二分查找

最大化查找（查找第一个<=q的数的下标）

```
#include <bits/stdc++.h>
using namespace std;
int find(int q){
    int l = 0,r = n + 1;
    while(l + 1 < r){
        int mid =l + (r - l) / 2;
        if(a[mid] <= q) l = mid;
        else r= mid;
    }
    return l;
}
```

最小化查找(查找第一个>=q的数的下标)

```
int find(int q){
    int l = 0,r = n + 1;
    while(l + 1 < r){
        int mid =l + (r - l) / 2;
        if(a[mid] >= q) r = mid;
        else l= mid;
    }
    return r;
}
```

set的运用

```
st.insert(x)        //st中插入一个元素x
st.size()       //返回集合set中元素的个数
st.begin()      //set集合起始地址
st.end()        //set集合结束的地址
set<Type>::iterator it; //定义一个对应类型的迭代器  来遍历set
set遍历: for(auto it = set.begin();auto != set.end(),it++)
set.find(x)找不到就返回st.end();
```

map运用（当成字典来用就好了）

```
map<char, int> q;
q.insert(pair<int, string>(111, "kk"));
遍历map
for(pair<char,int> x:q)
```

```
{
    cout<<x.first<<' '<<x.second<<'\n';
}
```

求一个数二进制1的个数

```
void count(int m){
    int tmp = 0;
    while(m > 0){
        tmp++;
        m &= (m - 1);
    }
}
```

构建链表

构建链表时，使用指针的部分比较抽象，光靠文字描述和代码可能难以理解，建议配合作图来理解。

单向链表

单向链表中包含数据域和指针域，其中数据域用于存放数据，指针域用来连接当前结点和下一节点。

"C++"

```
struct Node {
    int value;
    Node *next;
    };
```

双向链表

双向链表中同样有数据域和指针域。不同之处在于，指针域有左右（或上一个、下一个）之分，用来连接上一个结点、当前结点、下一个结点。

```
struct Node {
    int value;
    Node *left;
    Node *right;
};
```

向链表中插入（写入）数据

单向链表

流程大致如下：

1. 初始化待插入的数据 `node`；
2. 将 `node` 的 `next` 指针指向 `p` 的下一个结点；
3. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

代码实现如下：

"C++"

```
void insertNode(int i, Node *p) {
    Node *node = new Node;
    node->value = i;
    node->next = p->next;
    p->next = node;
}
```

单向循环链表

将链表的头尾连接起来，链表就变成了循环链表。由于链表首尾相连，在插入数据时需要判断原链表是否为空：为空则自身循环，不为空则正常插入数据。

大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `next` 指针和 `p` 都指向自己；
4. 否则，将 `node` 的 `next` 指针指向 `p` 的下一个结点；
5. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

代码实现如下：

C++"实现"

```
void insertNode(int i, Node *p) {
    Node *node = new Node;
    node->value = i;
    node->next = NULL;
    if (p == NULL) {
        p = node;
        node->next = node;
    }
    else{
        node->next = p->next;
```

```
            p->next = node;
            }
    }
```

双向循环链表

在向双向循环链表插入数据时，除了要判断给定链表是否为空外，还要同时修改左、右两个指针。

大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `left` 和 `right` 指针，以及 `p` 都指向自己；
4. 否则，将 `node` 的 `left` 指针指向 `p`；
5. 将 `node` 的 `right` 指针指向 `p` 的右结点；
6. 将 `p` 右结点的 `left` 指针指向 `node`；
7. 将 `p` 的 `right` 指针指向 `node`。

代码实现如下：

```
void insertNode(int i, Node *p)
{
   Node *node = new Node;
   node->value = i;
   if (p == NULL) {
       p = node;
       node->left = node;
       node->right = node;
   }
   else{
            node->left = p;
            node->right = p->right;
            p->right->left = node;
            p->right = node;
   }
}
```

从链表中删除数据

单向（循环）链表

设待删除结点为 `p`，从链表中删除它时，将 `p` 的下一个结点 `p->next` 的值覆盖给 `p` 即可，与此同时更新 `p` 的下下个结点。

流程大致如下：

1. 将 `p` 下一个结点的值赋给 `p`，以抹掉 `p->value`；
2. 新建一个临时结点 `t` 存放 `p->next` 的地址；
3. 将 `p` 的 `next` 指针指向 `p` 的下下个结点，以抹掉 `p->next`；

4. 删除 `t`。此时虽然原结点 `p` 的地址还在使用，删除的是原结点 `p->next` 的地址，但 `p` 的数据被 `p->next` 覆盖，`p` 名存实亡。

具体过程可参考下图：

代码实现如下：

```
void deleteNode(Node *p) {
    p->value = p->next->value;
    Node *t = p->next;
    p->next = p->next->next;
    delete t;
}
```

双向循环链表

流程大致如下：

1. 将 `p` 左结点的右指针指向 `p` 的右节点；
2. 将 `p` 右结点的左指针指向 `p` 的左节点；
3. 新建一个临时结点 `t` 存放 `p` 的地址；
4. 将 `p` 的右节点地址赋给 `p`，以避免 `p` 变成悬垂指针；
5. 删除 `t`。

代码实现如下：

"C++"

```
void deleteNode(Node *&p) {
    p->left->right = p->right;
    p->right->left = p->left;
    Node *t = p;
    p = p->right;
    delete t;
}
```