数据结构

第 9 周 – 动态数据结构：内存管理模型与单链表

Koczka Ferenc 博士 · 埃斯特哈齐·卡罗伊大学 · 2026

复习

- 到目前为止，我们的数据结构都基于**数组**：顺序存储、直接访问
- 栈、队列：在数组的**两端**操作 → O(1)
- **指针 / 引用**（第 1 周）：存放内存地址的数据

今天的问题

- 静态（基于数组的）存储有什么局限?
- 程序怎样分配和释放内存?
- **链表**是怎样构成的？它的操作有多快?

静态数据结构的局限

- 大小必须**事先**给定 → 太小或者浪费
- 需要一块**连续**的空闲内存 – 对于很大的数组，不一定总能找到
- 在中间插入和删除是 **O(n)** – 需要移动元素
- 动态数组（第 3 周）有所帮助，但扩容时**所有元素都要复制**

> **思路：**把元素**分散**地存放在内存中的任意位置，并让每个元素**知道下一个元素在哪里**。

动态内存管理模型

一个简单的模型

- **内存**是一个大数组，其元素是带有 **Value** 和 **Next** 字段的记录
- **指针**是这个数组的**下标** – 特殊值 **Nil** 表示“不指向任何地方”
- 记号：`Elem[P].Value`、`Elem[P].Next` – P 所指元素的字段

下标	Value	Next
1	C	5
2		4
3	A	7
4		6
5	D	Nil
6		8
7	B	1
8		Nil



链表（从 Head 开始）：

3 → 7 → 1 → 5 → Nil

A → B → C → D

空闲链表：

2 → 4 → 6 → 8 → Nil

这里的“指针”是数组下标 –
在真实内存中它是地址。

分配与释放

空闲位置也组织成一个**链表**（FreeList）。

```
Function Reserve(P)
  If FreeList = Nil then
    Return false        // 内存已满
  else
    P ← FreeList
    FreeList ← Elem[P].Next
    Return true
  End if
End function
```

```
Procedure Free(P)
  Elem[P].Next ← FreeList
  FreeList ← P
End procedure
```

- 两者都是 **O(1)**
- 从空闲链表的**前端**取出，也放回前端 – 就像栈一样！

实际中空闲内存由操作系统和运行时管理 – 在旧的作业中也可以省略它的实现。

实际情况

	分配	释放
C	`p = malloc(sizeof(Node))`	`free(p)` – 程序员的责任
Java	`new Node(x)`	**自动** – 垃圾回收器（GC）
Go	`&Node{Value: x}` 或 `new(Node)`	**自动** – 垃圾回收器（GC）

手工管理内存时的典型错误：

- **内存泄漏**：分配的内存从不释放 → 程序占用的内存越来越多
- **悬空指针**：指向已经释放的内存并继续使用它
- 垃圾回收器可以防止这些错误：**没有任何指针指向**的内存会被释放

动态数据结构：优点与代价

优点

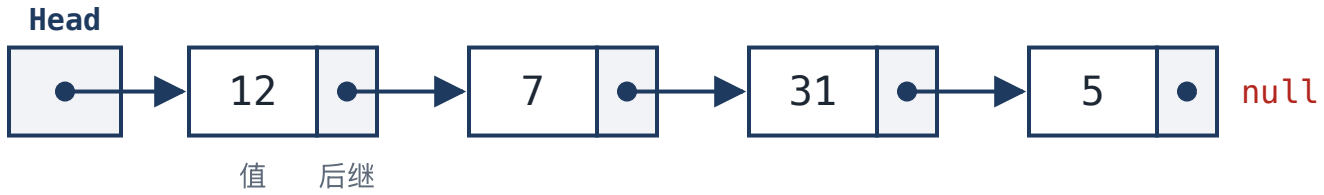
- 只占用**当前需要**的内存 → 内存可以在多个数据结构之间**共享**（资源共享）
- 不需要连续的内存区域
- 通过**修改指针**来插入和删除，不需要移动元素

代价

- 每个元素还要存放**一个或多个指针** → 占用更多内存
- **没有直接访问** – 要到达第 i 个元素，必须经过前面的元素
- 分配/释放和跟随指针都**需要时间**

单链表

链表的结构



- 每个**结点**：一个值 + 一个指向**下一个**结点的指针
- **Head**：指向第一个结点的指针 · 最后一个结点的 `Next` 字段为 **Nil**
- **空链表**：Head = Nil

