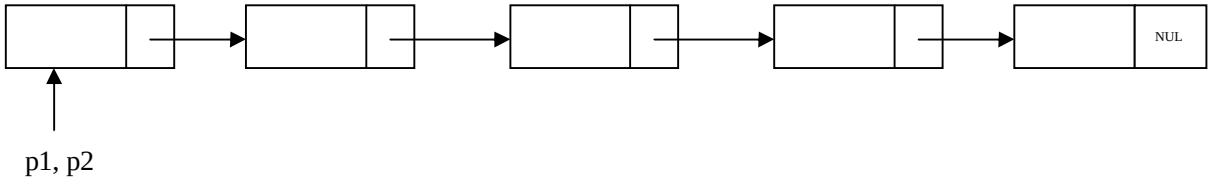
麦洛克菲-常见算法归纳总结

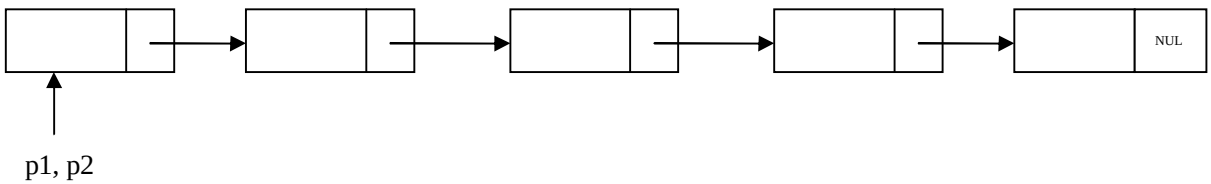
1. 两个指针跑步法

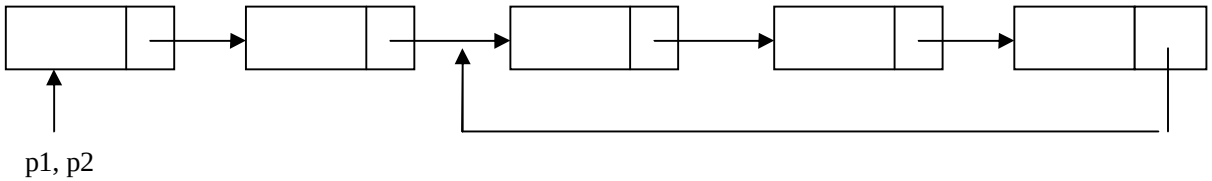
题目 1：计算一个链表的中间节点。




```
node * find_list_middle_node(node * list)
{
    if(list == NULL || list->next == NULL)
    {
        return list;
    }
    node *p1 = list;
    node *p2 = list;
    while(p1&&p2&&p2->next)
    {
        p1=p1->next;
        p2=p2->next->next;
    }
    if(p2->next==NULL || p2==NULL)
    return p1;
}
```

题目 2：判断一个单向链表是否存在循环。

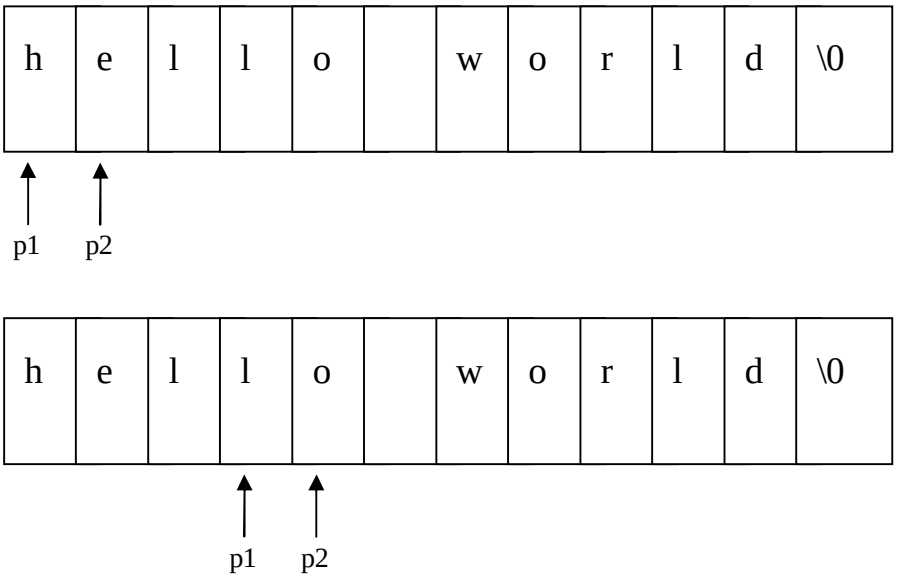





```
int find_loop(node *head)
{
	node *p = NULL;
	node *q = NULL;
	if (head == NULL)
		return 0;

	p = head;
	q = head->next;
	while (p!=NULL &&
		q!=NULL&&q->next!=NULL&&p!=q)
	{
		p = p->next;
		q = q->next->next
	}
	if (p==q)
		return 1;
	else
		return 0;
}
```

