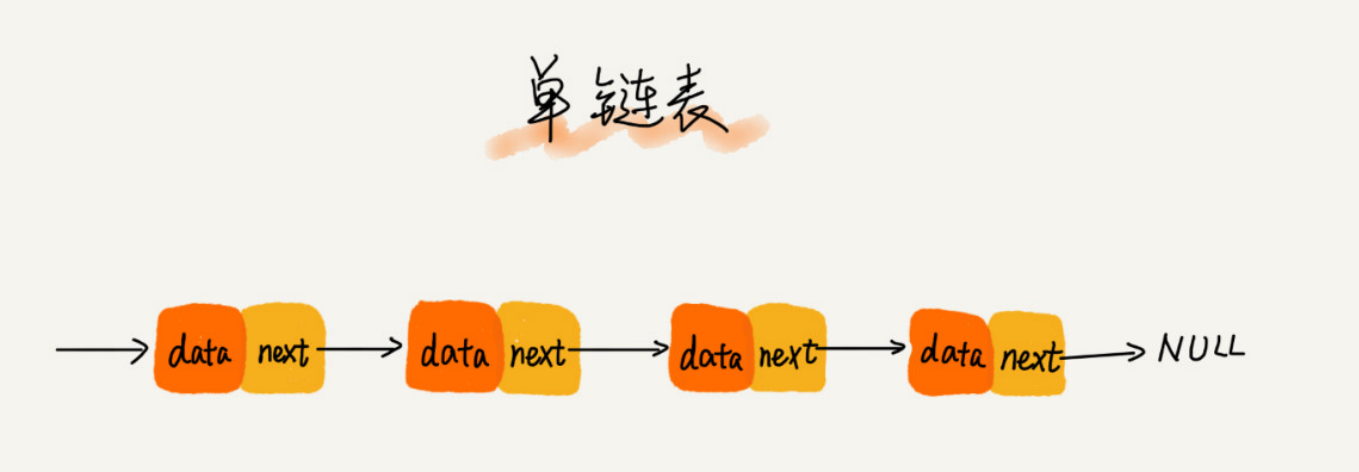
链表

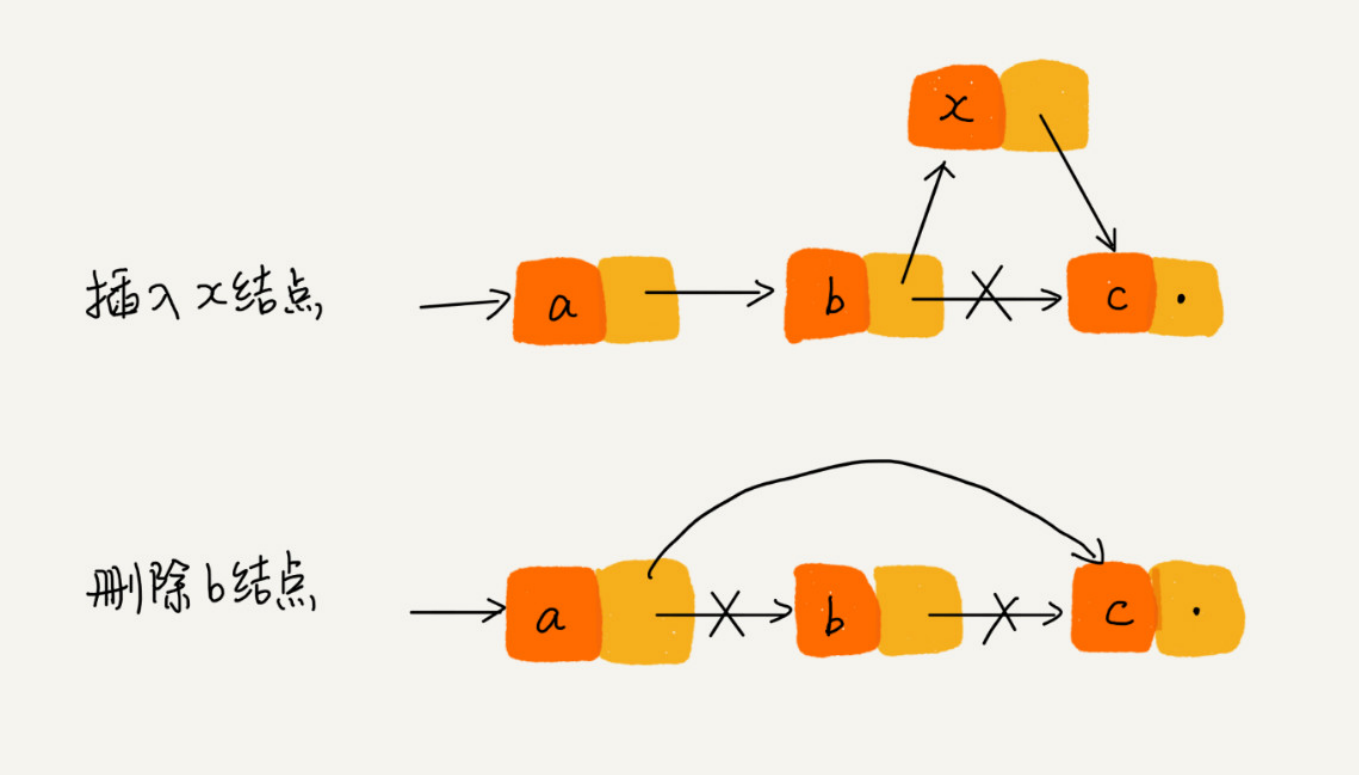
缓存大小一定，当缓存满了之后，如何清理呢，三种策略：先进先出策略FIFO，最少使用策略LFU，最近最少使用策略LRU

