**线性表的基本概念和实现**

线性表的存储结构有顺序存储和链式存储结构两种。前者被称为顺序表，后者被称为链表。

**顺序表**

顺序表就是把线性表中所有的元素按照其逻辑顺序，依次存储到从指定的存储位置开始的一块连续的存储空间，这样线性表的第一个元素的存储位置就是指定的存储位置，第i+1个的存储位置紧跟着第i个元素的位置

**链表**

在链表存储中，每个结点不仅包含所存元素的信息，还包含元素之间的逻辑关系的信息，如单链表  
中前驱结点包含后继结点的地址信息，这样就可以通过前驱结点的位置找到后继结点的位置

**两种存储结构的比较**

顺序表

* 随机访问特性，位置是确定的
* 要求占用连续的存储空间，存储分配只能预先进行，即静态分配
* 插入操作要移动多个元素

链表

* 不支持随机访问
* 结点中的存储精简利用率稍低一些
* 支持存储空间的动态分配
* 插入操作无需移动元素

**多种链表形式**

**单链表**

**带头结点的单链表**

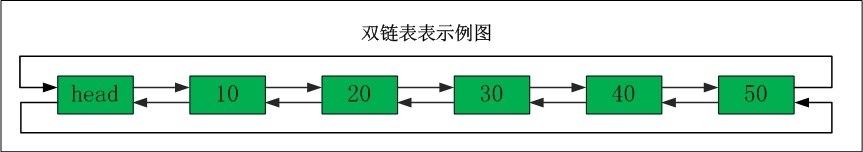
头指针head 指向头结点，头指针值域不包含任何信息，从头结点的后继结点开始存储信息，头指针始终不为null，**当head->next 等于NULL的时候，链表为空**

**不带头结点的单链表**

头指针head直接指向开始结点，**当head为NULL的时候，链表为空**

注意：不论是带有节点的链表还是不带头结点的链表。头指针都指向链表中的第一个结点，；而头结点是带头结点的链表中的第一个结点，只作为链表存在的标志

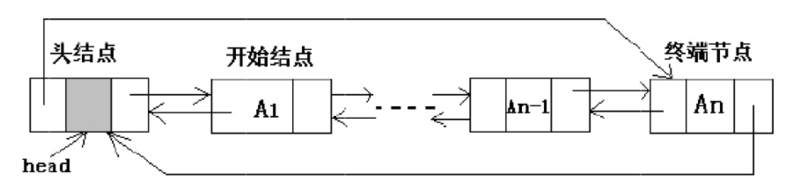
**双链表**

[](https://cloud.8hfq.com/img/20181025100822.png)  
双链表就是在单链表的结点上增添一个指针域，指向当前结点的前驱  
同样 双链表也分为带头结点的双链表和不带头结点的双链表

**循环单链表**

将单链表的最后一个指针域指向链表中的第一个结点即可

**循环双链表**

[](https://cloud.8hfq.com/img/20181025101602.png)  
循环双链表的构造源自双链表，即将终端结点的next指针指向链表中第一个结点，将链表中第一个结点的prior指针指向终端结点。

带头结点的循环双链表当head->next和heaad->prior两个指针都等于head时链表为空。  
不带头结点的循环双链表当head等于null的时候为空。

