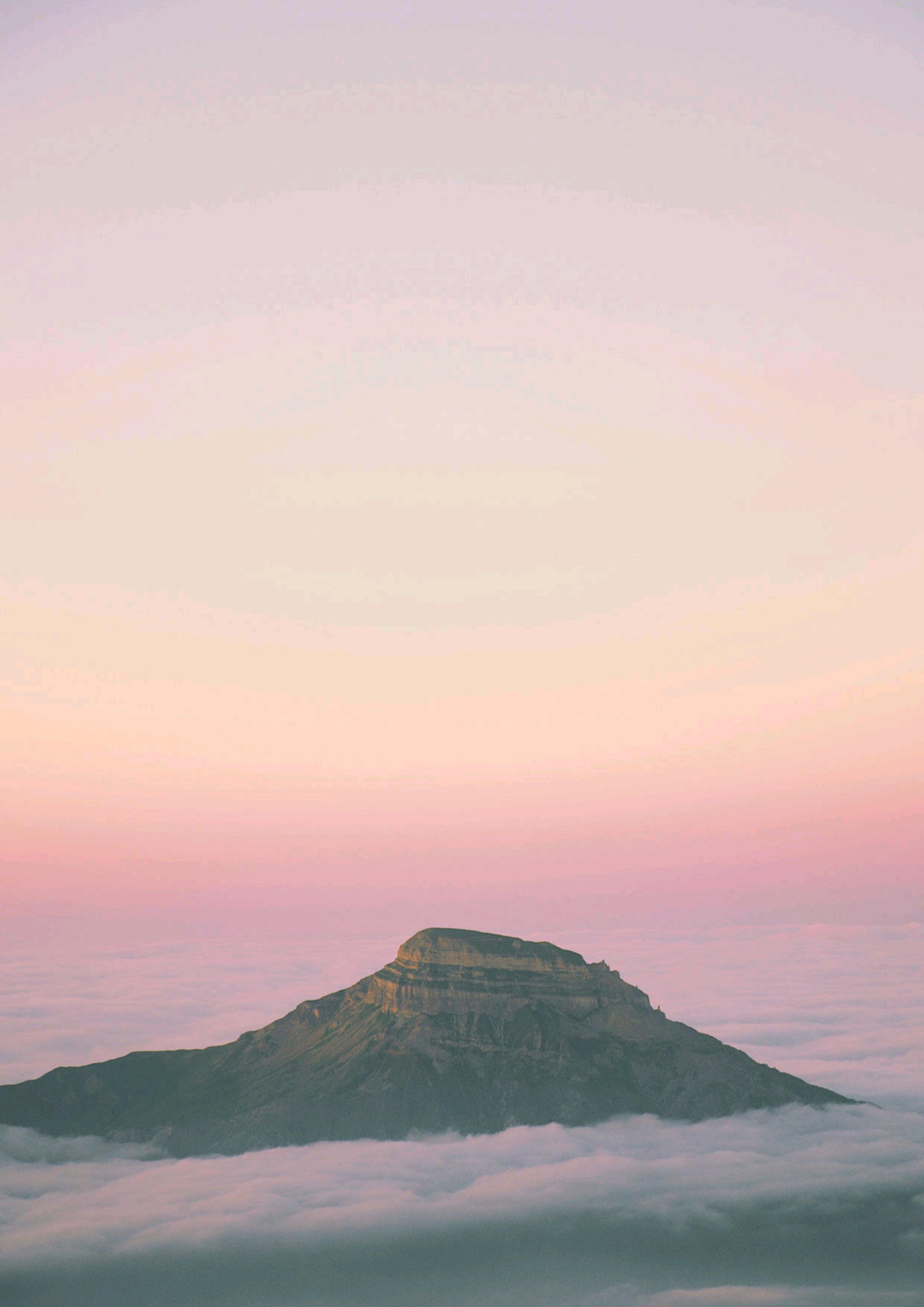


查找

静态查找表：只作查找操作的查找表。

动态查找表：在查找过程中同时插入查找表中不存在的元素，或删除元素。

一、顺序查找表

又称线性查找

```c
/* 顺序查找，a为数组，n为要查找的数组个数，key为要查找的关键字 */
int Sequential_Search(int *a,int n,int key)
{
    int i;
    for(i=1;i<=n;i++)
    {
        if (a[i]==key)
            return i;
    }
    return 0;
}
```

