

基本的链表（Linked List）数据结构

叶家炜

Sep 09, 2025

在计算机科学中，数据结构是组织和存储数据的基础。数组作为一种常见的数据结构，提供连续的内存空间和随机访问能力，时间复杂度为 $O(1)$ 。然而，数组存在局限性：大小固定，插入和删除元素时需要移动后续所有元素，导致时间复杂度为 $O(n)$ ，这可能造成效率低下和内存浪费。例如，预分配固定大小可能导致内存不足或闲置。为了解决这些问题，链表应运而生。链表是一种动态数据结构，物理存储非连续，逻辑顺序通过指针链接实现，允许高效插入和删除操作。本文将带您深入理解链表的原理，并通过代码实现掌握其核心操作。

1 核心概念：链表的构成

链表的基本构建块是节点（Node）。每个节点包含两个部分：数据域（data）和指针域（next）。数据域存储实际值，指针域存储指向下一个节点的引用。头指针（Head Pointer）是访问链表的入口点，指向第一个节点。如果头指针丢失，整个链表将无法访问，因为它没有其他引用方式。节点结构可以简单表示为 [data | next]，其中 next 指向下一个节点或 null，表示链表结束。

2 链表 vs. 数组：一场经典的博弈

链表和数组在特性上各有优劣，选择取决于具体应用场景。数组分配静态连续内存，大小固定，支持随机访问，时间复杂度为 $O(1)$ ，但插入和删除元素效率低，为 $O(n)$ ，因为需要移动元素。链表分配动态非连续内存，大小可动态调整，访问元素需顺序进行，时间复杂度为 $O(n)$ ，但插入和删除在已知位置时效率高，为 $O(1)$ 。空间开销方面，数组无额外指针开销，而链表每个节点都有指针开销。因此，如果应用需要频繁随机访问，数组更合适；如果需要频繁插入删除，链表更优。

3 单链表的基本操作

单链表是最简单的链表形式，我们将使用 Python 实现基本操作，包括定义类、遍历、插入、删除和搜索。每个操作都配以算法思路、代码实现和详细解读，并分析时间复杂度。

3.1 定义节点类和链表类

首先，定义节点类。每个节点有数据域和指向下一个节点的指针。

```
1 class Node:  
2     def __init__(self, data):
```

```

3     self.data = data # 初始化数据域，存储用户提供的数据
        self.next = None # 初始化指针域，默认指向 None，表示无后续节点

```

这段代码创建了一个 Node 类，构造函数接受 data 参数并初始化 next 为 None。这确保了每个节点可以独立存在，并准备被链接。

接下来，定义链表类，包含头指针。

```

class LinkedList:
2     def __init__(self):
            self.head = None # 初始化头指针为 None，表示空链表

```

链表类通过 head 属性管理整个链表。初始时链表为空，head 为 None。

3.2 遍历 (Traversal)

遍历链表意味着从头指针开始，依次访问每个节点，直到遇到 null。算法思路是使用循环结构，从 head 出发，每次移动到 next 指针，直到 next 为 None。时间复杂度为 $O(n)$ ，因为需要访问每个节点一次。

```

1 def traverse(self):
    current = self.head # 从头指针开始，设置当前节点变量
3     while current is not None: # 循环条件：当前节点不为空
        print(current.data) # 输出当前节点的数据，可根据需要处理数据
5     current = current.next # 移动到下一个节点，更新当前节点引用

```

此代码通过 while 循环遍历链表。current 变量用于跟踪当前位置，循环继续直到 current 为 None。每一步输出数据，并更新 current 到 next。这确保了所有节点都被访问。

3.3 插入 (Insertion)

插入操作分三种情况：在头部、尾部或指定节点后插入。每种情况有不同的时间复杂度。

在头部插入最高效，时间复杂度为 $O(1)$ 。只需创建新节点，将其 next 指向当前 head，然后更新 head。

```

1 def insert_at_head(self, data):
    new_node = Node(data) # 创建新节点实例
3     new_node.next = self.head # 新节点的 next 指向当前头节点，链接到现有链表
        self.head = new_node # 更新头指针指向新节点，使其成为新头

```

这段代码首先创建新节点，然后调整指针：新节点的 next 指向原 head，最后 head 更新为新节点。这确保了新节点插入到链表头部。

在尾部插入需遍历到最后一个节点，因此时间复杂度为 $O(n)$ 。找到尾节点 (next 为 None)，将其 next 指向新节点。

```

def insert_at_tail(self, data):
2     new_node = Node(data) # 创建新节点
        if self.head is None: # 检查链表是否为空

```

```

4     self.head = new_node # 如果为空，直接设置 head 为新节点
5 else:
6     current = self.head # 从头开始遍历
7     while current.next is not None: # 循环直到找到最后一个节点 (next 为 None)
8         current = current.next # 移动到下一个节点
9     current.next = new_node # 将最后一个节点的 next 指向新节点，完成插入

```

代码首先处理空链表情况。如果不是空链表，则遍历到最后一个节点（`current.next` 为 `None`），然后设置其 `next` 为新节点。这确保了新节点添加到链表尾部。

