数据结构介绍

前端精进（科班方向）：算法与数据结构入门

程序 = 算法 + 数据结构

这是 Pascal 之父 Niklaus Wirth 写的一本书的书名

1976 年发表

引用次数非常多

很多讲算法和数据结构的书都会引用这句话

我并不认同这句话

- 理由

✓ 程序并不只是这两点
✓ 还有团队合作、测试、部署、工具使用等很多点
✓ 算法和数据结构也不是对等关系
✓ 数据结构更像是算法的基础
✓ 你必须先搞清楚数据结构，才能想出算法

```
array = [0,1,2,3,4,5,6,7,8,9,10]
```

最常见的数据

一组非负数 list

list 的 API

- 功能

✓ `list#add(item)`
✓ `list#delete(index)`
✓ `list#set(index, item)`
✓ `list#insert(index, item)`
✓ `list#get(index)`
✓ `list#length`

- 也就是增删改查

如何存放这一组数

默认放在内存

#100	#101	#102	#103	#104
1	2	3	4	5

一：顺序分配内存

一个挨一个存

#100	#101	#102	#103	#104
1	2	3	4	5

list = [1,2,3,4,5]

list = #100

如何获取 list#length

- 记在 array 上
  - ✓ 在 list 上再存一个 length 属性，记录长度
  - ✓ 缺点：要求 list 是一个更复杂的结构
- 在末尾标记
  - ✓ 遇到一个特殊值 \0 (空) 就表示数组结束
  - ✓ 缺点：无法存储特殊值
- 在开头标记
  - ✓ 把长度存在最前面一格（比如 4 个字节）
  - ✓ 缺点：第一个能存的数量有限
- 记在其他地方
  - ✓ 在其他地方记录 list 的长度
  - ✓ 缺点：需要额外的维护代码

list#add(item)

- 思路
  - ✓ 直接在内存末尾添加一个 item
- 问题
  - ✓ 如果内存末尾之后的内存已经被占用了怎么办

#100	#101	#102	#103	#104
1	2	3	4	5

#105	#106	#107	#108	#109
11	12	13	14	15

- 解决
  - ✓ 把 list 拷贝到另一块空地，然后再追加 item

list#add(item)

- 操作

1. 检查 list 后面是否有内存
2. 有：存入 item；增加 length
3. 无：查找满足条件的新地址，复制 list 的每一项到新地址，然后回到步骤 1

- 操作数

✓ 最好情况：3
✓ 最差情况：2 + n + 3

list#delete(index)

- 思路
  - ✓ 将 index 之后的每一项依次往前挪一位
- 操作（假设 n = 5, index = 2）
  - ✓ n - index - 1 次赋值
  - ✓ 将 length 减一
- 操作数
  - ✓ n - index - 1 + 1

list#set(index, item)

- 思路
  - ✓ 直接赋值
- 操作
  - ✓ 将 list[index] 赋值为 item
- 操作数
  - ✓ 1

list#insert(index, item)

- 思路
  - ✓ 挪开一个位置，放入 item
- 操作
  - ✓ n - index 次后挪
  - ✓ 将 list[index] 赋值为 item
  - ✓ 将 length 加一
- 操作数
  - ✓ n - index + 1 + 1

list#get(index)

- 思路
  - ✓ 直接读取
- 操作
  - ✓ 读取 list[index] 的值
- 操作数
  - ✓ 1

顺序分配总结

- 操作数

✓ list#length 需要 1 次
✓ list#get(index) 需要 1 次
✓ list#set(index, item) 需要 1 次
✓ list#insert(index, item) 需要 n - index + 2 次
✓ list#delete(index) 需要 n - index 次
✓ list#add(item) 需要 3 或 n + 5 次

#100	#101
1	#136

#072	#073
3	#202

#195	#196
5	null

#136	#137
2	#072

#202	#203
4	#195

二：链接分配内存

一个链一个

#XX0	#XX1
data	link

如何获取 list#length

- 如果没有额外信息

✓ 一直读取下一个节点，知道遇到 null

- 操作

1. 置 l 为 0
2. 读取一个节点，看它是否为 null
3. 是：返回 l
4. 否：l 加一，将此节点的 link 作为下一个节点，回到2

- 如果有额外信息

✓ 操作数 1

list#add(item)

- 思路
  - ✓ 创建一个节点，放在 list 末尾
- 操作
  - ✓ 创建节点 t (item, null)
  - ✓ 从 list 遍历到最后一项 last
  - ✓ 让 last.link = t
- 操作数
  - ✓ 1 + n + 1

list#delete(index)