优化：设置哨兵（在a[0]） 省去了每次循环判断是否越界的过程

```c
/* 有哨兵顺序查找 */
int Sequential_Search2(int *a,int n,int key)
{
    int i;
    a[0]=key;                /* 设置a[0]为关键字值，我们称之为“哨兵” */
    i=n;                     /* 循环从数组尾部开始 */
    while(a[i]!=key)
    {
        i--;
    }
    return i;                /* 返回0则说明查找失败 */
}
```

平均查找次数 $\frac{n+1}{2}$　　时间复杂度 $O(n)$

二、有序查找表（线性表已排好序）

1. 折半查找（二分查找）

设置 上下界：high，low；

```
1  int Binary_Search(int *a,int n,int key)
2  {
3      int low,high,mid;
4      low=1;                      /* 定义最低下标为记录首位 */
5      high=n;                     /* 定义最高下标为记录末位 */
6      while(low<=high)
7      {
8          mid=(low+high)/2;       /* 折半 */
9          if (key<a[mid])         /* 若查找值比中值小 */
10             high=mid-1;         /* 最高下标调整到中位下标小一位 */
11         else if (key>a[mid])    /* 若查找值比中值大 */
12             low=mid+1;          /* 最低下标调整到中位下标大一位 */
13         else
14             return mid;         /* 若相等则说明mid即为查找到的位置 */
15     }
16     return 0;
17 }
```

最坏查找情况：$\lfloor \log_2 n \rfloor + 1$

时间复杂度 $O(\log n)$

2. 插值查找

将 $mid = \frac{low+high}{2} = low + \frac{1}{2}(high-low)$

改为 $mid = low + \frac{key-a[low]}{a[high]-a[low]}(high-low)$

适用于分布均匀的数据。

3、斐波那契查找（黄金分割原理）

```
1  int Fibonacci_Search(int *a,int n,int key) /* 斐波那契查找 */
2  {
3      int low,high,mid,i,k;
4      low=1;                          /* 定义最低下标为记录首位 */
5      high=n;                         /* 定义最高下标为记录末位 */
6      k=0;
7      while(n>F[k]-1)                 /* 计算n位斐波那契数列的位置 */
8          k++;
9      for (i=n;i<F[k]-1;i++)          /* 将不满的数值补全 */
10         a[i]=a[n];
11     while(low<=high)
12     {
13         mid=low+F[k-1]-1;           /* 计算当前分隔的下标 */
14         if (key<a[mid])             /* 若查找记录小于当前分隔记录 */
15         {
16             high=mid-1;             /* 最高下标调整到分隔下标mid-1处 */
17             k=k-1;                  /* 斐波那契数列下标减一位 */
18         }
19         else if (key>a[mid])        /* 若查找记录大于当前分隔记录 */
20         {
21             low=mid+1;              /* 最低下标调整到分隔下标mid+1处 */
22             k=k-2;                  /* 斐波那契数列下标减两位 */
23         }
24         else
25         {
26             if (mid<=n)
27                 return mid;         /* 若相等则说明mid即为查找到的位置 */
28             else
29                 return n;           /* 若mid>n说明是补全数值，返回n */
30         }
31     }
32     return 0;
33 }
```

最简单的加减法

三、线性索引查找

索引就是把一个关键建字与它对应的记录相关联的过程。

线性索引就是将索引项集合组成为线性结构，也称为索引表。

1、稠密索引

指在索引中，将数据集中的每个记录对应一个索引项

对于稠密索引这个索引表来说，索引项一定是按照关键码有序排列的

数据集大，查找性索下降。

2. 分块索引

分块有序，是把数据集的记录分成了若干块。

且满足 { 块内无序：不要求，但可以
块间有序：后一块全部大于前一块的

索引项结构 { 最大关键码：每一块中的最大关键字
块长：存储块中的记录个数
块首指针：指向块首数据元素的指针

n个记录，m块，t条记录

平均查找长度：$ASL = \frac{m+1}{2} + \frac{t+1}{2} = \frac{1}{2}(m+t)+1 = \frac{1}{2}(\frac{n}{t}+t)+1$