题目 3：从一个字符串中删除某一个特定的字符。

如：”hello world”,’o’→”hell wrld”




```
char* del_char(char *str,char ch)
{
	char *p1=str,*p2=str;
	if(NULL==str)
	{
		return NULL;
	}
	while(*p2!='\0')
	{
		if(*p2!=ch)
		{
			*p1++=*p2++;
		}
		else
		{
			p2++;
		}
	}
	*p1='\0';
	return str;
}
```

题目 4：将一个字符串按照单词顺序逆置，如 hello this world→world this hello

题目 5：将一个 IP 地址字符串转化为 32 位整数。如 255.255.255.255→0xffffffff

2. 递归法

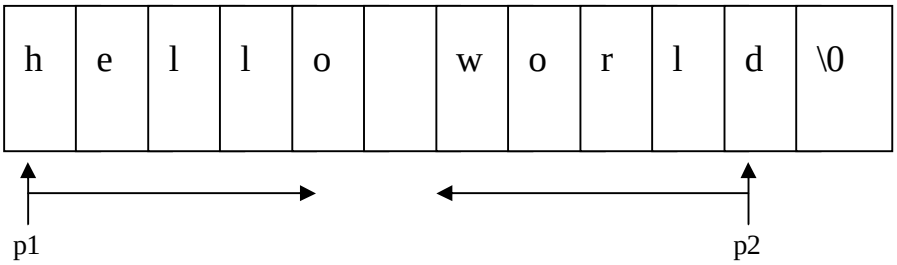
题目：用一条语句，不声明临时变量，实现 strlen 来计算一个字符串的长度。

如 strlen(“hello world”)=11

```
size_t    strlen (const char * str )//常规方法
{
        const char *eos = str;
        while( *eos++ ) ;
        return( eos - str - 1 );
}

size_t strlen(const char *str)//递归方法
{
    return (str==NULL || *str==’\0’)?0:1+strlen(str+1);
}
```

题目：将一个字符串逆置。

比如:”hello world”→”dlrow olleh”




```
void reverse_str(char* str, unsigned int len)//常规方法
{
    if (NULL == str)
    {
        return;
    }
    char *p1=str;
    char *p2=str+len-1;
    while(p1<p2)
    {
        char c = *p1;
```

```
            *p1 = *p2;
            *p2 = c;
            p1++;
            p2--;
        }
}

void reverse_str(char *str, size_t len)//递归方法
{
        if(len==0 || len==1 || str == NULL || *str=='\0')
        {
                return;
        }

        char tmp = str[0];
        str[0]=str[len-1];
        str[len-1]=tmp;

        reverse_str(str+1, len-2);

}
```

3. 熟练掌握循环

1. 数组

```
for (int i = 0; i < n; i++)
{
    printf("%d\n", a[i]);
}
```

2. 字符串

```
while (*str != '\0')
{
    str++;
}
```

3. 链表

```
while (p)
{
    p = p->next;
```

```
}
```

4．栈空/栈满/队空/队满

```
while(栈非空)
{
    出栈；
    操作出栈元素
}
```

5．指针大小比较

```
while (pStart < pEnd)
{
    pStart++;
}
```

6．数为正

```
while (size--)
{
}
do
{
}while(size--)
```

7．反复循环直到条件成熟

```
while(1)
{
    if (something is true)
        break;
}
```

4. 严进宽出

对输入的参数进行有效性的判断，排除不合理或者非法输入。
指针判断是否为 NULL。
Buffer 的长度是否合理范围

```
void reverse_str(char *str, size_t len)
{
    if(len==0 || len==1 || str == NULL || *str=='\0')
    {
```

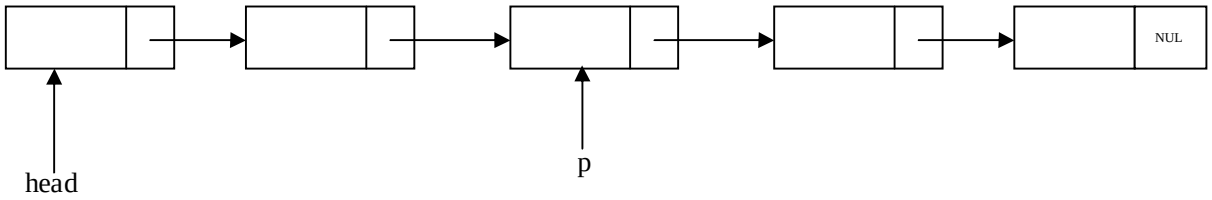
```
        return;
    }

    char tmp = str[0];
    str[0]=str[len-1];
    str[len-1]=tmp;

    reverse_str(str+1, len-2);

}
```

5. 效率（时间空间复杂度）

时间复杂度

```
// 第一个
for (i=0; i<N; i++)
{
    if (condition)
        DoSomething();
    else
        DoOtherthing();
}
// 第二个
if (condition)
{
    for (i=0; i<N; i++)
        DoSomething();
}
else
{
    for (i=0; i<N; i++)
        DoOtherthing();
}
```

