查找

给定数值并在查找表中确定关键字等于给定值的记录或数据元素。

基本概念：

- 查找表：相同类型的数据元素的集合，每个元素由若干数据项构成；
- 关键字（Key，码）：数据元素中数个数据项的数值，可以表示一个数据元素。
- 静态查找staticSearch：只对数据元素进行查询火箭所，静态查找表；
- 动态查找dynamicSearch：查找的同时插入表中不存在的记录，或删除表中已存在的记录，动态查找表。
- 平均比较次数ASL：衡量查找算法效率的高低`ASL = n∑i=1(Pi×Ci)`，n为查找表中的记录个数：
    1. ASL成功 = 比较次数/元素个数；
    2. ASL失败 = 比较次数/不成功的位置个数。

四种查找方式：

1. 顺序表查找：给定值与表中记录相比较；
2. 链表查找：给定值与表中记录逐个比较；
3. 散列表查找：给定值直接访问表记录；
4. 索引表查找：根据索引确定带查找记录所在的块，从块中查找。

顺序查找

从表的一端关键词逐个将记录中的关键字与给定值比较。若扫描完整个表均无相应记录，则查找失败。

算法分析：

查找成功比较次数`n`（元素在第几位，n就等于几），查找失败`n+1`次；

ASL：查找成功`(n+1)/2`，查找失败`3(n+1)/2`。

折半查找（二分查找）

查找表需要是有序的（升序、降序均可），先确定带查找记录在表中的范围（与表中的中值相比较，大于或小于中值均表示查找数值的范围在中值的其中一侧），然后逐步缩小（每次缩小一半），直到记录的存在与否被确定。

前提条件：必须是有序的，且用顺序结构存储。

折半算法思想，用low、high、mid分别表示上界、下界、中间位置指针，初值low=1、high=n：

1. 取中间位置mid：`mid = ⌊(low+high)/2⌋`；
2. 比较中间位置的值和查找值：
    1. 相等：查找成功；
    2. 大于：查找值在区间的前半段，修改上界指针：`high = mid-1`，回到1.取mid；
    3. 小于：查找值在区间的后半段，修改下届指针：`low = mid+1`，回到1.取mid；
3. 直到`low > high`越界，查找失败。

```
int binSearch(int []st, int n, int key){
    int low = 0, high = n-1, mid;
    while(low < high){
        mid = (low+high)/2;
        if(st[mid] == key) return mid;
        else if(st[mid] < key) low = mid+1;
        else high = mid-1;
    }
    return -1;  //查找失败
}
```

折半查找的查找表可以构建一颗折半树，根为mid、左子树low、右子树high；
折半树只有最下层是不满的，元素个数为n时树高`h = ⌈log2(n+1)⌉`，时间复杂度`O(h) = O(⌈log2(n+1)⌉)`；
ASL成功 = 比较次数/元素个数，
ASL失败 = 比较失败次数/不成功的位置个数；

若无特殊说明，默认向下⌊⌋取整。

动态查找与BST、AVL树

根据查找结果进行 增、删、改 操作。

二叉排序树BST

左子树的结点小于根节点且不为空，右子树的结点大于根节点且不为空，左右子树均为二叉排序树。

查找效率较高，但容易受树的形态所影响。

BST查找

待查找数值K与根节点比较：相等，查找成功；小于，继续沿左子树查找；大于，继续沿右子树查找。

```
/*一种基于递归的BST查找算法*/
BSTNode *BSTSearch(BSTNode *T, keyType){
  if(t == NULL) return NULL;
  else{
    if(T->key == key) return T;
    else if(key < T->key) return BSTSearch(T->LChild, key);
    else return BSTSearch(T->RChild, key);
  }
}
```

BST插入

插入结点s时若BST为空，则s作为根节点。
否则，与根节点比较：相等，不插入；小于，进入左子树，继续比较；大于，进入右子树，继续比较。

```c
/*一种基于递归的插入算法*/
void BSTInsert(BSTNode *T, int key){
  BSTNode *x;
  x = (BSTNode *)malloc(sizeof(BSTNode));
  x->key = key;
  x->LChild = x->RChild = NULL;
  if(T == NULL) T = x;
  else{
    if(T->key == x->key) return NULL; //已有结点，无需插入
    else if(x->key < T->key) BSTInsert(T->LChild, key);
    else BSTInsert(T->RChild, key);
  }
}
```

BST删除

删除结点p，其父节点为f。
如果p是叶子节点，直接删除；如果p只有左子树或右子树，则直接用子树取代p，称为f的子树；
如果p同时又左右子树，有两种方式：

1. 直接用p的中序前驱节点取代p，从p左子树选择最大结点s放在p的位置，然后删除s。
2. 用p的直接中序后继结点代替p，从p右子树选择最小节点s放在p的位置，然后删除s。

BST构造

```c
BSTNode *createBST(){
  keyType key;
  BSTNode *T = NULL;
  scanf("%d", &key);
  while(key != 65535){
    insertBST(T, key);
    scanf("%d", &key);
  }
  return T;
}
```

平衡二叉树AVL

形态总体均匀，查找效率最好。左右子树的深度差不超过1，且都是平衡二叉树。

平衡因子：节点左子树的深度减去右子树的深度。所以平衡二叉树的平衡因子只会是-1、0、1，否则不是平衡二叉树。

如果一个树同时满足二叉排序树和平衡二叉树的特点，则称为平衡二叉排序树BBST。
BBST的平均查找长度为`O(log2n)`，平均时间复杂度为`O(log2n)`。

一般二叉排序树不是平衡的，可以通过构造平衡二叉树进行平衡化旋转。

对AVL树进行删除或插入操作，通常会影响从根节点到插入/删除结点路径上的某些节点，使得这些结点的子树可能会变化（**LL，LR，RR，RL**）。

平衡化旋转

层次路径是从根到叶子的一条路径。

新插入的结点会影响叶子的层次路径，沿着插入节点上行到根节点——这样就可以找到失衡节点。从失衡结点往下推三个点，这三个点就是要调整的点。

红黑树RedBlackTree【2022新增】

红黑树在AVL的基础上放宽条件：左右子树高度差不超过两倍，一种近似平衡的结构。