面经

算法&数据结构

1.二分查找法

非递归：

```
int bin_search(int* arr, int n, int key)
{
    int start = 0;
    int end = n-1;
    while(start <= end) {
        int mid = (end - start)/2 + start;
        if (arr[mid] < key)
            start = mid + 1;
        else if (arr[mid] > key)
            end = mid - 1;
        else
            return mid;// end = mid;
    }
    return -1;
}
```

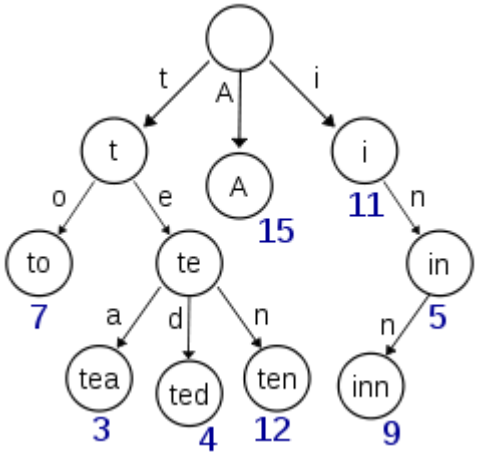
递归：

```
int bin_search(int* arr, int start, int end, int key)
{
    int mid = (end - start)/2 + start;
    if (arr[mid] == key)
        return mid;// return bin_search(arr, start,
mid, key);

    if (start >= end)
        return -1;
    if (arr[mid] > key)
        return bin_search(arr, start, mid - 1, key);
    if (arr[mid] < key)
        return bin_search(arr, mid + 1, end, key);
}
```

2.字典树Trie

Hash的变种，以空间换时间。查询和插入复杂度为O(len(key))。主要用于大数据量的字符串统计和排序，比如电商关键词推荐系统。



结构体：

```
// 最大数组长度，比如英文字母26，中文常用字是3500
#define MAX_LEN 26
typedef struct TrieNode {
    int count;  // 统计匹配次数
    bool leaf;  // 是否构成单词，即叶节点
    struct TrieNode* keys[MAX_LEN]; // 下一层节点
} TrieNode , * Trie;
```

3.Skiplist

通过概论平衡替代严格平衡，性能与红黑树相同，但实现更简单。并发环境下，红黑树可能需要加锁整棵树；而跳表涉及的节点更局部，只需锁定部分节点。Redis里面使用skiplist的理由是：1.内存不敏感。2.range操作更方便。3.实现和调试简单。

```
// 这里仅仅是一个指针
typedef struct nodeStructure
{
    keyType key; // key值
    valueType value; // value值
    // 向前指针数组，根据该节点层数的
    // 不同指向不同大小的数组
    node forward[1];
}*node;

// 定义跳表数据类型
typedef struct listStructure{
  int level;    /* Maximum level of the list (1 more than the number of levels in the list) */
  struct nodeStructure * header; /* pointer to header */
  } * list;
```

4.快速排序

通过左右半边数据进行交换，每次得到两个有序的子序列。
复杂度递归O(T) = 2*O(T/2) + n，调用深度log(n)，所以复杂度为nlog(n)。只能递归。

//思路：从右边找到第一个小于pivotal，交换；从左边找到第一个大于pivotal的值，交换；重复。

```
void quick_sort(int* arr, int n)
{
    int i = 0;
    int j = n-1;
    int p = 0;
    int pv = arr[p];
    if (n <= 0)
        return;

    while(i < j) {
        while(i<j && arr[j]>=pv)
            j--;
        if (arr[j]>=arr[p]) {
            arr[p] = arr[j];
            p = j;
        }
        while (i<j && arr[i]<= pv)
            i++;
        if (arr[i]<=pv) {
            arr[p] = arr[i];
            p = i;
        }
    } // while i <= j
    arr[p] = pv;
    quick_sort(arr, p-1);
    quick_sort(arr+p+1, n-p-1);
} // quick_sort
```

