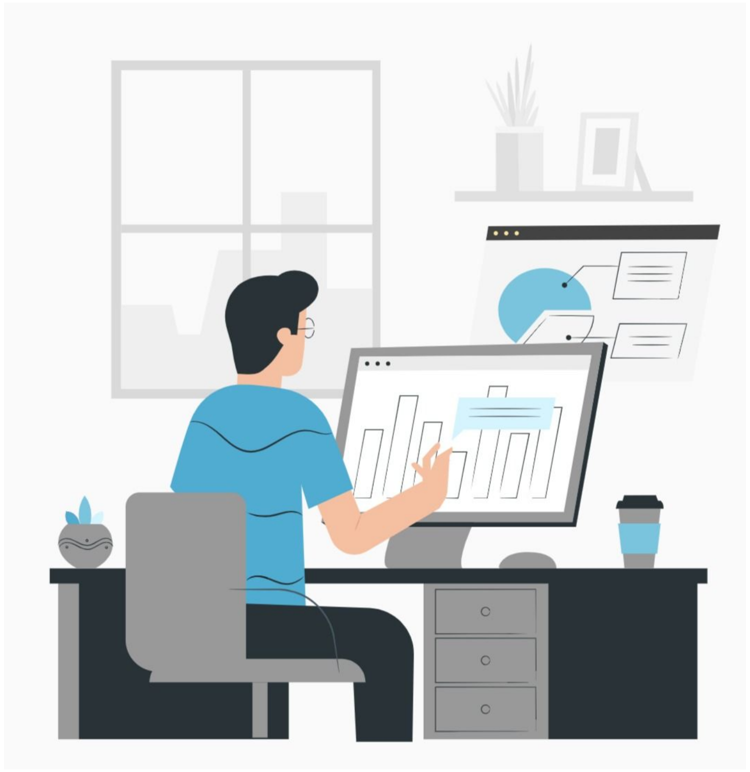
graph

```go
func (g *Graph) getVertex(k int) *Vertex {
    for i, v := range g.vertices {
        if v.key == k {
            return g.vertices[i]
        }
    }
    return nil
}

func contains(s []*Vertex, k int) bool {
    for _, v := range s {
        if k == v.key {
            return true
        }
    }
    return false
}

func (g *Graph) Print() {
    for _, v := range g.vertices {
        fmt.Printf("\nVertex-%v : ", v.key)
        for _, v := range v.adjacent {
            fmt.Printf(" (%v) ", v.key)
        }
    }
}
```

06 CASE STUDY

(DAG) Directed Acyclic Graph on Apache Airflow

07 Q & A



THANK YOU!