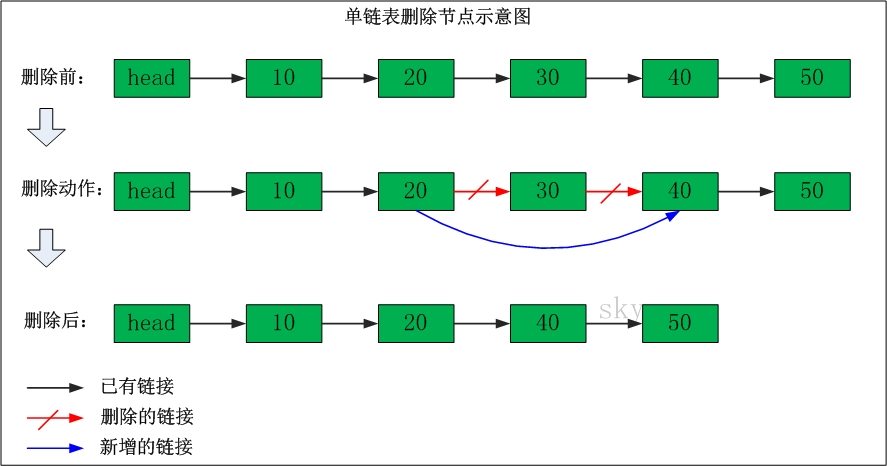
**静态链表**

静态链表借助一位数组来表示，结构体数组中的每一个结点含有两个分量：一个是数据结构元素分量data，另一个是指针分量，指示了当前结点的直接后继结点在数组中的位置

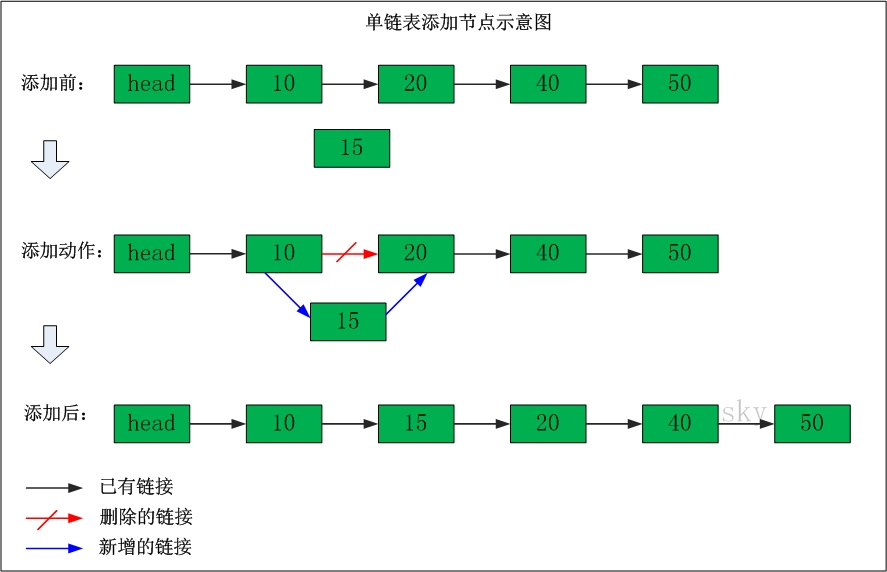
**链表的操作**

**单链表的操作**

**单链表删除结点**

[](https://cloud.8hfq.com/img/20181025104317.png)  
删除”节点30”  
删除之前：”节点20” 的后继节点为”节点30”，而”节点30” 的后继节点为”节点40”。  
删除之后：”节点20” 的后继节点为”节点40”。