4.1获取前第k个最大/最小元素

通过左右半边数据进行交换，每次得到以某个关键字为分界的两个相对有序的子序列。副作用是子序列内部不保证有序。复杂度与快排相同，最糟nlog(n)，平均nlog(k)，因为数组的关系，适合少量数据。还有最小堆法，额外存储k，复杂度nlog(k)，适合海量数据.

```
int max_kth(int* arr, int n, int k)
{
    int i = 0;
    int j = n-1;
    int p = i;
    int pv = arr[p];
    if (n <= 0)
        return;

    while(i < j) {
        while(i<j && arr[j]>=pv)
            j--;
        if (arr[j]>=arr[p]) {
            arr[p] = arr[j];
            p = j;
        }
        while (i<j && arr[i]<= pv)
            i++;
        if (arr[i]<=pv) {
            arr[p] = arr[i];
            p = i;
        }
    } // while i <= j
    arr[p] = pv;
    if (p == k) return arr[p];
    else if (p > k)
        quick_sort(arr, p-1, k);
    else
        quick_sort(arr+p+1, n-p-1, k-p);
} // quick_sort
```

5.堆排序

堆是父节点大于等于子节点（最大堆）或父节点小于等于子节点（最小堆）的完全二叉树。
建堆：从中间节点[n/2]开始调整内部所有节点，复杂度nlog(n)。
插入/删除：复杂度log(n)
堆排序：n个元素，每次调整最多log(n)次交换，复杂度为nlog(n)。

```
void heap_sort(int* arr, int n) {
    // build heap
    for(int i = n/2; i>0;i--) {
        heap_adjust(arr, i, n);
    }

    for(int i =n; i>0;i--) {
        // swap first&last element
        int tmp = arr[n-1];
        arr[n-1] = arr[0];
        arr[0] = tmp;

        heap_adjust(arr, 0, i);
    }
}
```

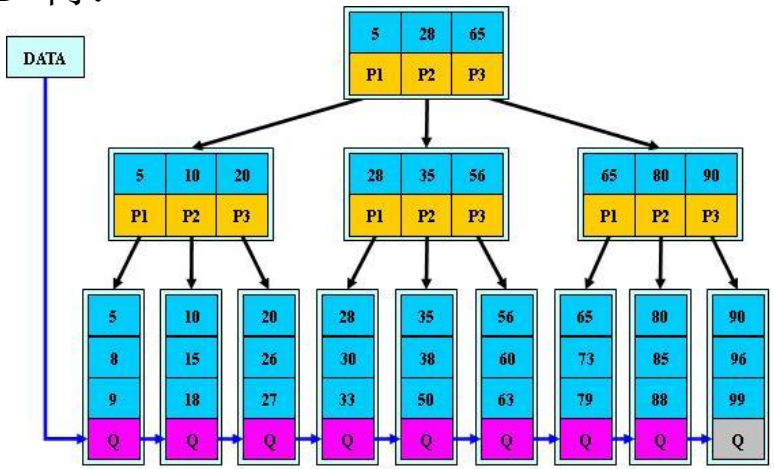
```
void heap_adjust(int* arr, int p, int len) {
    int c = 2*p;// child
    int val = arr[p];
    while(c<len) {
        if (arr[c] < arr[c+1])
            c++;
        if (val <= arr[c])
            break;

        arr[c] = val; // swap
        p = c;
        c = 2*c;
    }
    arr[p] = val;
}
```

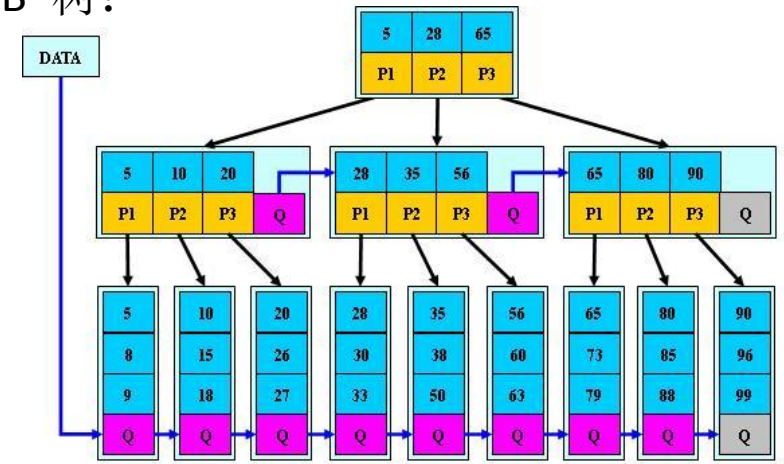
6.B-Tree

B树：二叉树。
B-树：多叉树。每个节点都存储数据。
B+树：叶子节点增加横向指针，只有叶子节点存储数据。
B*树：叶子节点和内层非跟节点增加横向指针，只有叶子节点存储数据。
搜索和删除性能等价于二分查找，即树高log(n)。