- 思路
  - ✓ 让 index 的上一项指向 index 的下一项
- 操作
  - ✓ 找到第 index - 1 项，设为 t1
  - ✓ 要删除的项 d = t1.link，下一项 t2 = d.link
  - ✓ 赋值 t1.link = d.link = t1.link.link
  - ✓ 主要内存回收 delete d
- 操作数
  - ✓ index - 1 + 1 + 1

list#set(index, item)

- 思路
  - ✓ 找到第 index 项，然后赋值
- 操作
  - ✓ 读取 index + 1 次 link，然后赋值
- 操作数
  - ✓ index + 1 + 1

list#insert(index, item)

- 思路
  - ✓ index -> item -> index.link
- 操作
  - ✓ 读取 index + 1 次，找到 index 项，设为 t1
  - ✓ 用 item 创建节点，设为 t0
  - ✓ t0.link = t1.link
  - ✓ t1.link = t0
- 操作数
  - ✓ index + 1 + 1 + 1 + 1

list#get(index)

- 思路
  - ✓ 找到第 index 项
- 操作
  - ✓ 读取 index + 1 次 link，然后赋值
- 操作数
  - ✓ index + 1

链接分配总结

- 操作数

✓ list#length 需要 n 次
✓ list#get(index) 需要 index + 1 次
✓ list#set(index, item) 需要 index + 2 次
✓ list#insert(index, item) 需要 index + 4 次
✓ list#delete(index) 需要 index + 1 次
✓ list#add(item) 需要 n + 2 次

对比两种方式

链接分配完全没好处嘛

那是因为 API 不合适

如果改变 API 的参数会怎样

改变 API

- 直接获取 item

✓ item = list.get(2)
✓ list.insert(item, 9)

- 操作数

✓ list#length 需要 n 次
✓ list#get(index) 需要 index + 1 次
✓ list#set(item, value) 需要 1 次
✓ list#insert(item, item2) 需要 3 次
✓ list#delete(item) 需要 2 次
✓ list#add(item) 需要 n + 2 次

重新对比

- ✓ list#length 需要 1 次
- ✓ list#get(index) 需要 1 次
- ✓ list#set(index, item) 需要 1 次
- ✓ list#insert(index, item) 需要 n - index + 2 次
- ✓ list#delete(index) 需要 n - index 次
- ✓ list#add(item) 需要 3 或 n + 5 次

顺序分配 +API1

链接分配 + API2

- ✓ list#length 需要 n 次
- ✓ list#get(index) 需要 index + 1 次
- ✓ list#set(item, value) 需要 1 次
- ✓ list#insert(item, item2) 需要 3 次
- ✓ list#delete(item) 需要 2 次
- ✓ list#add(item) 需要 n + 2 次

总结

- 同样一组数

✓ 逻辑结构一样
✓ 存储结构不一样
✓ 可以提供不同的 API

什么是数据结构

定义

数据结构

=

数据 + 逻辑结构 + API

但是程序员需要研究存储结构以加速 API

逻辑结构举例

- 线性表 Linear List
  - ✓ 顺序存储的线性表叫做数组（应该叫顺序表）
  - ✓ 链接存储的线性表叫做链表
  - ✓ 链表又分为单向链表、双向链表、循环链表等
- 树型结构
  - ✓ 树、二叉树*、二叉搜索树、红黑树、B树、堆
- 哈希结构
- 图
- 其他

存储结构举例

- 顺序存储
  - ✓ 用连续的内存存储
  - ✓ 二叉树可以顺序存储
- 链接存储
  - ✓ 用分散的内存存储，中间用地址来链接
  - ✓ 链表、树都可以用这种方式存
- 混合存储
  - ✓ 分散存储，但是每一块存储空间里面又是连续内存
- 其他

API 举例

- 队列 Queue
  - ✓ 提供入队 enqueue 和出队 dequeue 操作的线性表
- 栈 Stack
  - ✓ 提供押栈 push 和弹栈 pop 操作的线性表
- 二叉堆 Binary Heap
  - ✓ 提供维护堆性质的 API 的树型结构

学习数据结构的难点

- 了解各种数据结构
  - ✓ 逻辑结构是什么（用途）
  - ✓ API 有什么（细节）
- 用代码实现各种数据结构
  - ✓ 用什么存储结构
  - ✓ API 怎么实现
- 实际使用
  - ✓ 面试题、考题
  - ✓ 用于软件开发

那算法呢?

算法和数据结构不可分割

我们会在学习排序算法的时候，学习各种数据结构

这节课没有代码吗?

- 降低难度

✓ 一开始就代码，容易觉得枯燥
✓ 一开始就代码，容易陷入细节

- 锻炼思维

✓ 数据结构是一种抽象的知识
✓ 工作中使用数据结构也只是记忆 API，都被封装好了

再见

下节课开始学习算法基本知识