目　录

快速排序算法

快速排序算法

算法描述：是对插入算法的一种优化，利用对问题的二分化，实现递归完成快速排序，在所有算法中二分化是最常用的方式，将问题尽量的分成两种情况加以分析，最终以形成类似树的方式加以利用，因为在比较模型中的算法中，最快的排序时间负载度为 **O(nlgn)**.

算法步骤

- 将数据根据一个值按照大小分成左右两边，左边小于此值，右边大于
- 将两边数据进行递归调用步骤1
- 将所有数据合并

优化

标准的快速排序只是取，数组的第一个元素进行分组，对于随机的数据会产生如果数据已经排好序，算法的时间会上升到O(n^2),所以一般为了优化，避免产生最坏情况，会采取随机数来避免这种情况产生

优化步骤

- 随机选取一个数，当分割值
- 递归1方法
- 将所有数据进行合并

```
package sort

import "fmt"

func QuickSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }
    splitdata := arr[0]           //第一个数据
    low := make([]int, 0, 0)      //比我小的数据
    hight := make([]int, 0, 0)    //比我大的数据
    mid := make([]int, 0, 0)      //与我一样大的数据
    mid = append(mid, splitdata) //加入一个
```

```
    for i := 1; i < len(arr); i++ {
        if arr[i] < splitdata {
            low = append(low, arr[i])
        } else if arr[i] > splitdata {
            hight = append(hight, arr[i])
        } else {
            mid = append(mid, arr[i])
        }
    }
    low, hight = QuickSort(low), QuickSort(hight)
    myarr := append(append(low, mid...), hight...)
    return myarr
}

//快读排序算法
func main() {
    arr := []int{1, 9, 10, 30, 2, 5, 45, 8, 63, 234, 12}
    fmt.Println(QuickSort(arr))
}
```

堆排序算法

堆排序算法

算法描述：首先建一个堆，然后调整堆，调整过程是将节点和子节点进行比较，将其中最大的值变为父节点，递归调整调整次数lgn,最后将根节点和尾节点交换再n次调整**O(nlgn)**.

算法步骤

- 创建最大堆或者最小堆（我是最小堆）
- 调整堆
- 交换首尾节点(为了维持一个完全二叉树才要进行收尾交换)

```
package sort

import "fmt"

//堆排序
func main() {
    arr := []int{1, 9, 10, 30, 2, 5, 45, 8, 63, 234, 12}
    fmt.Println(HeapSort(arr))
}
func HeapSortMax(arr []int, length int) []int {
    // length := len(arr)
    if length <= 1 {
        return arr
    }
    depth := length/2 - 1 //二叉树深度
    for i := depth; i >= 0; i-- {
        topmax := i //假定最大的位置就在i的位置
        leftchild := 2*i + 1
        rightchild := 2*i + 2
        if leftchild <= length-1 && arr[leftchild] > arr[topmax] { //防止越过界
限
            topmax = leftchild
        }
        if rightchild <= length-1 && arr[rightchild] > arr[topmax] { //防止越过
界限
            topmax = rightchild
        }
        if topmax != i {
```

```
            arr[i], arr[topmax] = arr[topmax], arr[i]
        }
    }
    return arr
}
func HeapSort(arr []int) []int {
    length := len(arr)
    for i := 0; i < length; i++ {
        lastlen := length - i
        HeapSortMax(arr, lastlen)
        if i < length {
            arr[0], arr[lastlen-1] = arr[lastlen-1], arr[0]
        }
    }
    return arr
}
```

冒泡排序算法

冒泡排序算法

算法描述：冒泡算法，数组中前一个元素和后一个元素进行比较如果大于或者小于前者就进行交换，最终返回最大或者最小都冒到数组的最后序列时间复杂度为**O(n^2)**.

算法步骤

- 从数组开头选择相邻两个元素进行比较，并进行交换
- 不停向后移动

```
package sort

import "fmt"

//冒泡排序
func main() {
    arr := []int{1, 9, 10, 30, 2, 5, 45, 8, 63, 234, 12}
    fmt.Println(GetMax(arr))
    fmt.Println(BubbleSort(arr))
}

//冒泡排序获取最大值
func GetMax(arr []int) int {
    for j := 1; j < len(arr); j++ {
        if arr[j-1] > arr[j] {
            arr[j-1], arr[j] = arr[j], arr[j-1]
        }
    }
    return arr[len(arr)-1]
}

//冒泡排序
func BubbleSort(arr []int) []int {
    for i := 0; i < len(arr); i++ {
        for j := i + 1; j < len(arr); j++ {
            if arr[i] > arr[j] {
                arr[i], arr[j] = arr[j], arr[i]
            }
        }
```

```
    }
    return arr
}
```

二分查找方法

二分查找方法

算法描述：在一组有序数组中，将数组一分为二，将要查询的元素和分割点进行比较，分为三种情况

- 相等直接返回
- 元素大于分割点，在分割点右侧继续查找
- 元素小于分割点，在分割点左侧继续查找

时间复杂： **O(lgn)**.