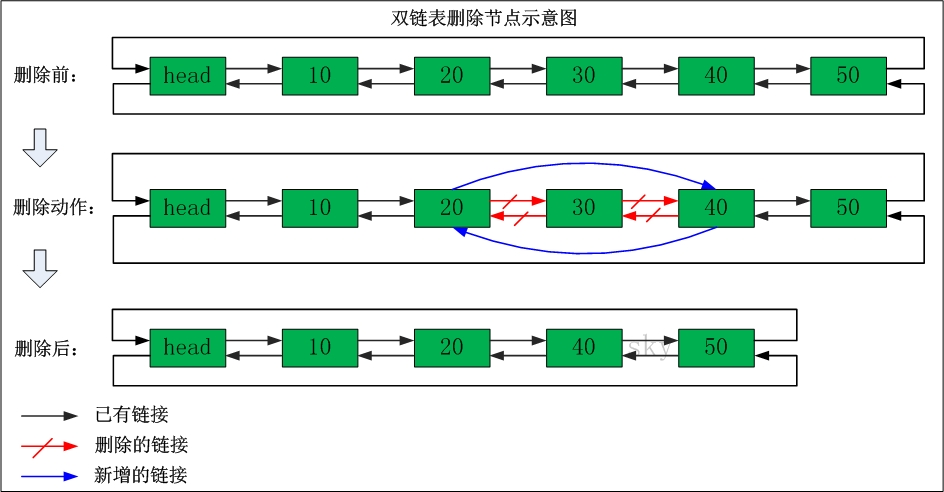
**单链表增加结点**

[](https://cloud.8hfq.com/img/20181025104505.png)  
在”节点10”与”节点20”之间添加”节点15”  
添加之前：”节点10” 的后继节点为”节点20”。  
添加之后：”节点10” 的后继节点为”节点15”，而”节点15” 的后继节点为”节点20”。

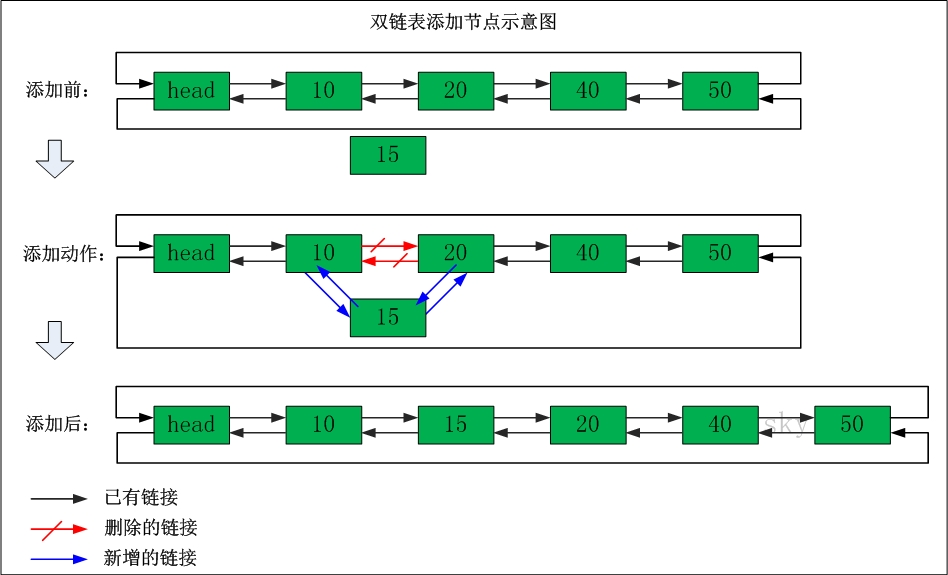
单链表的特点是：节点的链接方向是单向的；相对于数组来说，单链表的的随机访问速度较慢，但是单链表删除/添加数据的效率很高。

**双链表操作**

**双链表删除节点**

[](https://cloud.8hfq.com/img/20181025104559.png)  
删除”节点30”  
删除之前：”节点20”的后继节点为”节点30”，”节点30” 的前继节点为”节点20”。”节点30”的后继节点为”节点40”，”节点40” 的前继节点为”节点30”。  
删除之后：”节点20”的后继节点为”节点40”，”节点40” 的前继节点为”节点20”。

**双链表添加节点**

[](https://cloud.8hfq.com/img/20181025104726.png)  
在”节点10”与”节点20”之间添加”节点15”  
添加之前：”节点10”的后继节点为”节点20”，”节点20” 的前继节点为”节点10”。  
添加之后：”节点10”的后继节点为”节点15”，”节点15” 的前继节点为”节点10”。”节点15”的后继节点为”节点20”，”节点20” 的前继节点为”节点15”。