如何使用链表实现LRU缓存淘汰策略呢？

首先数组是一块连续的内存空间来存储，对内存要求很高。而链表不需要一块连续的内存空间，通过指针将一堆零散的内存块连起来使用。 链表分为单链表，双链表和循环链表

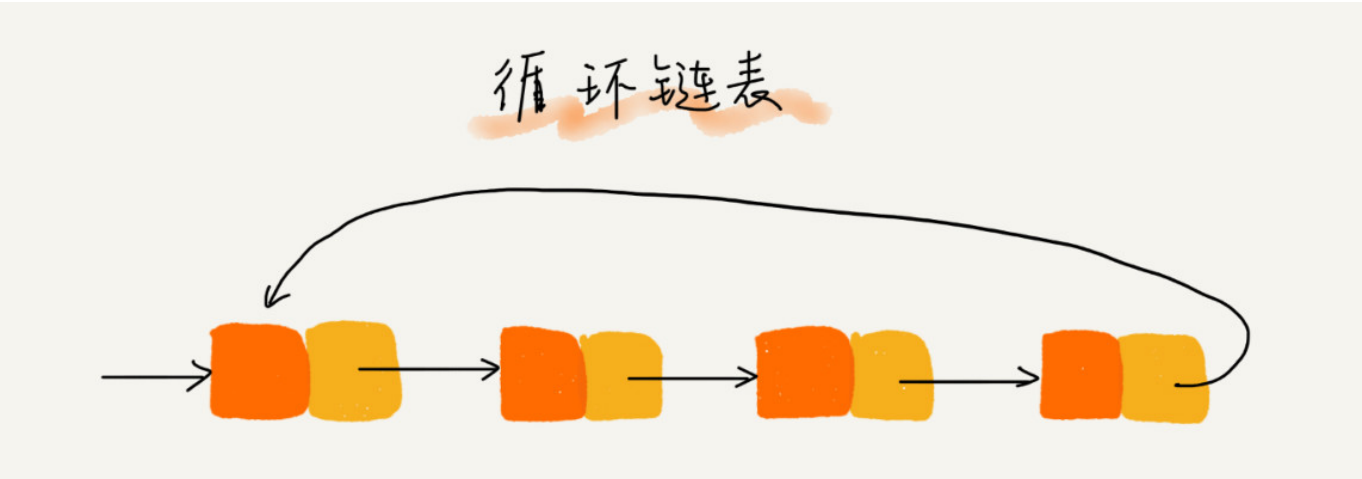


插入和删除

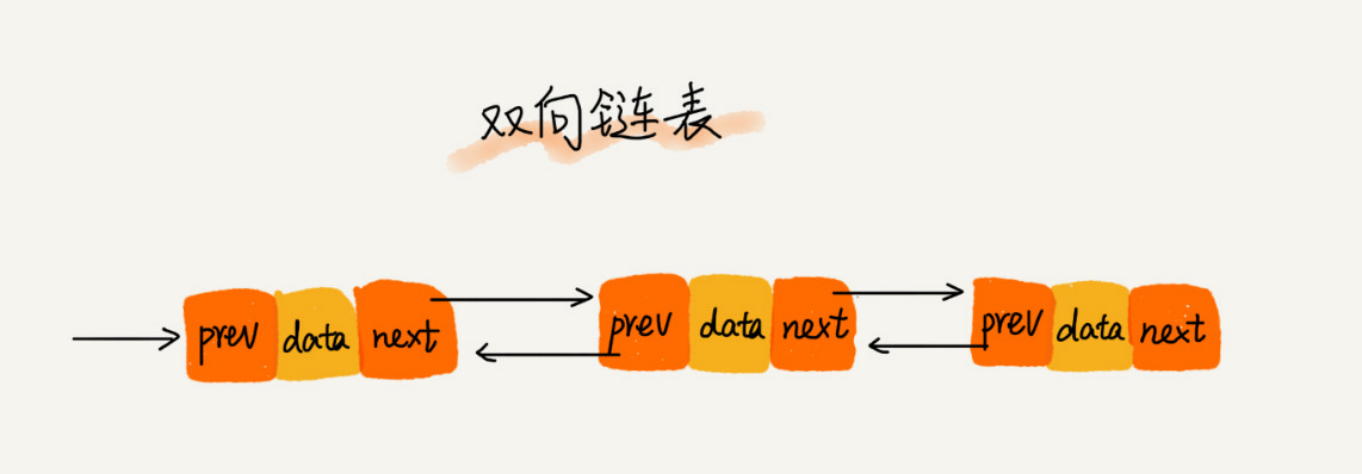


链表的随机访问没有数组好，需要O(n)的时间复杂度

循环链表和双向链表，循环链表是一种特殊的单链表，即尾指针指向了链表的头节点。



双向链表不光只有后向指针还有前向指针



双向链表可以支持 O(1) 时间复杂度的情况下找到前驱结点

尽管单纯的删除操作时间复杂度是 O(1)，但遍历查找的时间是主要的耗时点，对应的时间复杂度为 O(n)。根据时间复杂度分析中的加法法则，删除值等于给定值的结点对应的链表操作的总时间复杂度为 O(n)。

如果我们希望在链表的某个指定结点前面插入一个结点，双向链表比单链表有很大的优势。双向链表可以在 O(1) 时间复杂度搞定，而单向链表需要 O(n) 的时间复杂度

对链表进行频繁的插入、删除操作，还会导致频繁的内存申请和释放，容易造成内存碎片，如果是 Java 语言，就有可能会导致频繁的 GC（Garbage Collection，垃圾回收）

LRU算法的实现： 维护一个有序单链表，越靠近尾部的节点是越早之前访问的，当有一个新的数据比访问时，我们从链表头开始遍历链表

1. 如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据对应的结点，并将其从原来的位置删除，然后再插入到链表的头部。
2. 如果此数据没有在缓存链表中，又可以分为两种情况： 如果此时缓存未满，则将此结点直接插入到链表的头部； 如果此时缓存已满，则链表尾结点删除，将新的数据结点插入链表的头部。