要求

- 必须是有序的数组，并能支持随机访问

变形

- 查找第一个值等于给定的
  - 在相等的时候做处理，向前查
- 查找最后一个值等于给定的值
  - 在相等的时候做处理，向后查
- 查找第一个大于等于给定的值
  - 判断边界减1
- 查找最后一个小于等于给定的值
  - 判断边界加1

实际应用

- 用户ip区间段查询
- 用于相似度查询

```
package sort

import "fmt"

func bin_search(arr []int, finddata int) int {
```

```
    low := 0
    high := len(arr) - 1
    for low <= high {
        mid := (low + high) / 2
        fmt.Println(mid)
        if arr[mid] > finddata {
            high = mid - 1
        } else if arr[mid] < finddata {
            low = mid + 1
        } else {
            return mid
        }
    }
    return -1
}

func main() {
    arr := make([]int, 1024*1024, 1024*1024)
    for i := 0; i < 1024*1024; i++ {
        arr[i] = i + 1
    }
    id := bin_search(arr, 1024)
    if id != -1 {
        fmt.Println(id, arr[id])
    } else {
        fmt.Println("没有找到数据")
    }
}
```

选择排序算法

选择排序算法

算法描述：从未排序数据中选择最大或者最小的值和当前值交换 **O(n^2)**.

算法步骤

- 选择一个数当最小值或者最大值，进行比较然后交换
- 循环向后查进行

```
package sort

import "fmt"

//获取切片里面的最大值
func SelectMax(arr []int) int {
    length := len(arr)
    if length <= 1 {
        return arr[0]
    }
    max := arr[0]
    for i := 1; i < length; i++ {
        if arr[i] > max {
            max = arr[i]
        }
    }
    return max
}

//切片排序
func SelectSort(arr []int) []int {
    length := len(arr)
    if length <= 1 {
        return arr
    }
    for i := 1; i < length; i++ {
        min := i
        for j := i + 1; j < length; j++ {
            if arr[min] > arr[j] {
                min = j
            }
```

```
        }
        if i != min {
            arr[i], arr[min] = arr[min], arr[i]
        }
    }
    return arr
}

//选择排序
func main() {
    arr := []int{1, 9, 10, 30, 2, 5, 45, 8, 63, 234, 12}
    max := SelectMax(arr)
    selectsort := SelectSort(arr)
    fmt.Println(max)
    fmt.Println(selectsort)
}
```

基数排序算法

基数排序算法

算法描述：基数排序类似计数排序，需要额外的空间来记录对应的基数内的数据额外的空间是有序的，最终时间复杂度**O(nlogrm)**,r是基数，r^m=n.当给定特定的范围，计数排序又可以叫桶排序，当以10进制为基数时就是简单的桶排序

算法步骤

- 从个位开始排序，从低到高进行递推
- 比较过程中如果遇到高位相同时，顺序不变

算法分两类

1. 低位排序LSD
2. 高位排序MSD

```
package sort

import "fmt"

func main() {
    var arr [3][]int
    myarr := []int{1, 2, 3, 1, 1, 2, 2, 2, 2, 2, 3}
    for i := 0; i < len(myarr); i++ {
        arr[myarr[i]-1] = append(arr[myarr[i]-1], myarr[i])
    }
    fmt.Println(arr)
}
```

拓扑排序

拓扑排序

定义

对于一些有前后依赖关系的排序算法，是利用有向无环图进行实现，通过局部依赖关系确定全局顺序的算法

应用场景

- 编译有序依赖的文件

两种拓扑算法

Kahn算法

- 算法逻辑

1. 利用贪心算法，如果两个顶点，顶点b依赖于顶点a,就将a指向b,当一个顶点的入度为零，将这个顶点就是最优排序点，
   并且将顶点从图中移除，将可达顶点的入度减一。

DFS算法

1.使用深度算法，产生逆向邻接表先输出其他依赖，最后输出自己。

```
package main

import (
    "fmt"
)

//有向图
type graph struct {
    vertex int           //顶点
    list   map[int][]int //连接表边
}

//添加边
func (g *graph) addVertex(t int, s int) {
    g.list[t] = push(g.list[t], s)
```

```
}

func main() {
    g := NewGraph(8)
    g.addVertex(2, 1)
    g.addVertex(3, 1)
    g.addVertex(7, 1)
    g.addVertex(4, 2)
    g.addVertex(5, 2)
    g.addVertex(8, 7)
    g.DfsSort()
}

//创建图
func NewGraph(v int) *graph {
    g := new(graph)
    g.vertex = v
    g.list = map[int][]int{}
    i := 0
    for i < v {
        g.list[i] = make([]int, 0)
        i++
    }
    return g
}

//取出切片第一个
func pop(list []int) (int, []int) {
    if len(list) > 0 {
        a := list[0]
        b := list[1:]
        return a, b
    } else {
        return -1, list
    }
}

//推入切片
func push(list []int, value int) []int {
    result := append(list, value)
    return result
}

//添加边
func (g *graph) KhanSort() {
    var inDegree = make(map[int]int)
```

```
    var queue []int
    for i := 1; i <= g.vertex; i++ {
        for _, m := range g.list[i] {
            inDegree[m]++
        }
    }
    for i := 1; i <= g.vertex; i++ {

        if inDegree[i] == 0 {
            queue = push(queue, i)
        }
    }
    for len(queue) > 0 {
        var now int
        now, queue = pop(queue)
        fmt.Println("->", now)
        for _, k := range g.list[now] {
            inDegree[k]--
            if inDegree[k] == 0 {
                queue = push(queue, k)
            }
        }
    }
}

func (g *graph) DfsSort() {
    inverseList := make(map[int][]int)
    //初始化逆向邻接表
    for i := 1; i <= g.vertex; i++ {
        for _, k := range g.list[i] {
            inverseList[k] = append(inverseList[k], i)
        }
    }
    visited := make([]bool, g.vertex+1)
    visited[0] = true
    for i := 1; i <= g.vertex; i++ {
        if visited[i] == false {
            visited[i] = true
            dfs(i, inverseList, visited)
        }
    }

}

func dfs(vertex int, inverseList map[int][]int, visited []bool) {
    for _, w := range inverseList[vertex] {
```

```
        if visited[w] == true {
            continue
        } else {
            visited[w] = true
            dfs(w, inverseList, visited)
        }
    }
    fmt.Println("->", vertex)
}
```