**栈**

**栈的基本概念**

* 栈是一种只能在一端进行插入或删除操作的线性表。其中允许进行插入或者删除操作的一端称为栈顶（TOP）。栈顶由一个称谓栈顶指针的位置指示器（其实就是一个变量），对于顺序栈，就是记录栈顶元素所在数组位置标号的一个整形变量，对于链式栈就是记录栈顶元素所在的结点地址的指针。另一端被称为栈底，栈底是固定不变的。
* 栈的特点，先进后出（FILO）
* 栈的存储结构，可用顺序表和链表来存储栈，分为顺序栈和链式栈。 即栈的本质上是线性表
* 当n各元素以某种顺序进栈，并且可以在任意时刻出栈，所获得的元素排列的数目N满足函数  
  N=1/(n+1)Cn2nN=1/(n+1)C2nn
* 栈通常包括的三种操作：push、peek、pop。  
  push – 向栈中添加元素。  
  peek – 返回栈顶元素。  
  pop – 返回并删除栈顶元素的操作。

**顺序栈（栈的数组实现）**

顺序栈的特殊状态和操作

1. 栈空状态  
   st.top==-1 有的书上规定st.top ==0为栈空条件
2. 栈满状态  
   st.top==maxSize-1 maxSize为栈中最大元素的个数，则maxSize-1 为栈满时栈顶元素在树组中的位置
3. 非法状态（上溢和下溢）栈满继续进入栈就会出现上溢状态，栈空继续出栈就会出现下溢状态
4. 进栈操作 ++(st.top);st.data[st.top]=x;
5. 出栈操作 x=st.data[st.top];–(st.top);

**链栈**

链栈的特殊状态和操作

1. 栈空状态 lst->next==NULL
2. 栈满状态 不存在栈满的情况
3. 元素进栈操作 p->next=lst->next; lst->next=p
4. p=lst->next;x=p->data;lst->next=p->next;free(p)