性质	链表
元素	同构
大小	**动态**
存储	**链式**
访问	**顺序**

Java 和 Go 中的链表结点

Java

```java
class Node {
    int value;
    Node next;      // 指向下一个结点的引用

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

Node head = null;   // 空链表
```

Go

```go
type Node struct {
    Value int
    Next  *Node     // 指向下一个结点的指针
}

type List struct {
    Head *Node      // nil:空链表
}
```

伪代码	Java	Go
`Elem[P].Value` · `Elem[P].Next`	`p.value` · `p.next`	`p.Value` · `p.Next`
`Nil` · `Head`	`null` · `head`	`nil` · `l.Head`

遍历与查找

遍历（处理）

```
Procedure Traverse(Head)
  P ← Head
  While P ≠ Nil
    Process(Elem[P].Value)
    P ← Elem[P].Next
  End while
End procedure
```

查找

```
Function Search(Head, x)
  P ← Head
  While P ≠ Nil and Elem[P].Value ≠ x
    P ← Elem[P].Next
  End while
  Return P              // 未找到时为 Nil
End function
```

Java

```java
for (Node p = head; p != null; p = p.next)
    System.out.println(p.value);

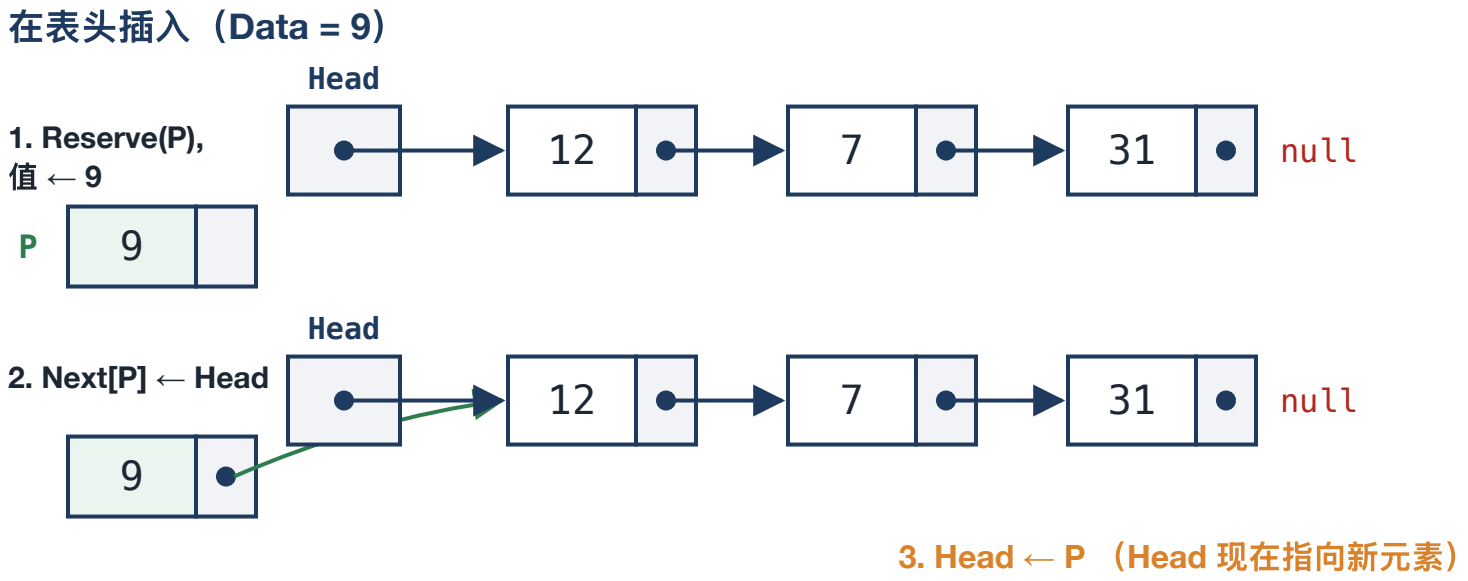
Node find(int x) {
    Node p = head;
    while (p != null && p.value != x)
        p = p.next;
    return p;
}
```

Go

```go
for p := l.Head; p != nil; p = p.Next {
    fmt.Println(p.Value)
}
```

两者都是 **O(n)**。即使链表有序也不能二分查找 – 没有直接访问！

在表头插入


在表头插入（Data = 9）
Head
1. Reserve(P),
值 ← 9
12
7
31
null
P
9
Head
2. Next[P] ← Head
12
7
31
null
9
3. Head ← P （Head 现在指向新元素）