插入排序算法

插入排序算法

算法描述：插入算法，从第一个数开始进行循环，插入到一个已经排序的数组中循环遍历所有元素，最终返回所有元素的排好的序列时间复杂度为 **O(n^2)**.

算法步骤

- 选择一个数进行比较然后将比这个值小的元素插入这个值之前
- 循环向后查进行

错误总结

- 插入算法一定和冒泡算法区分开
- 插入算法是将需要的元素插入到当前元素之前
- 冒泡是两两交换将想要的元素置顶

```
package main

import (
    "fmt"
)

func main() {
    arrList := []int{1, 2, 8, 11, 3, 6, 8, 4, 9, 343, 3}
    arrList = standardInsertSort(arrList)
    fmt.Println(arrList)
}

func standardInsertSort(list []int) []int {
    resultList := []int{}
    length := len(list)
    i := 1
    resultList = append(resultList, list[0])
    for i < length {
        for j := 0; j < len(resultList); j++ {
            if list[i] <= resultList[j] {
                resultList = insertList(resultList, j, list[i])
                break
            }
```

```
            if j == len(resultList)-1 && list[i] > resultList[j] {
                resultList = insertList(resultList, j+1, list[i])
                break
            }
        }
        i++
    }
    return resultList
}
func insertList(list []int, i int, x int) []int {
    returnList := []int{}
    n := 0
    if i == len(list) {
        returnList = append(list, x)
        return returnList
    }
    for n < len(list) {
        if n < i {
            returnList = append(returnList, list[n])
        } else if n == i {
            returnList = append(returnList, x)
            returnList = append(returnList, list[n])
        } else {
            returnList = append(returnList, list[n])
        }
        n++
    }
    return returnList
}
```

字符串匹配

字符串匹配

BF(Brute force)算法

实现：每次向后移动一位进行匹配

RK(Rabin-Karp)算法

实现：将每组要匹配长度的字符串进行hash，再hash后的元素里找

BM(Boyer-Moore)算法

有两部分组成：并且是由大到小，倒着匹配

1. 坏前缀：普通匹配只一位一位移动，移动规则为 si(坏字符的位置) xi(坏字符在匹配字符最后出现的位置) 都没有xi=-1
   移动距离等于si-xi
2. 好后缀：坏前缀有可能产生负数，所以还要利用好后缀来进行匹配，好后缀类似坏前缀如果匹配串中有和好后缀相同的子串
   ，移动到最靠后的子串的位置，如果没有相同的子串，就需要在匹配的子串中，查找和前缀子串匹配最长的子串进行移动。

KMP（Knuth Morris Pratt）算法

实现：关键部分next数组，失效函数。next数据就是匹配串字符串最长匹配前缀和最长匹配后缀的关系。

```
package main

import "fmt"

func main() {
    a := "ababaeabacaaaaaddfdfdfdfdf"
    b := "aca"
    pos := Kmp(b, a)
    fmt.Println(pos)
}
func Kmp(needle string, str string) int {
    next := getNext(needle)
    fmt.Println(next)
    j := 0
```

```
    for i := 0; i < len(str); i++ {
        for j > 0 && str[i] != needle[j] {
            j = next[j-1] + 1
        }
        if str[i] == needle[j] {
            j++
        }
        if j == len(needle) {
            return i - j + 1
        }
    }
    return -1
}

func getNext(needle string) []int {
    var next = make([]int, len(needle))
    fmt.Println(next)
    next[0] = -1
    k := -1
    for i := 1; i < len(needle); i++ {
        for k != -1 && needle[k+1] != needle[i] {
            k = next[k]
        }
        if needle[k+1] == needle[i] {
            k++
        }
        next[i] = k
    }
    return next
}
```

二叉搜索树

二叉搜索树

算法描述：二叉搜索树，是按照一个节点的左子叶小于节点的值，右子叶大于节点的值搜索二叉树的，插入和搜索时间复杂都是**O(lgn)**.