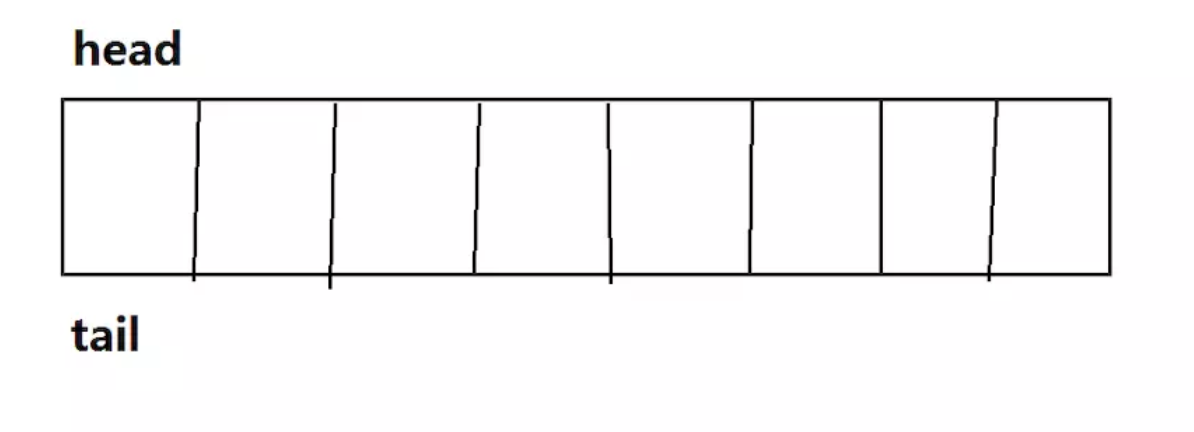
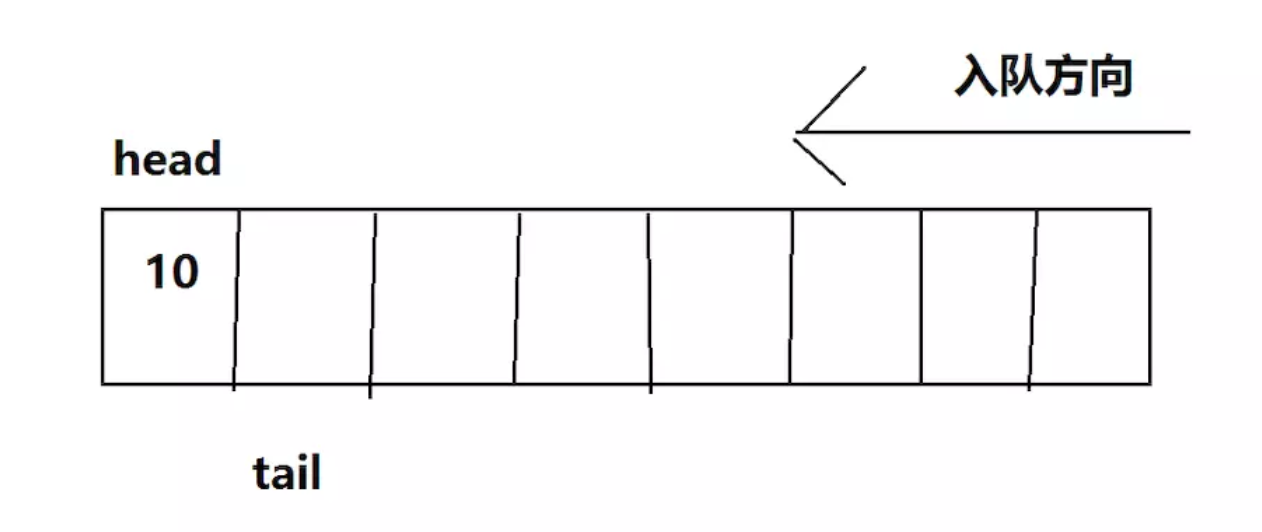
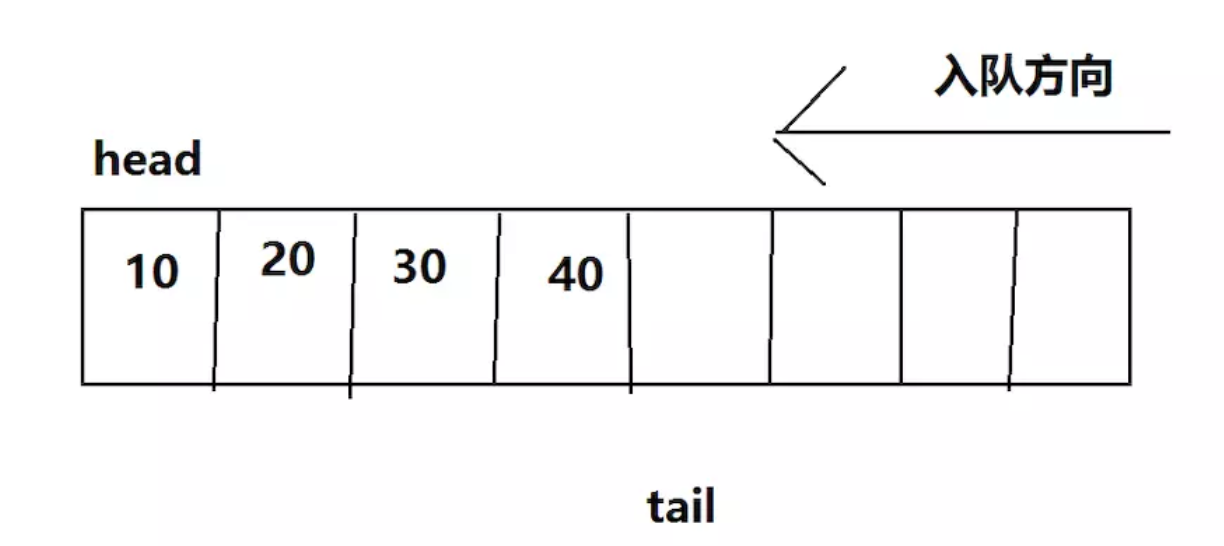