在表头插入和删除

在表头插入 – O(1)

```
Procedure InsertFirst(Head, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Head
    Head ← P
  else
    ERROR("no free memory")
  End if
End procedure
```

删除表头 – O(1)

```
Procedure DeleteFirst(Head)
  If Head ≠ Nil then
    P ← Head
    Head ← Elem[P].Next
    Free(P)
  End if
End procedure
```

Java

```java
void insertFirst(int x) {
    head = new Node(x, head);
}

int removeFirst() {
    int x = head.value;
    head = head.next;   // 由 GC 释放
    return x;
}
```

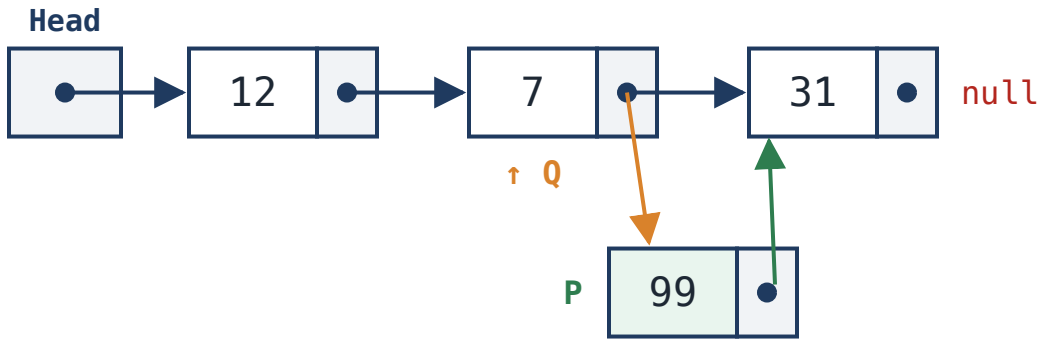
Go

```go
func (l *List) InsertFirst(x int) {
    l.Head = &Node{Value: x, Next: l.Head}
}

func (l *List) RemoveFirst() int {
    x := l.Head.Value
    l.Head = l.Head.Next
    return x
}
```

在给定结点之后插入

在 Q 所指元素之后插入



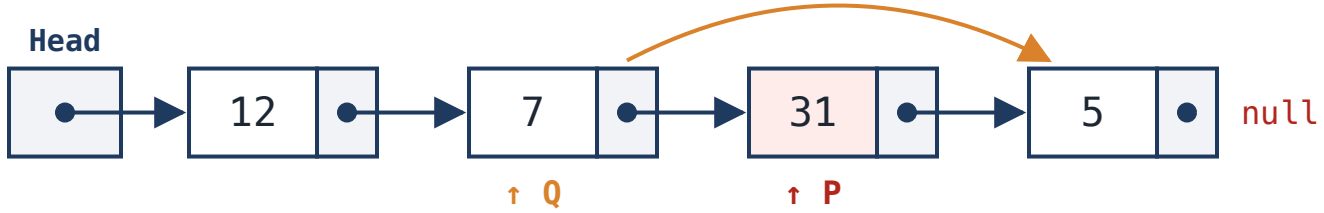
1. `Next[P] ← Next[Q]`
2. `Next[Q] ← P`

顺序很重要!

如果顺序反过来（先执行 `Next[Q] ← P`），链表的剩余部分就会**丢失** – 再也没有指针指向它了!

删除给定结点之后的结点

删除 Q 所指元素之后的元素



P ← Next[Q] ; Next[Q] ← Next[P] ; Free(P)

```
Procedure InsertAfter(Q, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Elem[Q].Next
    Elem[Q].Next ← P
  End if
End procedure
```

```
Procedure DeleteAfter(Q)
  P ← Elem[Q].Next
  If P ≠ Nil then
    Elem[Q].Next ← Elem[P].Next
    Free(P)
  End if
End procedure
```

两者都是 **O(1)** – 前提是已经有了 Q。（找到 Q 需要 O(n)。）

在表尾的操作

在表尾插入 – O(n)

```
Procedure InsertLast(Head, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Nil
    If Head = Nil then
      Head ← P
    else
      Q ← Head
      While Elem[Q].Next ≠ Nil
        Q ← Elem[Q].Next
      End while
      Elem[Q].Next ← P
    End if
  End if
End procedure
```