算法步骤

- 选取根节点
- 按照算法描述构建树

算法分析

- 如果想树的搜索深度尽量浅，就改把中间值作为根节点,数据尽量也不要是有序的否则会形成单边树，或者就是一个链表，修改和查找的时间复杂都变为**O(n)**

B+树在数据库中的应用

思路总结

1. 解决问题前先定义清楚问题
    - 对问题进行假设，限定解决问题的范围
    - select from a where id=1 主键搜索
    - select from a where id>19 范围搜索
    - 非功能性需求
        - 安全
        - 性能
        - 用户体验
    - 由于我们想学习算法和数据结构，所以我们只从性能上进行讨论
        - 执行效率
        - 存储空间
2. 尝试用学习过的数据结构解决问题
    - 散列表：
        - 查询效率O(1)但是不支持范围查询
    - 平衡二叉树：

        - 可以快速查询O(lgn)，中序遍历是有序数组单不支持范围查询
    - 跳表
        - 查询数据O(lgn)，并可以支持范围查询
        - 虽然满足条件但有更好的方法，减少内存使用
3. 优化
    - 使用跳表虽然可以实现需求，但是我们有更好的数据结构，二叉查找树。二叉查找树和跳表类似查询速度O(lgn)
    - 遇到问题：内存消耗巨大
        - 如果数据量巨大的二叉树，比如1亿条数据，就要有1亿个节点，如果一个节点占用16字节，那占用内存1g,10个表就是10gb
    - 解决问题：为了减少内存消耗，只能通过将数据放在硬盘中
    - 遇到分体：读取磁盘中的节点IO消耗很大
    - 解决问题：所以尽量降低树的高度，减少io操作
    - 遇到问题：如何降低树的高度
    - 解决问题：从二叉树，变为N叉树
    - 遇到问题：是不是N越大越好呢?
    - 解决问题：cup的缓存是以也为单位的，如果数据超过一页，需要分页处理所以为了尽量减少io操作，节点数据尽量不要超过一页（4KB）

结论

综上所述：使用某种算法是要全面考虑到实际的问题，根据实际的问题进行假设，选择合适的算法，结合软硬件效率综合得出一个合理方案。

```
package main

import (
    "fmt"
    "math"
)

type Node struct {
    value  int
    left   *Node
    right  *Node
    parent *Node
}

type Tree struct {
```

```
    root   *Node
    length int
}

func main() {
    creatTree()
}

func creatTree() {
    arrList := []int{14, 2, 5, 7, 23, 35, 12, 17, 31}
    myTree := Tree{}
    for i := 0; i < len(arrList); i++ {
        myTree = insertNode(myTree, arrList[i])
        myTree.length++
    }
    fmt.Println(myTree)
    //LDR(myTree)
    TreeHeight(myTree)
}
func TreeHeight(tree Tree) {
    var hl = 1
    if tree.root.left != nil {
        hl = heightMax(tree.root.left, hl)
    }
    var hr = 1
    if tree.root.right != nil {
        hr = heightMax(tree.root.left, hr)
    }
    fmt.Println(hl, hr)
    fmt.Println("Tree height is ", int(math.Max(float64(hl), float64(hr))))
}

func heightMax(node *Node, h int) int {
    var hL = h
    var hR = h
    if node.left == nil && node.right == nil {
        fmt.Println(node)
        return h
    }
    if node.left != nil {
        h++
        hL = heightMax(node.left, h)
    }
    if node.right != nil {
        h++
        hR = heightMax(node.right, h)
```

```
    }
    return int(math.Max(float64(hL), float64(hR)))
}

//中序遍历
func LDR(tree Tree) {
    readList := make(map[int]int)
    i := 0
    var currentNode *Node
    currentNode = tree.root
    for {
        //fmt.Println(currentNode)
        if i == tree.length {
            //fmt.Println(currentNode.value)
            break
        }
        if currentNode.left == nil {
            if readList[currentNode.value] == 1 {
                if readList[currentNode.right.value] == 1 {
                    currentNode = currentNode.parent
                    continue
                } else {
                    currentNode = currentNode.right
                    continue
                }
            } else {
                fmt.Println(currentNode.value)
                readList[currentNode.value] = 1
                i++
                if currentNode.right == nil {
                    currentNode = currentNode.parent
                    continue
                } else {
                    if readList[currentNode.right.value] == 1 {
                        currentNode = currentNode.parent
                        continue
                    } else {
                        currentNode = currentNode.right
                        continue
                    }
                }
            }
        } else {
            if readList[currentNode.left.value] == 1 {
                if readList[currentNode.value] == 1 {
                    currentNode = currentNode.right
```

```
                    continue
                } else {
                    fmt.Println(currentNode.value)
                    readList[currentNode.value] = 1
                    i++
                    if currentNode.right == nil {
                        currentNode = currentNode.parent
                        continue
                    } else {
                        if readList[currentNode.right.value] == 1 {
                            currentNode = currentNode.parent
                            continue
                        } else {
                            currentNode = currentNode.right
                            continue
                        }

                    }
                }
            } else {
                currentNode = currentNode.left
                continue
            }

        }

    }
}

func insertNode(tree Tree, insertValue int) Tree {
    var currentNode *Node
    var tmp *Node
    i := 0
    if tree.length == 0 {
        currentNode = new(Node)
        currentNode.value = insertValue
        tree.root = currentNode
        return tree
    } else {
        currentNode = tree.root
    }
    for {
        //fmt.Println(currentNode)
        if currentNode.value < insertValue {
            //判断是否有右节点
            if currentNode.right == nil {
```

```
                tmp = new(Node)
                tmp.value = insertValue
                currentNode.right = tmp
                tmp.parent = currentNode
                break
            } else {
                currentNode = currentNode.right
                continue
            }
        } else {
            if currentNode.left == nil {
                tmp = new(Node)
                tmp.value = insertValue
                currentNode.left = tmp
                tmp.parent = currentNode
                break
            } else {
                currentNode = currentNode.left
                continue
            }
        }
        i++
    }
    return tree
}
```