在指定节点后插入，假设已知某个节点引用，时间复杂度为 $O(1)$ 。只需调整指针，无需遍历。

```

1 def insert_after(self, prev_node, data):
2     if prev_node is None: # 检查前驱节点是否存在，避免错误
3         print("Previous node cannot be None") # 输出错误信息
4         return
5     new_node = Node(data) # 创建新节点
6     new_node.next = prev_node.next # 新节点的 next 指向 prev_node 的当前 next
7     prev_node.next = new_node # prev_node 的 next 更新为新节点，完成插入

```

这段代码首先验证 `prev_node` 不为 `None`。然后创建新节点，并调整指针：新节点的 `next` 指向 `prev_node` 的 `next`，`prev_node` 的 `next` 指向新节点。这实现了在指定节点后插入。

3.4 删除 (Deletion)

删除操作也分三种情况：删除头节点、尾节点或指定值节点。每种情况有不同的复杂度。

删除头节点简单，时间复杂度为 $O(1)$ 。只需将 `head` 指向 `head.next`。

```

1 def delete_head(self):
2     if self.head is None: # 检查链表是否为空
3         return # 如果为空，直接返回，无操作
4     self.head = self.head.next # 更新头指针指向下一个节点，原头节点被绕过

```

代码首先检查空链表。然后直接更新 `head` 为 `head.next`，这有效地删除了头节点，因为原头节点不再被引用。删除尾节点需找到倒数第二个节点，因此时间复杂度为 $O(n)$ 。遍历到倒数第二个节点，将其 `next` 设为 `None`。

```

1 def delete_tail(self):
2     if self.head is None: # 空链表检查
3         return
4     if self.head.next is None: # 检查是否只有一个节点
5         self.head = None # 如果是，设置 head 为 None，链表变为空
6         return
7     current = self.head
8     while current.next.next is not None: # 遍历直到倒数第二个节点

```

```

10     current = current.next # 移动当前节点
      current.next = None # 设置倒数第二个节点的 next 为 None, 删除尾节点

```

代码处理了空链表和单节点链表的情况。对于多节点链表，遍历到倒数第二个节点（`current.next.next` 为 `None`），然后设置其 `next` 为 `None`，从而删除尾节点。

删除指定值节点需找到该节点及其前驱节点，时间复杂度为 $O(n)$ 。遍历链表，比较数据，找到后调整指针。

```

def delete_value(self, key):
1   current = self.head
2   if current is not None and current.data == key: # 如果头节点匹配要删除的值
3       self.head = current.next # 更新 head 指向下一个节点, 删除头节点
4       return
5   prev = None # 用于跟踪前驱节点
6   while current is not None and current.data != key: # 遍历寻找匹配节点
7       prev = current # 保存当前节点为前驱
8       current = current.next # 移动到下一个节点
9   if current is None: # 如果未找到匹配节点
10      return # 直接返回
11   prev.next = current.next # 绕过当前节点, 链接前驱节点的 next 到当前节点的 next
12

```

代码首先检查头节点是否匹配。如果不匹配，则遍历链表，使用 `prev` 变量记录前驱节点。找到匹配节点后，通过设置 `prev.next` 为 `current.next` 来删除当前节点。这确保了链表连续性。

3.5 搜索 (Search)

搜索操作遍历链表，比较每个节点的数据与目标值，时间复杂度为 $O(n)$ 。

```

def search(self, key):
1   current = self.head
2   while current is not None: # 遍历整个链表
3       if current.data == key: # 检查当前节点数据是否匹配
4           return True # 找到匹配, 返回 True
5       current = current.next # 移动到下一个节点
6   return False # 遍历结束未找到, 返回 False

```

代码通过 `while` 循环遍历链表，逐个比较节点数据与 `key`。如果找到匹配，立即返回 `True`；否则，循环结束后返回 `False`。这实现了线性搜索。

4 常见的链表变体

除了单链表，还有其他变体如双向链表和循环链表。双向链表每个节点包含 `prev`、`data` 和 `next` 三部分，允许双向遍历，删除操作更高效，因为不需要寻找前驱节点，但空间开销更大。循环链表的尾节点 `next` 指向头节点，形成环状结构，适用于需要循环访问的场景，如操作系统中的轮询调度或约瑟夫问题。这些变体扩展了链表

的应用范围，但增加了实现复杂度。

5 实战应用：链表在现实世界中的身影

链表在许多实际系统中扮演关键角色。它是高级数据结构的基础，例如栈和队列可以通过链表实现动态大小。在操作系统中，链表用于内存管理，如维护空闲内存块链表。文件系统目录结构常使用链表来组织文件和文件夹。浏览器历史记录功能常用双向链表实现前进和后退操作，因为双向遍历效率高。此外，LRU（最近最少使用）缓存淘汰算法结合哈希表和双向链表，实现 $O(1)$ 时间复杂度的存取操作，提升性能。这些应用展示了链表的实用性和灵活性。

链表作为一种动态数据结构，通过节点和指针实现非连续存储，支持高效插入和删除操作，但访问效率较低。本文详细介绍了单链表的原理和实现，包括遍历、插入、删除和搜索操作，并讨论了常见变体和实际应用。链表是学习更复杂数据结构如树和图的基石，鼓励读者动手实践，例如实现双向链表或解决实际问题如 LRU 缓存。未来，可以探索更多优化和变体，以应对不同场景的需求。