数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 链表

1.1 链表

1.1.1 单向链表（Singly Linked List）

为避免元素的移动，采用线性表的另一种存储方式：链式存储结构。链表是一种在物理上非连续、非顺序的数据结构，由若干结点（node）所组成。

单向链表的每一个结点又包含两部分，一部分是存放数据的数据域 data，另一部分是指向下一个结点的指针域 next。结点可以在运行时动态生成。

```
1 typedef struct Node {
2     dataType data;          // 数据域
3     struct Node *next;      // 指针域
4 } Node;
```

链表的第一个结点被称为头结点，最后一个节点被称为尾结点，尾结点的 next 指针指向空 NULL。

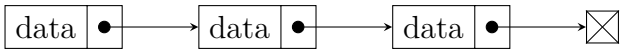


图 1.1: 单向链表

与数组按照下标来随机寻找元素不同，对于链表的其中一个结点 A，只能根据结点 A 的 next 指针来找到该结点的下一个结点 B，再根据结点 B 的 next 指针找到下一个节点 C……

数组在内存中的存储方式是顺序存储，链表在内存中的存储方式则是随机存储。链表采用了见缝插针的方式，每一个结点分布在内存的不同位置，依靠 next 指针关联起来。这样可以灵活有效地利用零散的碎片空间。

1.1.2 双向链表（Doubly Linked List）

那么，通过链表的一个结点，如何能快速找到它的前一个结点呢？要想让每个结点都能回溯到它的前置结点，可以使用双向链表。

双向链表比单向链表稍微复杂一点，它的每一个结点除了拥有 data 和 next 指针，还拥有指向前置结点的 prev 指针。

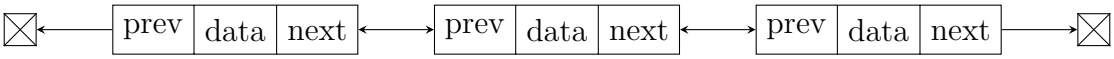


图 1.2: 双向链表

单向链表只能从头到尾遍历，只能找到后继，无法找到前驱，因此遍历的时候不会死循环。而双向链表需要多分配一个指针的存储空间，每个结点有两个指针，分别指向直接前驱和直接后继。

1.1.3 循环链表（Circular Linked List）

除了单向链表和双向链表以外，还有循环链表。对于单向循环链表，尾结点的 next 指针指向头结点。对于双向循环链表，尾结点的 next 指针指向头结点，并且头结点的 prev 指针指向尾结点。

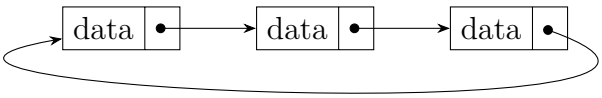


图 1.3: 单向循环链表

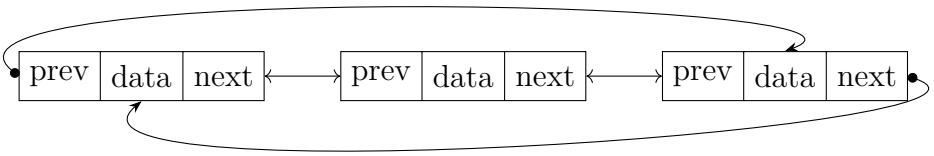


图 1.4: 双向循环链表

1.2 链表的增删改查

1.2.1 查找结点

在查找元素时，链表不像数组那样可以通过下标快速进行定位，只能从头结点开始向后一个一个结点逐一查找。

链表中的数据只能按顺序进行访问，最坏的时间复杂度是 $O(n)$。

查找结点

```
Node* search(List *head, dataType val) {
    // 查找元素位置
    Node *temp = head;
    while(temp) {
        if(temp->data == val) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL;        // 未找到
}
```

1.2.2 更新结点

如果不考虑查找结点的过程，链表的更新过程会像数组那样简单，直接把旧数据替换成新数据即可。

更新结点

```
void replace(List *head, int pos, dataType val) {
    // 找到元素位置
    Node *temp = head;
    for(int i = 0; i < pos; i++) {
```

```
5           temp = temp->next;
6       }
7       temp->data = val;
8   }
```