比顺序快，比折半慢。

3. 倒排索引

索引项：次关键码，记录号表

↓

储存相同关键字的所有记录的记录号

缺点：索引项建立时长未知。

四、二叉排序树

又称二叉排序树

1.查找操作

结点结构

```c
/* 二叉树的二叉链表的结点结构定义 */
typedef  struct BiTNode                    /* 结点结构 */
{
    int data;                              /* 结点数据 */
    struct BiTNode *lchild, *rchild;       /* 左右孩子指针 */
} BiTNode, *BiTree;
```

查找

```c
Status SearchBST(BiTree T, int key, BiTree f, BiTree *p)
{ /* 递归查找二叉排序树T中是否存在key, */
    if (!T)     /* 若查找不成功，指针p指向查找路径上访问的最后一个结点并返回FALSE */
    {
        *p = f;
        return FALSE;
    }
    else if (key==T->data) /* 若查找成功，则指针p指向该数据元素结点，并返回TRUE */
    {
        *p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);      /* 在左子树中继续查找 */
    else
        return SearchBST(T->rchild, key, T, p);      /* 在右子树中继续查找 */
}
```

2.插入操作

```c
Status InsertBST(BiTree *T, int key)
{
    BiTree p,s;
    if (!SearchBST(*T, key, NULL, &p))  /* 查找不成功 */
    {
        s = (BiTree)malloc(sizeof(BiTNode));
        s->data = key;
        s->lchild = s->rchild = NULL;
        if (!p)
            *T = s;                          /* 插入s为新的根结点 */
        else if (key<p->data)
            p->lchild = s;                   /* 插入s为左孩子 */
        else
            p->rchild = s;                   /* 插入s为右孩子 */
        return TRUE;
    }
    else
        return FALSE;                        /* 树中已有与关键字相同的结点，不再插入 */
}
```

3. 删除

```
Status DeleteBST(BiTree *T,int key)
{/* 二叉排序树T中存在关键字等于key的数据元素时，则删除该数据结点 */
   if(!*T)                              /* 不存在关键字等于key的数据元素 */
      return FALSE;
   else
   {
      if (key==(*T)->data)      /* 找到关键字等于key的数据元素 */
         return Delete(T);
      else if (key<(*T)->data)
         return DeleteBST(&(*T)->lchild,key);
      else
         return DeleteBST(&(*T)->rchild,key);

   }
}
```

Delete 函数.

```
1  Status Delete(BiTree *p)
2  {/* 从二叉排序树中删除结点p，并重接它的左或右子树。 */
3     BiTree q,s;
4     if((*p)->rchild==NULL) /* 右子树空则只需重接它的左子树（待删结点是叶子也走此分支） */
5     {
6        q=*p; *p=(*p)->lchild; free(q);
7     }
8     else if((*p)->lchild==NULL) /* 只需重接它的右子树 */
9     {
10       q=*p; *p=(*p)->rchild; free(q);
11    }
12    else                        /* 左右子树均不空 */
13    {
14       q=*p; s=(*p)->lchild;
15       while(s->rchild)         /* 转左，然后向右到尽头（找待删结点的前驱） */
16       {
17          q=s; s=s->rchild;
18       }
19       (*p)->data=s->data; /* s指向被删结点直接前驱(用被删结点前驱的值取代被删结点的值)
20       if(q!=*p)
21          q->rchild=s->lchild;/* 重接q的右子树 */
22       else
23          q->lchild=s->lchild;/* 重接q的左子树 */
24       free(s);
25    }
26    return TRUE;
27 }
```