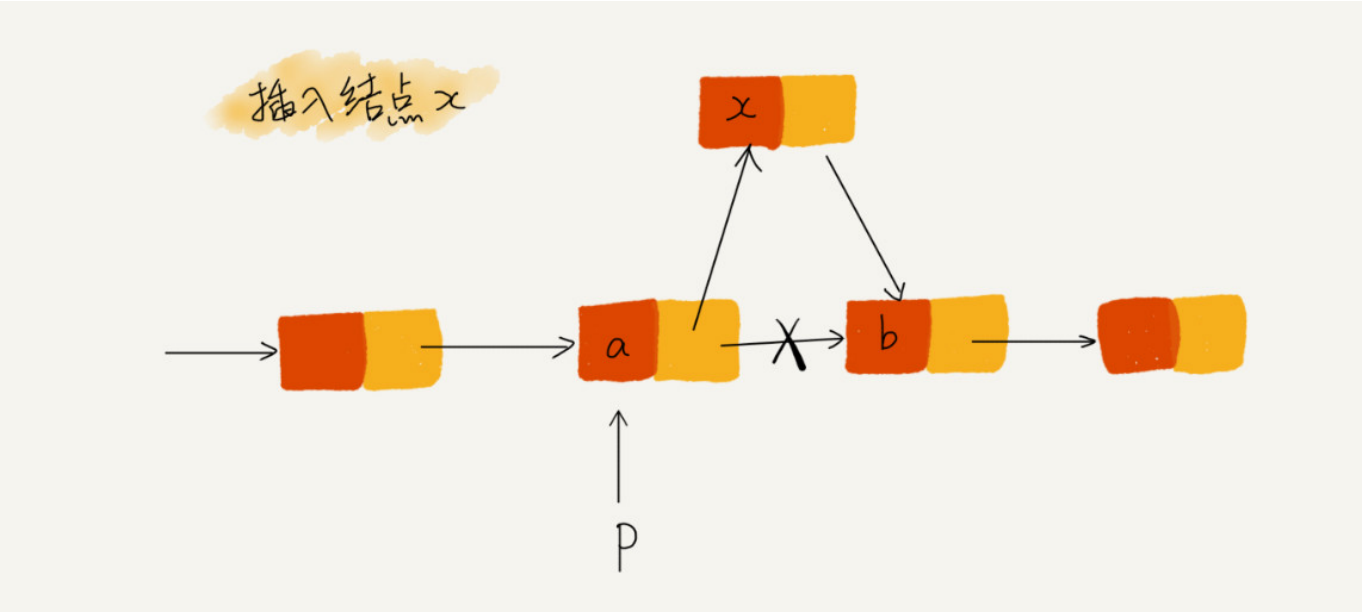
几个写链表代码的技巧

技巧一：理解指针或引用的含义

将某个变量赋值给指针，实际上就是将这个变量的地址赋值给指针，或者反过来说，指针中存储了这个变量的内存地址，指向了这个变量，通过指针就能找到这个变量。 经常会有这样的代码：p->next=q。这行代码是

说，p 结点中的 next 指针存储了 q 结点的内存地址。

技巧二：警惕指针丢失和内存泄漏



使用以下代码就将指针丢失了

```
p->next = x;
x->next = p->next;
```

这样的结果最后是x的指针指向x 插入结点时，一定要注意操作的顺序，如果上述代码，交换位置就是正确的了，x->next指向b，再把结点 a 的 next 指针指向结点 x，这样才不会丢失指针，导致内存泄漏。

技巧三：利用哨兵简化实现难度

如果在结点P后插入新结点，正确的代码为：

```
new_node->next = p->next;
p->next = new_node;
```

但如果要向空链表中插入第一个结点，刚刚逻辑不能用，而应该是这样的

```
if(head == null){
    head = new_node;
}
```