B+树：



B*树：



7.数据库索引

聚集索引：内部节点存储键值索引，叶子节点存储数据。索引顺序与数据存储顺序一致。一个表即只能有一个聚集索引。适合多行范围检索。其实就是数据库表本身，前半部存储索引，后面存储行数据。

非聚集索引：内部节点存储键值索引，叶子节点存储数据所在页面地址。索引键值顺序与实际数据存储顺序不相关，可以有多个。并保存为单独的索引文件。某些字段上建非聚集索引可能显著增大数据库文件大小，慎用。适合单行检索。

Mysql:(索引结构为B+树)

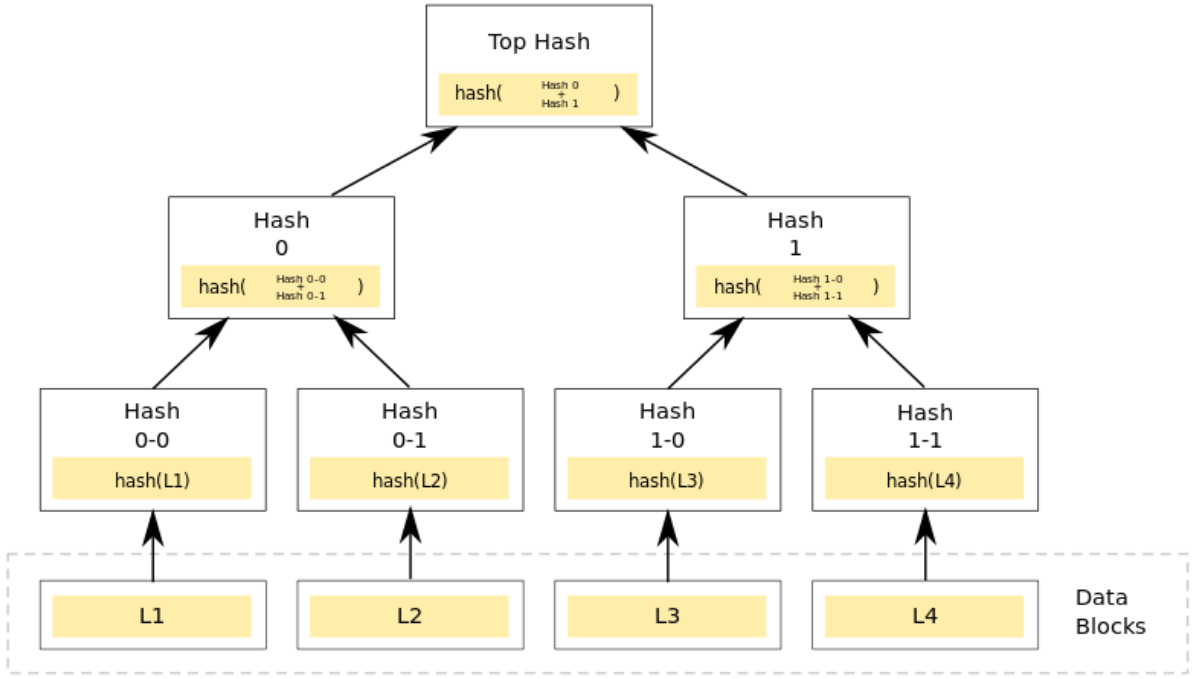
MyISAM：不支持聚簇索引；不支持行级锁，每次更新需要锁整个表。

InnoDB：支持聚簇索引，ACID事务，行级锁和外键约束；不支持全文搜索。

ACID：原子性，一致性，隔离性，持久性。

8.Merkle树（hash树）

默克尔树也叫hash树，叶节点是数据块的hash，内部节点是其孩子节点的加密hash。用于数据防篡改，身份验证。应用于数字签名，可信计算，p2p下载数据校验，区块链身份验证（比特币和以太坊底层技术），IPFS文件系统，数据块Cassadra，git等。