图

图

- 相对于树来说更复杂的非线性数据结构

图分类

- 无向图
  - 表示顶点到顶点没有方向
- 有向图
  - 表示顶点到顶点有方向
    - 入度：有多少个顶点进入
    - 出度：有多少个顶点出去
    - 六度分割理论
- 带权图
  - 表示顶点到顶点有权重

图的存储方式

- 邻接矩阵：
  - 缺点：对于无向图来说，用邻接矩阵存储是浪费空间的，因为邻接矩阵是对称的，并且对于大量顶点需要跟多的空间来存储
  - 优点：直观，并且可以利用矩阵计算，求最短路径
- 邻接表：
- 如何实现微博的关注关系
  - 用邻接表来实现用户的关注的人
    - 类似链表链表对缓存不友好，可以通过其他方式优化比如hash,跳表，红黑二叉树等来实现快速查询，B+树
    - 可以通过其他存储方式落地，比如mysql建立表。
  - 用逆邻接表来实现用户的粉丝
- 搜索
  - 广度搜索：地毯式搜索
  - 深度搜索：深度优先式搜索

```
package main

import "fmt"

//无向图
type graph struct {
    vertex int
    list   map[int][]int
    found  bool
}

func main() {
    g := NewGraph(8)
    g.addVertex(1, 2)
    g.addVertex(1, 4)
    g.addVertex(2, 3)
    g.addVertex(2, 5)
    g.addVertex(4, 5)
    g.addVertex(3, 6)
    g.addVertex(5, 6)
    g.addVertex(5, 7)
    g.addVertex(6, 8)
    g.addVertex(7, 8)
    g.bfs(1, 7)
}

//创建图
func NewGraph(v int) *graph {
    g := new(graph)
    g.vertex = v
    g.found = false
    g.list = map[int][]int{}
    i := 0
    for i < v {
        g.list[i] = make([]int, 0)
        i++
    }
    return g
}

//添加边
func (g *graph) addVertex(t int, s int) {
    g.list[t] = push(g.list[t], s)
    g.list[s] = push(g.list[s], t)
}
```

```
//取出切片第一个
func pop(list []int) (int, []int) {
    if len(list) > 0 {
        a := list[0]
        b := list[1:]
        return a, b
    } else {
        return -1, list
    }
}

//推入切片
func push(list []int, value int) []int {
    result := append(list, value)
    return result
}

//广度优先搜索
func (g *graph) bfs(s int, t int) {
    if s == t {
        return
    }
    visited := make([]bool, g.vertex+1)
    var queue []int
    queue = append(queue, s)
    prev := make([]int, g.vertex+1)
    i := 0
    for i < len(prev) {
        prev[i] = -1
        i++
    }
    for len(queue) != 0 {
        var w int
        w, queue = pop(queue)
        for j := 0; j < len(g.list[w]); j++ {
            q := g.list[w][j]
            fmt.Println(q)
            if !visited[q] {
                prev[q] = w
                if q == t {
                    fmt.Println(prev)
                    printPath(prev, s, t)
                    return
                }
                visited[q] = true
```

```
                queue = append(queue, q)
            }
        }
    }
}

//深度优先搜索
func (g *graph) dfs(s int, t int) {
    prev := make([]int, g.vertex+1)
    for i := 0; i < len(prev); i++ {
        prev[i] = -1
    }
    visit := make([]bool, g.vertex+1)
    g.recurDsf(s, t, prev, visit)
    fmt.Println(prev)
    printPath(prev, s, t)
}

func (g *graph) recurDsf(w int, t int, prev []int, visited []bool) {
    if g.found {
        return
    }
    if w == t {
        g.found = true
        return
    }
    visited[w] = true
    for i := 0; i < len(g.list[w]); i++ {
        q := g.list[w][i]
        if !visited[q] {
            prev[q] = w
            g.recurDsf(g.list[w][i], t, prev, visited)
        }
    }
}

//深度优先搜做
func printPath(prev []int, s int, t int) {
    if prev[t] != -1 && s != t {
        printPath(prev, s, prev[t])
    }
    fmt.Println(t, "  ")
}
```