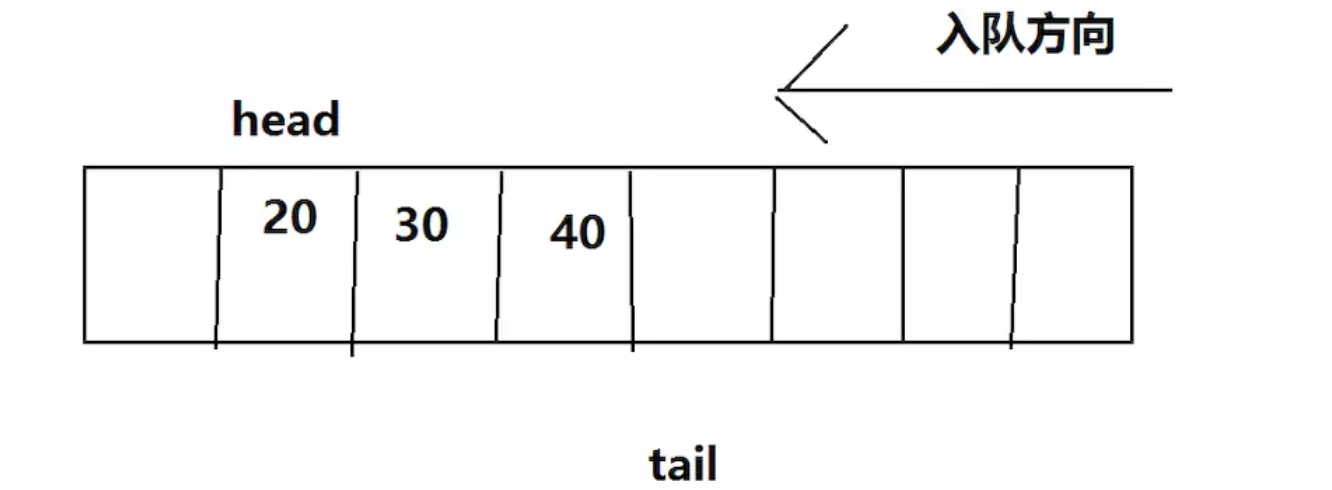
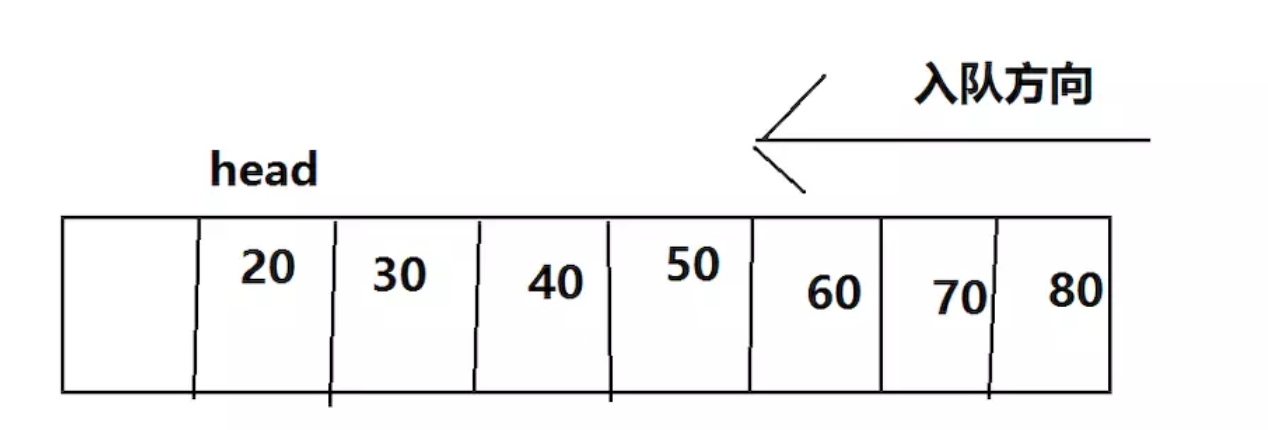
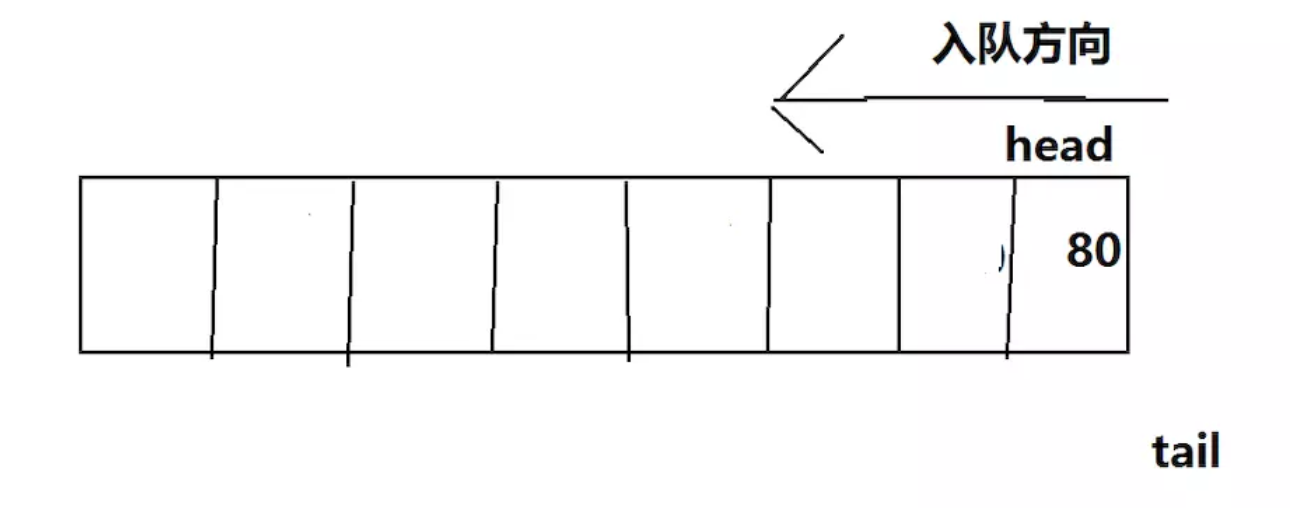
**队列**