1.2.3 插入结点

链表插入结点，分为 3 种情况：

尾部插入

把最后一个结点的 next 指针指向新插入的结点。

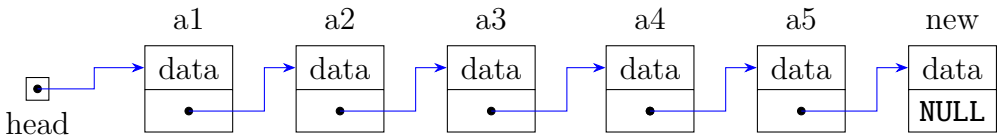


图 1.5: 尾部插入

头部插入

先把新结点的 next 指针指向原先的头结点，再把新结点设置为链表的头结点。

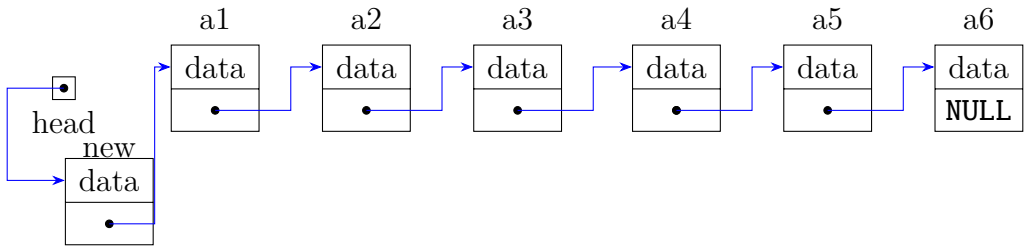


图 1.6: 头部插入

中间插入

先把新结点的 next 指针指向插入位置的结点，再将插入位置的前置结点的 next 指针指向新结点。

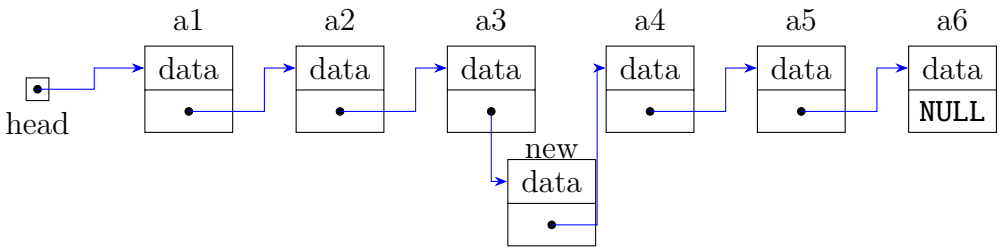


图 1.7: 中间插入

只要内存空间允许，能够插入链表的元素是无穷无尽的，不需要像数组考虑扩容的问题。如果不考虑插入之前的查找元素的过程，只考虑纯粹的插入操作，时间复杂度是 $O(1)$。

1.2.4 删除结点

链表的删除操作也分 3 种情况：

尾部删除

把倒数第二个结点的 next 指针指向空。

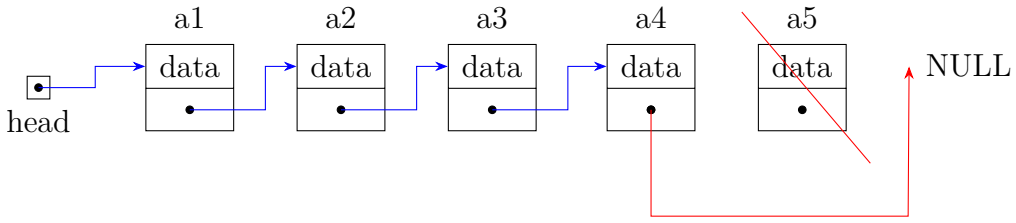
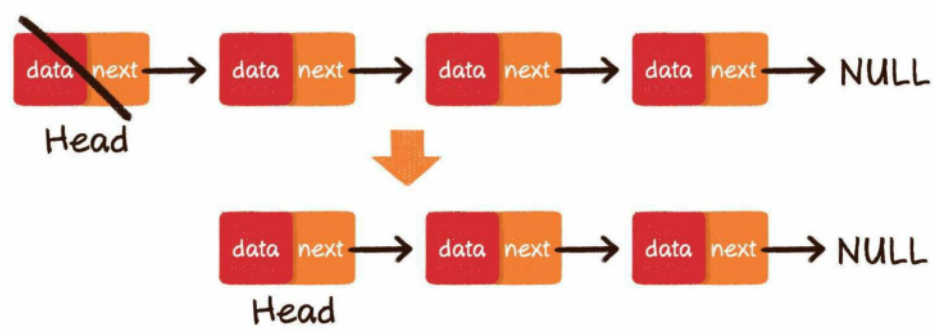