散列表

散列表

散列表分类

- 线性探索散列表
- 链表式散列表

时间复杂度

- 修改（增删改）时间复杂度 O(1)
- 查询时间复杂度O(1)

```
package hash

import (
    "bytes"
    "crypto/md5"
    "crypto/sha1"
    "encoding/binary"
    "fmt"
)

//无链表的散列表
type NoLinkHashTable struct {
    Buckets  []Bucket
    Capacity int32
    Used     int32
}

//桶
type Bucket struct {
    HKey string
    Data int
}

func NolinkHashTable(length int32) {
    //新建算列表
    ht := NoLinkHashTable{}
    ht.Capacity = length
```

```
    ht.Buckets = make([]Bucket, length)
    for i := 0; i < 5; i++ {
        ht.hSet("abc", i)
    }
    fmt.Println(ht)
    one := ht.hGet("abc")
    fmt.Println(one)
}

func (ht *NoLinkHashTable) hSet(key string, value int) {
    bucket := Bucket{key, value}
    hashCode := GetHashCode(key, 1)
    index := hashCode % (ht.Capacity - 1)
    ht.addBucket(index, key, bucket)
}

func (ht *NoLinkHashTable) hGet(key string) Bucket {
    hashCode := GetHashCode(key, 1)
    index := hashCode % (ht.Capacity - 1)
    return ht.findBucket(index, key)
}

func (ht *NoLinkHashTable) findBucket(index int32, key string) Bucket {
    var empty = Bucket{}
    if ok := ht.Buckets[index]; ok != empty {
        if ht.Buckets[index].HKey == key {
            return ht.Buckets[index]
        } else {
            index++
        }
        for {
            if ok := ht.Buckets[index]; ok != empty {
                if ht.Buckets[index].HKey == key {
                    return ht.Buckets[index]
                } else {
                    index++
                }
            } else {
                return empty
            }

        }
    }
    return empty
}
```

```
func (ht *NoLinkHashTable) addBucket(index int32, key string, bucket Bucket) {
    if ht.Capacity == ht.Used {
        panic("Table is full")
    }
    var empty = Bucket{}
    if ok := ht.Buckets[index]; ok != empty {
        if ok.HKey == key {
            ht.Buckets[index] = bucket
            return
        }
        //已经存在
        index++
        for {
            //fmt.Println(index)
            if index >= ht.Capacity {
                index = 0
            }
            if ok := ht.Buckets[index]; ok != empty {
                if ok.HKey == key {
                    ht.Buckets[index] = bucket
                    return
                }
                index++
            } else {
                ht.Buckets[index] = bucket
                ht.Used++
                break
            }
        }
    } else {
        ht.Buckets[index] = bucket
        ht.Used++
    }

}
func GetHashCode(key string, hashType int) int32 {
    var Result []byte
    switch hashType {
    case 1:
        Md5Inst := md5.New()
        Md5Inst.Write([]byte(key))
        Result = Md5Inst.Sum([]byte(""))
    case 2:
        Sha1Inst := sha1.New()
        Sha1Inst.Write([]byte(key))
        Result = Sha1Inst.Sum([]byte(""))
```

```
    }
    bin_buf := bytes.NewBuffer(Result)
    var x int32
    binary.Read(bin_buf, binary.BigEndian, &x)
    return x
}
```

堆

堆：是一个完全二叉树，可以用数组来保存它。

实现

- 创建堆：对顶的元素是最大或者最小的
- 调整堆：自下向上，或者自上而下进行调整，调整方式，就是交换
- 插入一个元素：将新元素插入堆底，自下上向上调整
- 删除堆顶：将堆底和堆顶元素调换，调整堆最后删除被移到堆底的元素

总结

- 所有的一起操作的都是为了保证堆是完全二叉树，这样保存是使用内存最少的，并且能实现快速排序和查找最新小值或者最大值。

应用

- 优先级队列
  - 合并小文件
  - 高性能定时器
- TopK问题
  - 维护一个大小为K的小顶堆，如果新来的元素比它大，就删除堆顶插入这个元素，小于这个元素不做处理
- 中位数问题
  - 维护两个堆，一个大顶堆，一个小顶堆，可以是一半一半，或者某个特定百分比。在插入完成后调整比例。两个杯子相互倒水

```
package heap

import (
    "fmt"
)

type Node struct {
    Value int
    Key   string
```

```
}

type Heap struct {
    list   []*Node
    length int
}

//创建堆
func CreateHeap() {
    arrList := []int{1, 2, 11, 3, 7, 8, 4, 5}
    var myHeap Heap
    myHeap.list = append(myHeap.list, &Node{})
    for _, value := range arrList {
        tmp := Node{}
        tmp.Value = value
        myHeap.InsertHeap(&tmp)
    }
    for {
        node := myHeap.GetTopHeap()
        fmt.Println(node)
    }
    myHeap.SortHeap(myHeap.list)
    heapShow(myHeap.list)
}

//插入堆
func (h *Heap) InsertHeap(one *Node) {
    h.list = append(h.list, one)
    length := len(h.list)
    h.length = length - 1
    h.AdjustHeap(h.length)
}

//堆排序
func (h *Heap) SortHeap(heaps []*Node) {
    length := len(heaps)
    length = length - 1
    if length == 1 {
        return
    }
    if length == 2 {
        h.AdjustHeap(length - 1)
    }
    for length > 0 {
        h.SliceNodeSwap(1, length)
        length--
```

```
        h.Heapfiy(length, 1)
    }
    //反序
    minPos := 1
    maxPos := h.length
    for minPos < maxPos {
        h.SliceNodeSwap(minPos, maxPos)
        minPos++
        maxPos--
    }
}

//自下而上调整
func (h *Heap) AdjustHeap(length int) {
    if length < 1 {
        return
    }
    if length == 2 {
        if h.list[length].Value > h.list[length-1].Value {
            h.SliceNodeSwap(length, length-1)
        }
        return
    }
    i := length
    for i/2 > 0 && h.list[i].Value > h.list[i/2].Value {
        h.SliceNodeSwap(i, i/2)
        i = i / 2
    }
    return
}

//输出heap
func heapShow(heaps []*Node) {
    for one, value := range heaps {
        fmt.Println(one, value)
    }
}

//node slice交换
func (h *Heap) SliceNodeSwap(i int, j int) {
    x := h.list[i]
    h.list[i] = h.list[j]
    h.list[j] = x
}

//自上向下堆化
```

```
func (h *Heap) Heapfiy(length int, pos int) {
    for {
        maxPos := pos
        if pos*2 < length && h.list[pos].Value < h.list[pos*2].Value {
            maxPos = pos * 2
        }
        if pos*2+1 < length && h.list[maxPos].Value < h.list[pos*2+1].Value {
            maxPos = pos*2 + 1
        }
        if maxPos == pos {
            break
        }
        h.SliceNodeSwap(pos, maxPos)
        pos = maxPos
    }
}

//获取堆顶
func (h *Heap) GetTopHeap() *Node {
    if h.length == 0 {
        panic("Heap is empty")
    }
    top := h.list[1]
    //堆顶和堆底交换
    h.SliceNodeSwap(1, len(h.list)-1)
    length := len(h.list) - 2
    fmt.Println(length)
    h.Heapfiy(length, 1)
    heapShow(h.list)
    h.list = append(h.list[:length+1], h.list[length+2:]...)
    h.length--
    return top

}
```