题目：删除一个链表结点，要求效率尽可能高。




```
bool delete_node(node *&head, node *p)
{
    if(!p || !head)
        return false;

    if(p->m_pNext != NULL)    //不是尾指针
    {
        node *del = p->m_pNext;
        p->m_nValue = del->m_nValue;
        p->m_pNext = del->m_pNext;
        delete del;
        del = NULL;
    }
    else if(head == p)        //是尾指针，同时只有一个结点
    {
        delete p;
        head = NULL;
    }
    else                      //是尾指针，同时有多个结点
    {
        node *tmp = NULL, *pre = head;
        while(pre->m_pNext != p)
        {
            pre = pre->m_pNext;
        }
        delete p;
        p = NULL;
        pre->m_pNext = NULL;
    }
    return true;
}
```

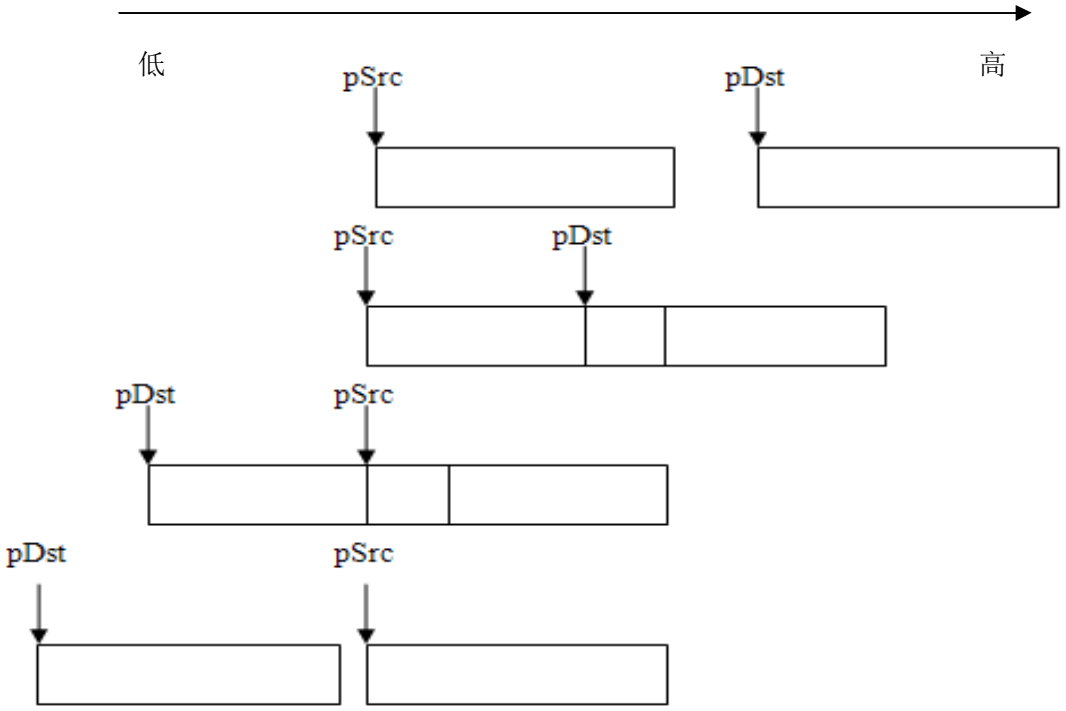
空间复杂度

```
void reverse_str1(char *str, size_t length)
{
```

```
    size_t len = length;

    char *buff = (char *)malloc(len+1);

    if(buff==NULL)
        return;
    memset(buff, 0,len+1);

    for(int i=0;len>0;len--,i++)
    {
        buff[i]=str[len-1];
    }

    for(int i=0;i<length;i++)
    {
        str[i]=buff[i];
    }
    free(buff);

}

void reverse_str2(char* str, unsigned int len)
{
    if (NULL == str)
    {
        return;
    }
    for (int i = 0; i < len /2; i++)
    {
        char c = str [i];
        str [i] = str [len -1- i ];
        str [len -1- i] = c;
    }
}
```

6.边界考虑

```
void memcpy(void *pDst,const void *pSrc, size_t size)
```



pSrc 与 pDst 的相对位置关系

```
void memcpy(void *pDst,const void *pSrc, size_t size)
{
    assert(pDst != NULL);
    assert(pSrc != NULL);
    /* pSrc 与 pDst 共享同一块内存区域 */
    if((pSrc<pDst) && ((char*)pSrc+size > pDst))
    {
        char *pstrSrc= (char *)pSrc + size -1;
        char *pstrDst = (char *)pDst + size -1;
        / * 从尾部逆向拷贝 */
        while(size--)
            *pstrDst -- = *pstrSrc--;
    }
    else
    {
        char *pstrSrc= (char *)pSrc ;
        char *pstrDst = (char *)pDst ;
        /* 从起始部正向拷贝 */
        while(size--)
            *pstrDst++ = *pstrSrc++;
    }
}
```

当从链表中插入或者删除一个结点时，就必须考虑该结点是否为首结点，尾结点等特

殊情况。此外，在打开文件，分配内存等时候，都要判断该操作是否成功