**队列的基本概念**

* 队列也是一种受限制的线性表，其显示为仅允许在表的一端进行插入，在表的另一端进行删除，可进行插入的一端是队尾，颗进行删除的一头是队头。
* 队列的特点，先进先出（FIFO）
* 可用顺序表和链表来存储队列，队列按存储结构颗分为顺序队和链队两种。

**顺序队**

**循环队列**

为了深刻体会到循环队列这个结构优于非循环队列的地方，我们将首先介绍数组实现的非循环队列结构。队列这种数据结构，无论你是用链表实现，还是用数组实现，它都是要有两个指针分别指向队头和队尾。在我们数组的实现方式中，用两个int型变量用于记录队头和队尾的索引。  
[](https://cloud.8hfq.com/img/20181111162742.png)  
一个队列的初始状态，head和tail都指向初始位置（索引为0处）。head永远指向该队列的队头元素，tail则指向该队列最后一个元素的下一位置，当有入队操作时：  
[](https://cloud.8hfq.com/img/20181111162827.png)  
[](https://cloud.8hfq.com/img/20181111162843.png)  
当有出队操作时：  
[](https://cloud.8hfq.com/img/20181111163104.png)  
当遇到出队操作时，head会移向下一元素位置。当然，对于这种方式入队和出队，队空的判断条件显然是head=tail，队满的判断条件是tail=array.length（数组最后一个位置的下一位置）。显然，这种结构最致命的缺陷就是，tail只知道向后移动，一旦到达数组边界就认为队满，但是队列可能时刻在出队，也就是前面元素都出队了，tail也不知道。例如：  
[](https://cloud.8hfq.com/img/20181111163222.png)  
此时tail判断队满，我们暂时认为资源利用是可以接受的，但是如果接下来不断发生出队操作：  
[](https://cloud.8hfq.com/img/20181111163253.png)  
此时tail依然通过判断，认为队满，不能入队，这时数组的利用率我们是不能接受的，这样浪费很大。所以，我们引入循环队列，tail可以通过mode数组的长度实现回归初始位置，下面我们具体来看一下。  
按照我们的想法，一旦tail到达数组边界，那么可以通过与数组长度取模返回初始位置，这种情况下判断队满的条件为tail=head