链表

链表分类

- 单链表
- 双链表
- 环链表

时间复杂度

- 修改（增删改）时间复杂度 O(1)
- 查询时间复杂度O(n)

```
package linkedList

import "fmt"

type Node struct {
    Data      int
    NextPoint *Node
    PrePoint  *Node
}

type LinkedList struct {
    head    *Node
    current *Node
    tail    *Node
}

func CreatLinkedList() {
    data := []int{1, 21, 31, 51, 62, 2, 3, 42, 33, 12, 12}
    link := LinkedList{}
    var currentNode *Node
    for i := 0; i < len(data); i++ {
        currentNode = new(Node)
        currentNode.Data = data[i]
        insertNode(&link, currentNode)
    }
    showLinkedList(link)
}
```

```
func showLinkedList(link LinkedList) {
    var currentNode *Node
    currentNode = link.head
    for {
        fmt.Println("Node:", currentNode.Data)
        if currentNode.NextPoint == nil {
            break
        } else {
            currentNode = currentNode.NextPoint
        }
    }
}

func insertNode(link *LinkedList, node *Node) {
    if link.head == nil {
        link.head = node
        link.tail = node
        link.current = node
    } else {
        link.tail.NextPoint = node
        node.PrePoint = link.tail
        link.tail = node
    }
}
```

跳跃表

定义

在一组数据中，按照一定规则对数据创建层级索引，通过自上至下的索引查询查找数据位置

特点

- 二分查找针对数组，为了实现不使用针对数组二分查找，产生了跳跃表
- 跳跃表使用链表
- 有多级索引

时间复杂度

- 修改（增删改）时间复杂度 O(lgn)
- 查询时间复杂度O(lgn)

空间复杂度

- 空间复杂度根据索引生成方法有关，例子中的大概是O(n)
- n/2+n/4+....+4+2 等比求和 Sn=(a1-anq)/1-q O(n-2) 约等于 O(n)

```
package skipList

import "fmt"

type Node struct {
    Data      int
    NextPoint *Node
    PrePoint  *Node
    NextLevel *Node
}

type LinkedList struct {
    Head    *Node
    Current *Node
    Tail    *Node
    Length  int
    Level   int
}
```

```
type SkipList struct {
    List        LinkedList
    FirstIndex  LinkedList
    SecondIndex LinkedList
    //ThirdIndex  LinkedList
}

func InitSkipList() {
    data := []int{11, 12, 13, 19, 21, 31, 33, 42, 51, 62}
    sl := SkipList{}
    sl.initSkip(data)
    //sl.find(19)
    sl.add(11)
    showSkipList(sl)
}

func showSkipLinkedList(link LinkedList, name int) {
    var currentNode *Node
    currentNode = link.Head
    for {
        i := 1
        fmt.Print(name, "-Node:", currentNode.Data)
        if currentNode.NextPoint == nil {
            break
        } else {
            currentNode = currentNode.NextPoint
        }
        if name == 1 {
            fmt.Print("-------->")
        } else if name == 2 {
            for i <= 3 {
                fmt.Print("-------->")
                i++
            }
        } else {
            for i <= 7 {
                fmt.Print("-------->")
                i++
            }
        }
    }
    fmt.Println("")
}

func (sl *SkipList) initSkip(list []int) {
```

```
    sl.List = LinkedList{}
    sl.FirstIndex = LinkedList{}
    sl.SecondIndex = LinkedList{}
    var currentNode *Node
    for i := 0; i < len(list); i++ {
        currentNode = new(Node)
        currentNode.Data = list[i]
        addNode(sl, currentNode)
        //insertToLink(&sl.List, currentNode)
    }
    showSkipList(*sl)
}
func (sl *SkipList) find(x int) {
    var current *Node
    current = sl.SecondIndex.Head
    if current.Data == x {
        fmt.Println(current.Data)
        return
    }
    if x < current.Data {
        panic("No exist in skipList")
        return
    }
    for {
        if x > current.Data {
            fmt.Println(current.Data)
            current = current.NextPoint
        } else if x < current.Data {
            //下到底层索引
            fmt.Println(current.Data)
            current = current.PrePoint.NextLevel.NextPoint
        } else {
            fmt.Println(current.Data)
            return
        }
    }
}
func (sl *SkipList) add(x int) {
    var current *Node
    current = sl.SecondIndex.Head
    if current.Data == x {
        panic("Had existed in skipList")
        return
    }
    if x < current.Data {
        newNode2 := new(Node)
```

```
        newNode2.Data = x
        newNode2.NextPoint = sl.SecondIndex.Head
        sl.SecondIndex.Head.PrePoint = newNode2
        sl.SecondIndex.Head = newNode2
        newNode1 := new(Node)
        newNode1.Data = x
        newNode1.NextPoint = sl.FirstIndex.Head
        sl.FirstIndex.Head.PrePoint = newNode1
        sl.FirstIndex.Head = newNode1
        newNode := new(Node)
        newNode.Data = x
        newNode.NextPoint = sl.List.Head
        sl.SecondIndex.Head.PrePoint = newNode
        sl.List.Head = newNode
        return
    }
    for {
        if x > current.Data {
            if current.NextPoint == nil {
                if current.NextLevel != nil {
                    current = current.NextLevel
                } else {
                    //插入
                    newNode := new(Node)
                    newNode.Data = x
                    current.NextPoint = newNode
                    newNode.PrePoint = current
                    return
                }
            } else {
                fmt.Println(current.Data)
                current = current.NextPoint
            }

        } else if x < current.Data {
            //向下去寻找第一个大于x的值
            fmt.Println(current.Data)
            if current.PrePoint.NextLevel != nil {
                current = current.PrePoint.NextLevel.NextPoint
            } else {
                //插入
                newNode := new(Node)
                newNode.Data = x
                current.PrePoint.NextPoint = newNode
                newNode.NextPoint = current
                current.PrePoint = newNode
```

```
                return
            }
        } else {
            fmt.Println(current.Data)
            return
        }
    }
}
func showSkipList(sl SkipList) {
    showSkipLinkedList(sl.SecondIndex, 3)
    fmt.Println("")
    showSkipLinkedList(sl.FirstIndex, 2)
    fmt.Println("")
    showSkipLinkedList(sl.List, 1)
}
func addNode(skipList *SkipList, node *Node) {
    insertToLink(&skipList.List, node)
    if skipList.FirstIndex.Length == 0 || ((skipList.List.Length-1)%2 == 0 && sk
ipList.List.Length > 2) {
        newNode := new(Node)
        newNode.Data = node.Data
        newNode.NextLevel = node
        insertToLink(&skipList.FirstIndex, newNode)
        if skipList.SecondIndex.Length == 0 || ((skipList.FirstIndex.Length-1)%2
== 0 && skipList.FirstIndex.Length > 2) {
            newNode2 := new(Node)
            newNode2.Data = node.Data
            newNode2.NextLevel = newNode
            insertToLink(&skipList.SecondIndex, newNode2)
        }
    }
}

func insertToLink(link *LinkedList, node *Node) {
    if link.Head == nil {
        link.Head = node
        link.Tail = node
        link.Current = node
    } else {
        link.Tail.NextPoint = node
        node.PrePoint = link.Tail
        link.Tail = node
    }
    link.Length++
}
```