五、平衡二叉树（AVL树）

是一种二叉排序树，其中每个结点的左子树和右子树的高度至多等于1

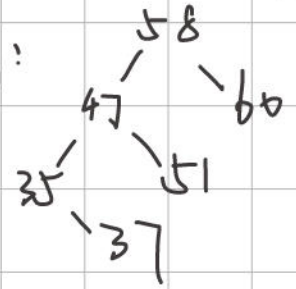
也叫 AVL树

平衡因子：二叉树结点上左子树的高度减去右子树的高度称为平衡因子（BF）

{0, -1, 1}

距离插入结点最近的，且平衡因子的绝对值大于1的结点，为根的子树，我们称为最小不平衡子树

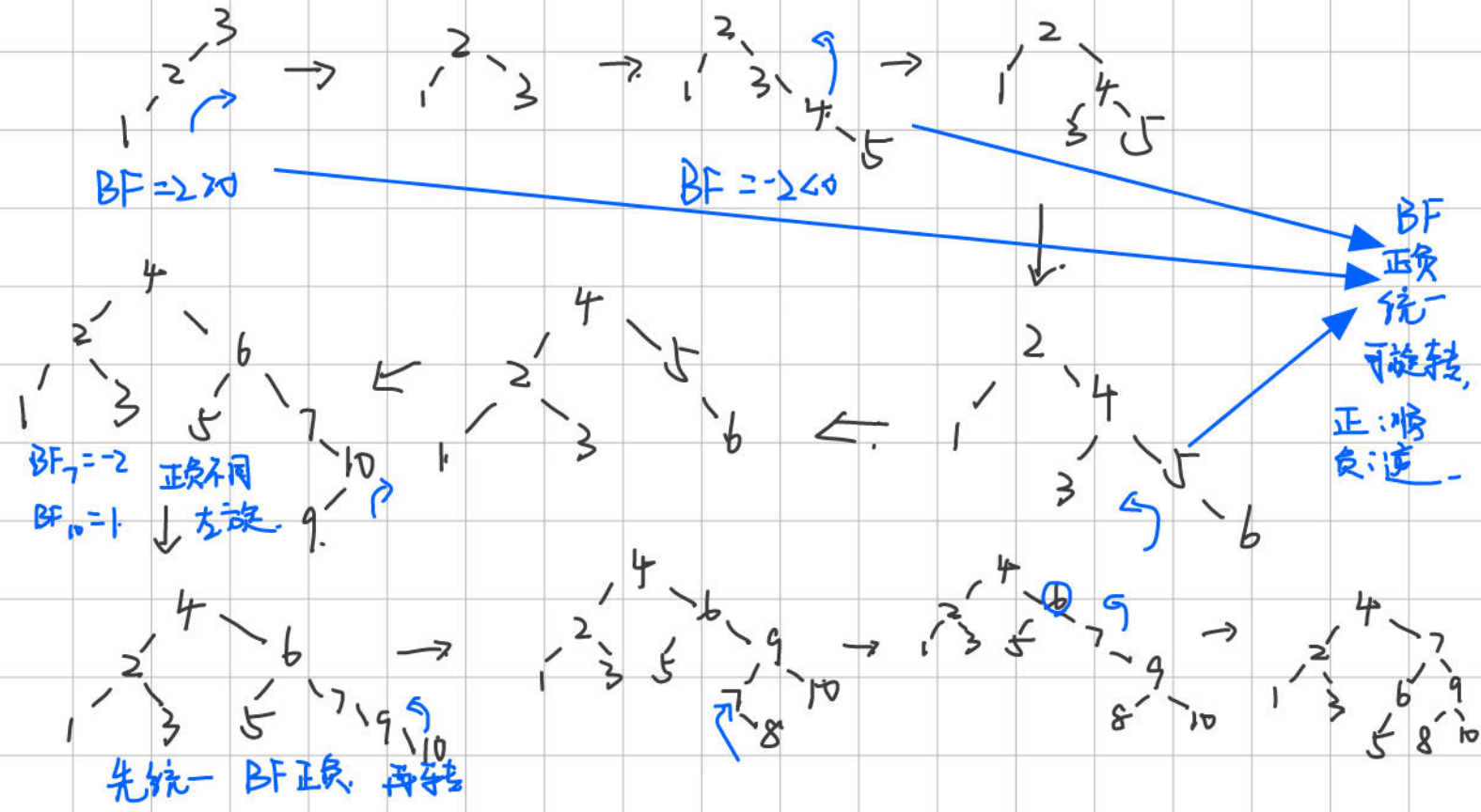
例：



1. 实现原理

插入后调整最小不平衡子树各结点之间的连接关系。

例 a[n] = {3, 2, 1, 4, 5, 6, 7, 10, 9, 8}



2、实现算法

结点结构

```c
/* 二叉树的二叉链表结点结构定义 */
typedef  struct BiTNode                    /* 结点结构 */
{
    int data;                              /* 结点数据 */
    int bf;                                /* 结点的平衡因子 */
    struct BiTNode *lchild, *rchild;       /* 左右孩子指针 */
} BiTNode, *BiTree;
```

