

# 线性链表

---

链式存储结构不要求逻辑上相邻的元素在物理位置上也相邻，因此他没有顺序存储结构所具有的缺点，但同时也失去了顺序表可随机存取的优点

## 特点

---

用一组任意的存储单元存储线性表的数据元素

这组存储单元可以是连续的，也可以是不连续的

因此，为了表示每个数据元素 $a_i$ 与其直接后继数据元素 $a_{i+1}$ 之间的逻辑关系，对于数据元素 $a_i$ 来说除了存储器本身的信息还需要存储指示其直接后继的信息

这两部分信息组成数据元素 $a_i$ 的存储映像，称为结点（node）

它包括两个域：

- 数据域；存储数据元素信息的域
- 指针域；存储直接后继存储位置的域

指针域中存储的信息称作指针或链

$n$ 个结点链结成一个链表，即为线性表的链式存储结构

- $(a_1, a_2, \dots, a_n)$

因为只包含一个指针域，又称为单链表

```
typedef {
    ElementType      data; //数据域
    struct ListNode *next; //指针域；指向其直接后继
} ListNode, *LinkedList;
```

整个链表的存取必须从头指针开始进行，头指针指示链表中第一个元素

同时，因为最后一个元素没有直接后继，则线性链表中最后一个结点的指针域为“NULL”

## 链表获取元素

---

时间复杂度  $O(n)$

因为其存储结构的特点，不能像顺序表那样随意访问

只能从头结点开始，通过结点的指针域访问其直接后继，直到找到目的下标的结点

```

ElementType getElement(LinkedList list,int index){

    //首先指向头结点
    ListNode *element = list;

    //循环几次就指向第几个结点
    //从0到index(用户要找的下标)
    for(int i = 0;i <= index;i++){
        //还没找到index的结点
        //但是当前结点为空 直接返回ERROR
        if(element == NULL)
            return NULL;

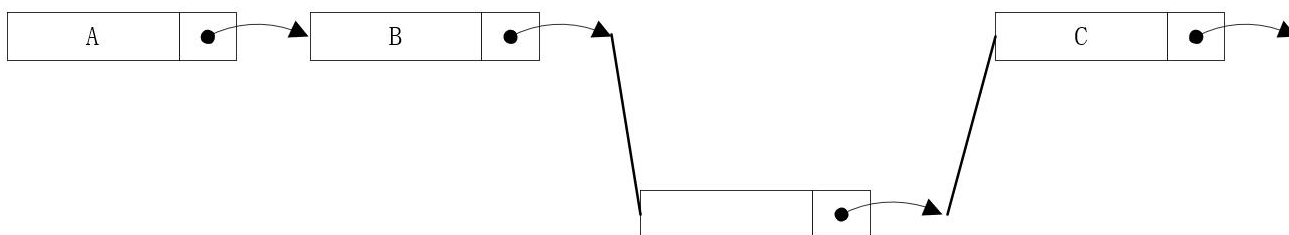
        //指向下一个结点
        element = element->next;
    }

    //返回下标为index的结点值
    return element->data;
}

```

## 链表插入结点

跟顺序表相比，链表插入元素时不需要后移元素，只需要修改结点的指针即可



所以其插入时的时间复杂度更低，为 $O(1)$

Code

```

Status insertElement(LinkedList &list,int index,ElementType newElement){

    //下标[index]不合法
    if(index < 0 || index > getLength(list)){
        return INFEASIBLE;
    }
}

```

```

}

//先指向头结点
LinkedList *element = list;

//指向第[i - 1]个结点
for(int i = 0; i <= (index - 1); i++){
    //当前结点为空 直接返回ERROR
    if(element == NULL)
        return ERROR;

    //指向下一个结点
    element = element->next;
}

//申请新结点的存储空间
LinkedList *newNode = (LinkedList *)malloc(sizeof(LinkedList));

//如果新结点分配空间失败 那么直接exit
if(newNode == NULL)
    exit(1);

newNode->data = newElement;

//新结点的下一个结点是下标为[index-1]结点的下一个[index]结点
newNode->next = element->next;

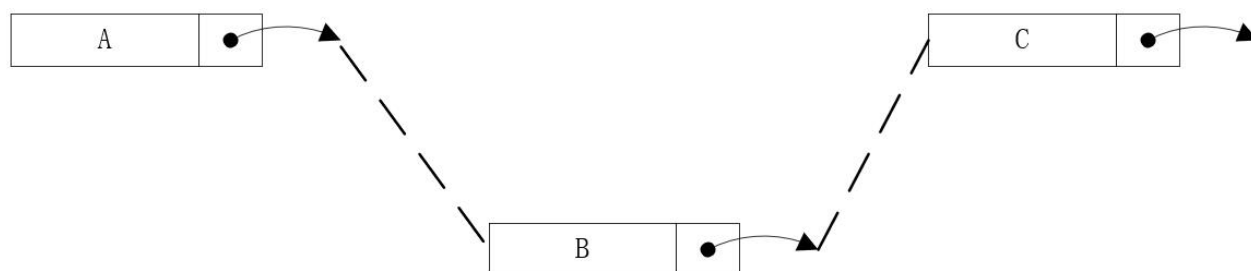
//新结点是下标为[index - 1]结点的后继
element->next = newNode;

return OK;
}

```

## 链表删除结点

时间复杂度 $O(1)$



Code

```
Status deleteElement(LinkedList &list,int index,ElementType &element){

    //下标[index]不合法
    if(index < 0 || (index > getLength(list))){
        return INFEASIBLE;
    }

    //先指向头结点
    ListNode *currentElement = list;

    //指向第[i - 1]个结点
    for(int i = 0;i <= (index - 1);i++){
        //当前结点为空 直接返回ERROR
        if(element == NULL)
            return ERROR;

        //指向下一个结点
        currentElement = currentElement->next;
    }

    //没有第i个结点
    if(currentElement->next == NULL){
        return OVERFLOW;
    }

    //把要删除的结点通过参数返回给调用者
    element = currentElement->next->data;

    //直接越过要删除的结点指向其指针域
    currentElement->next = currentElement->next->next;

    return OK;
}
```

## 链表合并

---

## 循环链表 (Circular linked list)

---

循环链表是另一种形式的链式存储结构

它的特点是表中最后一个结点的指针指向头结点，整个链表形成一个环

## 双向链表 (Double linked list)

---

单链表中，一个结点只有一个指向直接后继的指针域；也就是说只能从前往后找，不能从后往前找

单链表的寻找后继nextElement时间复杂度为 $O(1)$ ；直接调用`element->next`即可

单链表的寻找前驱priorElement时间复杂度却为 $O(n)$ ；只能从头结点开始找；

为了克服单链表这种单向性的缺点，可利用双向链表；

其有两个指针域，一个指向直接前驱，一个指向直接后继