图 1.8: 尾部删除

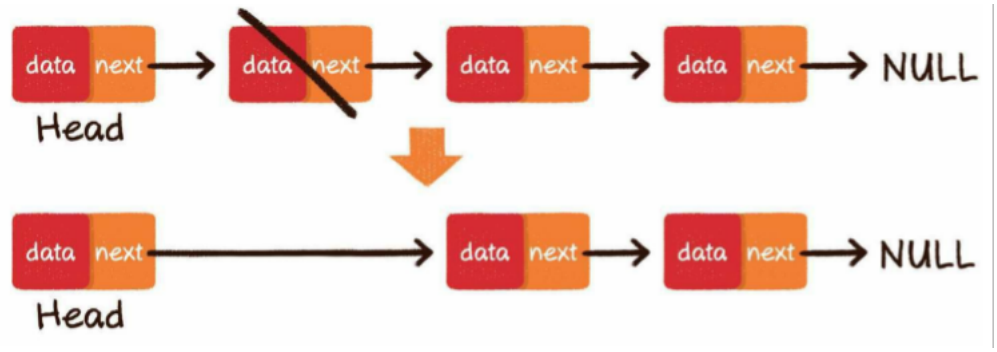
头部删除

把链表的头结点设置为原先头结点的 next 指针。



中间删除

把要删除的结点的前置结点的 next 指针，指向要删除结点的下一个结点。



许多高级语言，如 Java，拥有自动化的垃圾回收机制，所以不用刻意去释放被删除的结点，只要没有外部引用指向它们，被删除的结点会被自动回收。

如果不考虑删除操作之前的查找的过程，只考虑纯粹的删除操作，时间复杂度是 $O(1)$。

1.3 带头结点的链表

1.3.1 带头结点的链表

为了方便链表的插入、删除操作，在链表加上头结点之后，无论链表是否为空，头指针始终指向头结尾。因此对于空表和非空表的处理也统一了，方便了链表的操作，也减少了程序的复杂性和出现 bug 的机会。

插入结点

```
1  void insert(List *head, int pos, dataType val) {
2      Node *newNode = (Node *)malloc(sizeof(Node));
3      newNode->data = val;
4      newNode->next =  NULL;
5
6      // 找到插入位置
7      Node *temp = head;
8      for(int i = 0; i < pos; i++) {
9          temp = temp->next;
10     }
11     newNode->next = temp->next;
12     temp->next = newNode;
13 }
```

删除结点

```
1  void delete(List *head, int pos) {
2      Node *temp = head;
3      for(int i = 0; i < pos; i++) {
4          temp = temp->next;
5      }
6      Node *del = temp->next;
7      temp->next = del->next;
8      free(del);
9      del = NULL;
```

```
10 }
```

1.3.2 数组 VS 链表

数据结构没有绝对的好与坏，数组和链表各有千秋。

比较内容	数组	链表
基本	一组固定数量的数据项	可变数量的数据项
大小	声明期间指定	无需指定，执行期间增长或收缩
存储分配	元素位置在编译期间分配	元素位置在运行时分配
元素顺序	连续存储	随机存储
访问元素	直接访问：索引、下标	顺序访问：指针遍历
插入/删除	速度慢	快速、高效
查找	线性查找、二分查找	线性查找
内存利用率	低效	高效

表 1.1: 数组 VS 链表

数组的优势在于能够快速定位元素，对于读操作多、写操作少的场景来说，用数组更合适一些。

相反，链表的优势在于能够灵活地进行插入和删除操作，如果需要频繁地插入、删除元素，用链表更合适一些。