9.进程，线程，协程模型

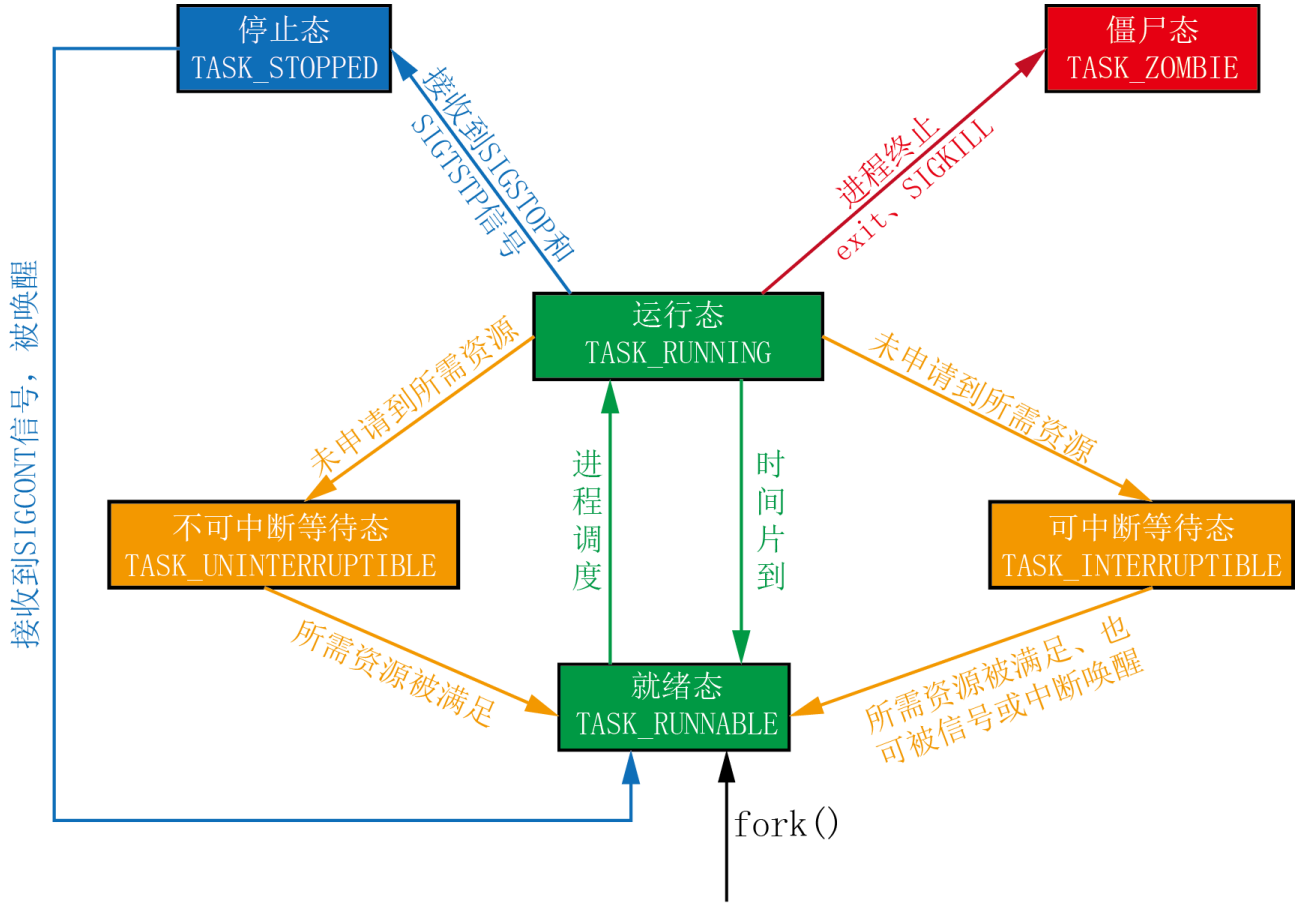
上下文切换：（进程和线程切换只能发生在内核）
进程：
1.切换页目录以使用新的地址空间。（file tables, signal tables, page tables, cpu caches）
2.切换内核栈和硬件上下文。（sp，pc，registers）
线程：
1.切换内核栈和硬件上下文。（sp，pc，registers）
协程：
1.切换用户栈。