字典树

算法描述：trie树的本质，就是利用字符串之间的公共前缀，将重复的前缀合并在一起。
时间复杂度：构建O(n)，查询O(k)

算法步骤

- 根节点/ 什么都不表示
- 做一个字典比如a-z 字母表
- 没一个节点包含这26个字母的字典表，每个位置保存下个节点的指针。

算法分析

缺点：

- trie树比较消耗内存：因为他没一层都保存一个字典表，就算这层的节点只有一个也要有一组表
- 使用的是指针类型，内存不连续对存储不友好，性能打折扣

  优点：
- 查询效率比较高，对于一些范围较小的或者内存不敏感的应用可以使用
- 特别适用自动补全类应用

```
package main

import "fmt"

type TrieNode struct {
    value      int
    dictionary [26]*TrieNode
}

type TrieTree struct {
    root *TrieNode
}

func main() {
    tree := createTree()
    //fmt.Println(tree)
    flag := tree.findWord("her")
    fmt.Println(flag)
```

```
    flag = tree.findWord("x")
    fmt.Println(flag)
}

func createTree() TrieTree {
    arrList := []string{"how", "hi", "her", "hello", "so", "see"}
    myTree := TrieTree{}
    //添加跟节点
    myTree.root = &TrieNode{}
    for _, value := range arrList {
        myTree.addWord(value)
    }
    return myTree
}
func (t *TrieTree) addWord(word string) {
    fmt.Println(word)
    nowNode := t.root
    a := int('a')
    var char int
    for i := 0; i < len(word); i++ {
        char = int(word[i])
        if nowNode.dictionary[char-a] != nil {
            nowNode = nowNode.dictionary[char-a]
            continue
        } else {
            newNode := &TrieNode{}
            nowNode.dictionary[char-a] = newNode
            nowNode = newNode
            continue
        }
    }
}

func (t *TrieTree) findWord(word string) int {
    nowNode := t.root
    a := int('a')
    var char int
    for i := 0; i < len(word); i++ {
        char = int(word[i])
        if nowNode.dictionary[char-a] != nil {
            return 0
        } else {
            nowNode = nowNode.dictionary[char-a]
        }
        if i == len(word)-1 {
            return 1
```

```
        }
    }
    return 0
}
```

向量空间

定义

将用户的各种纬度的数据，做成向量。在向量空间内计算，两个向量的相似度，实现近似比较和分类。

应用场景

- 推荐系统

两种推荐方式

- 基于用户相似度做推荐
- 基于内容相似度做推荐

算法

- 欧几里得距离：d=根号下（x1-y1）平方+（x2-y2）平方

应用

- 基于用户相似度正推
- 基于内容相似度反推