**循环队列的特殊状态和操作**

1. 队空状态 qu.rear=qu.front
2. 队满状态 （qu.rear+1）%maxSize == qr.front
3. 元素x进队操作（移动队尾指针）  
   qu.rear =(qu.rear+1) %maxSize;qu.date[qu.rear]=x;
4. 元素x出队操作（移动队首指针）  
   qu.front =(qu.front+1)%maxSize;x=qu.data[qu.front];

**链队**

链队的特殊状态和操作

1. 队空状态 lqu->rear=NULL 或者 lqu->front=NULL
2. 队满状态 不存在队满状态 （假设内存无限大的情况下）
3. 元素进队操作 lqu->rear->next->p;lqu->rear=p;
4. 元素出队操作 p=lqu->front;lqu->front=p->next;x=p->data;free(p)

**向量**

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。

1. 顺序序列  
   顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。
2. 动态数组  
   支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算述进行该操作。操供了在序列末尾相对快速地添加/删除元素的操作。
3. 能够感知内存分配器的（Allocator-aware）  
   容器使用一个内存分配器对象来动态地处理它的存储需求。

vector的扩充机制：按照容器现在容量的一倍进行增长。vector容器分配的是一块连续的内存空间，每次容器的增长，并不是在原有连续  
的内存空间后再进行简单的叠加，而是重新申请一块更大的新内存，并把现有容器中的元素逐个复制过去，然后销毁旧的内存。这时原有指向旧内存空  
间的迭代器已经失效，所以当操作容器时，迭代器要及时更新。

**抽象数据结构（ADT）**

一个抽象数据结构（Abstract Data type）可以看做一些数据对象以及附加在这些对象上的操作的集合。  
对于栈来说，数据对象集为存储在栈内的数据元素  
操作集为元素进栈，元素出栈，判断栈是否为空等操作

ADT 栈(stack)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | Data 同线性表。元素具有相同的类型，相邻元素具有前驱和后堆关系。 Operation InitStack ( \*S )：初始化操作.建立一个空栈S。 DestroyStack ( \*S )：若栈存在，則销毁它。 ClearStack (\*S)：将栈清空。 StackEmpty ( S ):若栈为空，返回true,否則返回 false。 GetTop (S,\*e)：若栈存在且非空，用e返回S的栈顶元素。 Push (\*S,e)：若栈S存在，插入新元素e到栈S中并成为栈頂元素。 Pop (\*S,\*e)：删除栈S中栈顶元素，并用e返回其值。 StackLength (S)：返回回栈S的元素个数。 endADT |

ADT 队列(Queue)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | Data  同线性表。元素具有相同的类型，相邻元素具有前驱和后继关系。  Operation  InitQueue(\*Q)：初始化操作，建立一个空队列Q。  DestroyQueue(\*Q)：若队列Q存在，則销毀它。  ClearQueue(\*Q)：将队列 Q 清空。  QueueEmpty(Q)：若队列Q为空，送回true,否則退回false。  GetHead(Q, \*e)：若队列Q存在且非空，用e返因队列Q的队头元素。  EnQueue(\*Q,e)：若队列Q存在，插入新元素e到队列Q中并成为队尾元素。   DeQueue(\*Q, \*e)：刪除队列Q中队头元素，并用e返回其值。   QueueLength(Q)：送回队列Q的元素个教。 endADT |