进程是系统资源分配基本单位，线程是系统调度基本单位。
fork：调用clone，拷贝数据尽量少。不包括内存，文件描述符，信号描述符等。
pthread_create：调用clone，拷贝数据尽量多。包括内存，文件描述符，信号描述符等。

进程间通信IPC：管道，信号，消息队列，共享内存，信号量，socket。

9.1.进程，线程，协程模型

进程状态：RUNNING（TASK_RUNNING, TASKR_UNABLE）, WAITTING（TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE ）, STOPPED（TASK_STOPPED）, ZOMBIE（TASK_ZOMBIE）。



进程调度和状态变化

10.TCP拥塞控制

Tahoe：
慢启动：窗口从1开始。
拥塞避免：发生丢包，窗口减到1。

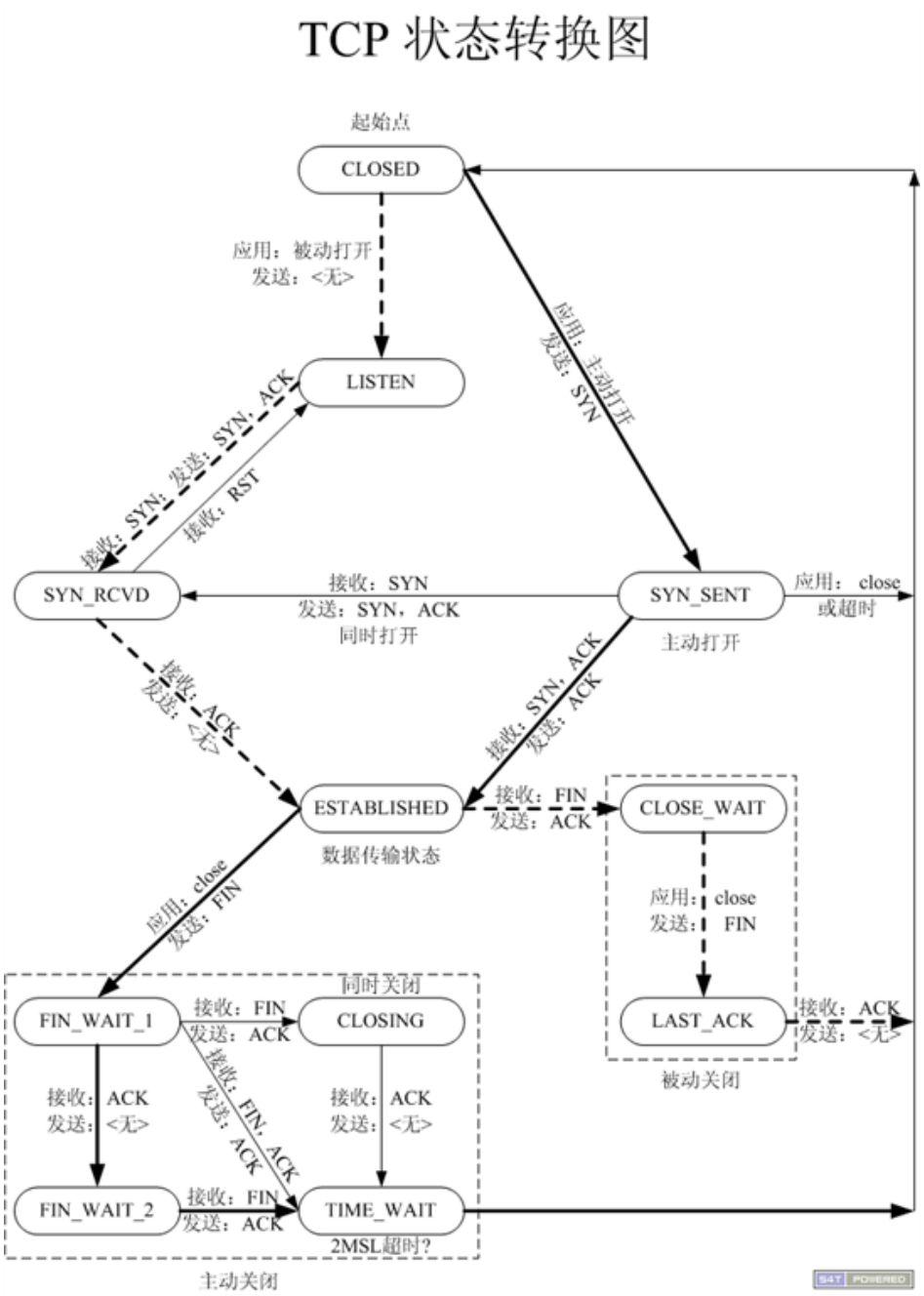
Reno：（认为发送窗口只有一个包丢失）
快重传：收到3次重复ACK，重传包，不等超时。
快恢复：丢包时，窗口减半（因为收到连续3次ack，网络还不算很差）；遇到新包的ack，结束快恢复，cwnd=ssthresh，进入拥塞避免。