删除结点

```
p->next = p->next->next;
```

最后一个结点删除

```
if(head->next == null){
    head = null;
}
```

针对链表的插入、删除操作，需要对插入第一个结点和删除最后一个结点的情况进行特殊处理

如果我们引入哨兵结点，在任何时候，不管链表是不是空，head 指针都会一直指向这个哨兵结点。我们也把这种有哨兵结点的链表叫带头链表。

技巧四：重点留意边界条件处理

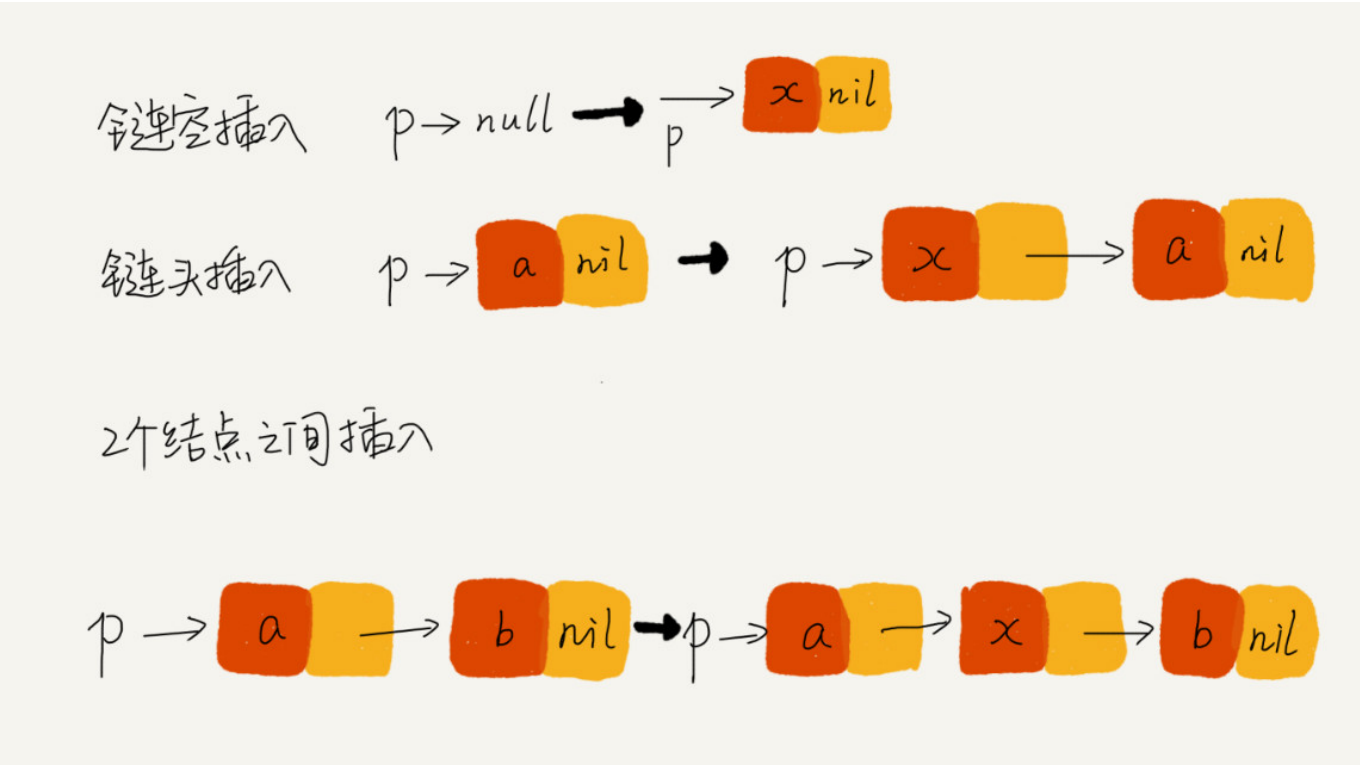
代码在一些边界或者异常情况下，最容易产生Bug。

检查边界条件：（也是其他编程时考虑的方法）

1. 如果链表为空时，代码是否正常工作；
2. 如果链表只含一个结点时，代码是否正常工作；
3. 如果链表只含两个结点时，代码是否正常工作；
4. 代码逻辑在处理头节点和尾节点的时候，代码是否正常工作；