删除表尾 – O(n)

```
Procedure DeleteLast(Head)
  If Head = Nil then exit
  If Elem[Head].Next = Nil then
    Free(Head) ; Head ← Nil
  else
    Q ← Head       // 寻找倒数第二个结点
    While Elem[Elem[Q].Next].Next ≠ Nil
      Q ← Elem[Q].Next
    End while
    Free(Elem[Q].Next)
    Elem[Q].Next ← Nil
  End if
End procedure
```

如果再保存一个 **End** 指针，在表尾插入就变成 O(1) – 但删除表尾不行（仍然要找到倒数第二个结点）。解决办法：双向链表（第 10 周）。

按值删除结点

在单链表中只能删除某个已知结点**之后**的结点 → 查找时还必须记住**前一个**结点。

```
boolean remove(int x) {
    Node prev = null, p = head;
    while (p != null && p.value != x) {
        prev = p;
        p = p.next;
    }
    if (p == null) return false;        // 没有这个元素
    if (prev == null) head = p.next;    // 删除第一个结点
    else prev.next = p.next;
    return true;
}
```

一定要考虑**特殊情况**：空链表、第一个结点、最后一个结点、只有一个结点的链表！

链表还是数组?

操作	数组	动态数组	链表
访问第 i 个元素	**O(1)**	**O(1)**	O(n)
查找	O(n) / 有序时 O(log n)	O(n) / O(log n)	O(n)
在开头插入 / 删除	O(n)	O(n)	**O(1)**
在已知结点之后插入 / 删除	O(n)	O(n)	**O(1)**
在末尾插入	有空位时 O(1)	均摊 O(1)	O(n)，有 End 指针时 O(1)
每个元素占用的空间	值	值（+ 预留）	值 + **指针**

实测（在开头插入 n = 160 000 个元素）：`ArrayList` 870 ms · `LinkedList` 2 ms

读取中间元素 20 000 次：`ArrayList` 0.1 ms · `LinkedList` 357 ms

小结

- **动态数据结构：**元素分散在内存中，由指针连接；大小在运行时改变
- **内存模型：**记录数组 + 空闲链表；`Reserve(P)`、`Free(P)` – 在 Java/Go 中是 `new` 和垃圾回收器
- **链表：**Head → 结点 → 结点 → … → Nil；**顺序**访问
- 在表头和已知结点之后：插入和删除 **O(1)** · 查找、在表尾：**O(n)**
- 修改指针的**顺序**很重要 · 特殊情况：空链表、第一个结点

下周：带哨兵的链表、双向链表、循环链表。

复习题

1. 判断对错：使用动态数据结构可以实现资源共享。
2. 判断对错：链表的元素可以直接（按下标）访问。
3. 判断对错：动态数据结构使用不当会导致内存泄漏。
4. 在 Q 之后插入时，如果先执行 `Elem[Q].Next ← P`，再执行 `Elem[P].Next ← Elem[Q].Next`，会怎样?
5. 用伪代码写出：在 Head 指针给出的链表**表头**插入值 `Data`！（`Reserve(P)` 成功时返回 true。）
6. 为什么即使知道 End 指针，删除表尾仍然是 O(n)?

实践课

✎ 示例程序

程序	Java	Go	演示内容
内存模型	`MemoryModel.java`	`w09/memorymodel`	记录数组、空闲链表、Reserve/Free
链表	`SinglyLinkedList.java`	`w09/singlylist`	所有基本操作
链表与数组	`ListVsArray.java`	–	在开头插入、读取中间元素

> `MemoryModel` 程序完全按照讲课中的伪代码实现 – 跟着打印出的表格看！

✎ 练习

1. **和与最大值：**只遍历一次计算出来！
2. **第 i 个元素：**返回链表的第 **i** 个元素！需要多少步？
3. **计数：****x** 在链表中出现了多少次？
4. **原地逆置：**通过修改指针把链表逆置，不创建新结点！
5. **删除所有 x：**删除所有值为 **x** 的结点！（注意开头的那些！）
6. **中间结点：**只遍历一次 – 一个指针每次走一步，另一个每次走两步！
7. **有序插入：**插入元素，使链表保持递增顺序！

参考答案：`Exercises09.java`、`w09/exercises`

✎ 课后作业

1. 为链表增加一个 **End** 指针！哪些操作变快了？哪些操作需要修改?
2. 把两个**有序**链表合并成一个有序链表，不创建新结点，用 O(n + m) 步！
3. 判断链表中是否有**环**（某个结点的 `Next` 字段指向前面的结点）！（提示：龟兔赛跑。）
4. 用链表实现栈（push = 在表头插入，pop = 删除表头）！为什么这比在表尾操作更好?