New-Reno：（处理发送窗口有多个包丢失的情况）
新快恢复：退出条件修改为“收到当前发送窗口中全部包的ack”，即如果是“部分确认”，不退出快恢复 。
避免一个窗口周期多次触发快恢复，导致窗口多次减小。

10.1 TCP状态图

TIME_WAIT状态作用：

1. 用于发送last_ack，保证对端收到。
2. 使网络中该连接的包失效，防止与新的连接混淆数据。



10.2 epoll LT&ET

LT和ET触发模式使用场景：
LT使用：
1.读缓冲区有数据可读，始终触发IN。
2.写缓冲区有空间可写，始终触发OUT。
使用方式：
IN事件时：读取任意大小的数据。
OUT事件时：写任意大小的数据。如果写出完毕，关闭OUT事件。

ET触发：
1.读缓冲区从没数据到有数据可读，触发IN。
2.写缓冲区从满数据到有空间可写，触发OUT。
使用方式：
IN事件时：一直读取数据，直到EAGAIN。
OUT事件时：一直写数据，直到EAGAIN。
当需要写数据时，直接写数据，直到EAGAIN。不用等OUT事件。

ET在写出时比LT少了一个关闭OUT事件的操作。事件操作更少，某些场景更高效。如写出时，在没数据可用的情况下，ET直接略过而LT需要暂时关闭OUT事件，不然就一直触发。简单说，ET在写出时比LT更高效，因为在没数据可写或写出完毕时，LT都需要关闭OUT（不关闭就一直上报）。

11 链表翻转

思路：

1.从头开始，左边成为一个新list，每次从右边拿出一个节点放入左侧新list。

2.从第2个节点开始，依次遍历并插入到第1个节点后面，最后把1移到最后。

```
// 就地翻转
Node* reverse_inplace(Node* list)
{
    Node* newlist = NULL;
    while(list) {
        Node* next = list->next; //保存
        list->next = newlist; // 翻转

        newlist = list;
        list = next;
    }
    return newlist;
}
```

```
//插入翻转
Node* reverse_insert(Node* list)
{
    Node* first = list;
    Node* second = first->next;
    Node* curr = second->next;
    while(curr) {
        first->next = curr;
        second->next = curr->next;
        curr->next = second;

        curr = second->next;
    }
    // 首节点移到最后
    first->next = NULL;
    second->next = first;
}
```

11.1 链表环检测及相交检测

思路：
环检测：快慢指针
相交检测：
思路1：两链表相交必定有公共结尾。
思路2：将两个链表首尾相连并检查环。

```
// 环检测
bool list_loop_check(Node* list)
{
    Node* slow = list;
    Node* fast = list;
    while(fast->next) {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow) {
            return true;
        }
    }
    return false;
}
```

11.1 链表环检测及相交检测

找出相交点：减去公共结尾，较长的链表往前移动|Len1-Len2|，然后同步后移，第一个相同点即是相交点。

```
// 找出相交点
Node* find_list_exchange(Node* list, Node* list2)
{
   Node *l1 = list, *l2=list2;
   int len1 = 0, len2 = 0;
   while(l1->next) {
      l1=l1->next;
      len1++;
   }
   while(l2->next) {
      l2=l2->next;
      len2++;
   }
   if (l1!=l2) return NULL;

   int diff = abs(len1, len2);
   l1 = list;
   l2 = list2;
   if (len1 > len2) {
      while(diff-- > 0)
         l1=l1->next;
   }
   if (len1 < len2) {
      while(diff-- > 0)
         l2=l2->next;
   }
   while(l1 != l2) {
      l1= l1->next;
      l2= l2->next;
   }
   return l1;

} // find_list_exchange
```