技巧五：举例画图，辅助思考

举例法和画图法



将要做的操作进行基本的画图操作。

技巧六：多写多练，没有捷径

几个常用链表操作，多写多练

1. 单链表反转
2. 链表中环的检测
3. 两个有序的链表合并
4. 删除链表倒数第n个结点
5. 求链表中间结点

```
public class MyLinked {
private Node1 head;
private Node1 last;
MyLinked() throws Exception{
    this(5);
}
MyLinked(int capacity) throws Exception{
    this('a',capacity,false);
}
MyLinked(char c,int capacity,boolean random) throws Exception{
    if(capacity <= 0){
        throw new Exception("链表容量不能为空或负");
    }
    char ch = c;
    for(int i = 0; i < capacity; i++){
        Node1 node = new Node1();
        node.c = ch;
        if(!random){
            ch++;
        }else{
            Random rand = new Random();
            ch+=rand.nextInt(5);
        }

        if(i == 0){
            head = node;
            last = node;
        }else{
            last.node = node;
            last = node;
        }

    }
    last.node = null;
}

public void out(){
    Node1 p = new Node1();
    p = head;
    while (p != null){
        System.out.print(p.c + " ");
```

```
            p = p.node;
        }
        System.out.println();
    }

    public void reverse(){
        Node1 newLast = head;
        Node1 newHead = head;

        Node1 t = head.node;
        while (t != null){
            Node1 node = new Node1();
            node = t;
            t = t.node;
            node.node = newHead;
            newHead = node;
        }
        newLast.node = null; //记得最后链表节点为null
        last = newLast;
        head = newHead;
    }

    //环的检测，快慢指针
    public boolean isCircle(){
        if(head == last){
            return false;
        }
        Node1 p = head;
        Node1 q = head;

        while(q.node != null && q.node.node != null && q != null){
            q = q.node.node;
            p = p.node;
            if(q == p){
                return true;
            }
        }
        return false;
    }

    //有序链表合并
    public MyLinked merge(MyLinked myLinked1,MyLinked myLinked2) throws
Exception {
        MyLinked mergeLinked = new MyLinked(1);
        Node1 head1 = myLinked1.head;
        Node1 head2 = myLinked2.head;
        Node1 mergehead = mergeLinked.head;

        while(head1 != null && head2 != null){
            if(head1.c <= head2.c){
                mergehead.node = head1;
```

```
            head1 = head1.node;
        }else{
            mergehead.node = head2;
            head2 = head2.node;
        }
        mergehead = mergehead.node;
    }
    if(head1 != null){
        mergehead.node = head1;
        mergeLinked.head = mergeLinked.head.node;
        mergeLinked.last = myLinked1.last;
    }else{
        mergehead.node = head2;
        mergeLinked.head = mergeLinked.head.node;
        mergeLinked.last = myLinked1.last;
    }
    return mergeLinked;
}

public void deleteLastNNode(int n){
    Node1 p = new Node1();
    Node1 q = new Node1();
    p.node = q.node = head;
    for(int i = 0; i < n;i++){
        if(p == null){
            System.out.println("链表总长不足:  " + n);
            return;
        }
        p = p.node;
    }
    while (p != last){
        p = p.node;
        q = q.node;
    }
    if(q.node == head){
        head = head.node;
        return;
    }
    q.node = q.node.node;

}

public Node1 middle(){
    Node1 p = head;
    Node1 q = head;
    while(p.node != null && p.node.node != null){
        p = p.node.node;
        q = q.node;
    }
    return q;
}
```

```
public static void main(String[] args) throws Exception{
    MyLinked myLinked = new MyLinked(10);
    myLinked.out();

    System.out.println("是否有环：");
    System.out.println(myLinked.isCircle());
    myLinked.last.node = myLinked.head;
    System.out.println(myLinked.isCircle());
    myLinked.last.node = null;

    System.out.println("求链表中间结点：");
    Node1 middle = myLinked.middle();
    System.out.println(middle.c);

    System.out.println("反转链表：");
    myLinked.reverse();
    myLinked.out();

    System.out.println("链表合并：");
    MyLinked myLinked1 = new MyLinked('a',4,true);
    myLinked1.out();
    MyLinked myLinked2 = new MyLinked('c',2,true);
    myLinked2.out();
    MyLinked mergeLinked = new MyLinked(1);
    mergeLinked = myLinked.merge(myLinked1,myLinked2);
    mergeLinked.out();

    System.out.println("删除倒数第n个结点：");
    myLinked.deleteLastNNode(5);
    myLinked.out();

}
}
```