右旋操作代码

```c
/* 对以P为根的二叉排序树作右旋处理，  */
/* 处理之后p指向新的树根结点，即旋转处理之前的左子树的根结点 */
void R_Rotate(BiTree *P)
{
    BiTree L;
    L=(*P)->lchild;              /* L指向P的左子树根结点 */
    (*P)->lchild=L->rchild; /* L的右子树挂接为P的左子树 */
    L->rchild=(*P);
    *P=L;                        /* P指向新的根结点 */
}
```

左旋操作代码

```c
/* 对以P为根的二叉排序树作左旋处理 */
/* 处理之后P指向新的树根结点，即旋转处理之前的右子树的根结点0 */
void L_Rotate(BiTree *P)
{
    BiTree R;
    R=(*P)->rchild;              /* R指向P的右子树根结点 */
    (*P)->rchild=R->lchild; /* R的左子树挂接为P的右子树 */
    R->lchild=(*P);
    *P=R;                        /* P指向新的根结点 */
}
```

左旋处理代码

```c
#define LH +1 /* 左高 */
#define EH 0  /* 等高 */
#define RH -1 /* 右高 */

/* 对以指针T所指结点为根的二叉树作左平衡旋转处理 */
/* 本算法结束时，指针T指向新的根结点 */
1  void LeftBalance(BiTree *T)
2  {
3      BiTree L,Lr;
4      L=(*T)->lchild;                    /* L指向T的左子树根结点 */
5      switch(L->bf)          /* 检查T的左子树的平衡度，并作相应的平衡处理 */
6      {
7          case LH:           /* 新结点插入在T的左孩子的左子树上，要作单右旋处理 */
8              (*T)->bf=L->bf=EH;
9              R_Rotate(T);
10             break;
11         case RH:           /* 新结点插入在T的左孩子的右子树上，要作双旋处理 */
12             Lr=L->rchild;              /* Lr指向T的左孩子的右子树根 */
13             switch(Lr->bf)             /* 修改T及其左孩子的平衡因子 */
14             {
15                 case LH: (*T)->bf=RH;
16                          L->bf=EH;
17                          break;
18                 case EH: (*T)->bf=L->bf=EH;
19                          break;
20                 case RH: (*T)->bf=EH;
21                          L->bf=LH;
22                          break;
23             }
24             Lr->bf=EH;
25             L_Rotate(&(*T)->lchild);    /* 对T的左子树作左旋平衡处理 */
26             R_Rotate(T);                /* 对T作右旋平衡处理 */
27     }
28 }
```

右旋处理代码

```c
void RightBalance(BiTree *T)
{
    BiTree R,Rl;
    R=(*T)->rchild;
    switch(R->bf)                          /* R指向T的右子树根结点 */
    { /* 检查T的右子树的平衡度，并作相应平衡处理 */
     case RH: /* 新结点插入在T的右孩子的右子树上，要作单左旋处理 */
              (*T)->bf=R->bf=EH;
              L_Rotate(T);
              break;
     case LH: /* 新结点插入在T的右孩子的左子树上，要作双旋处理 */
              Rl=R->lchild;                /* Rl指向T的右孩子的左子树根 */
              switch(Rl->bf)               /* 修改T及其右孩子的平衡因子 */
              {
                case RH: (*T)->bf=LH;
                         R->bf=EH;
                         break;
                case EH: (*T)->bf=R->bf=EH;
                         break;
                case LH: (*T)->bf=EH;
                         R->bf=RH;
                         break;
              }
              Rl->bf=EH;
              R_Rotate(&(*T)->rchild);  /* 对T的右子树作右旋平衡处理 */
              L_Rotate(T);              /* 对T作左旋平衡处理 */
}
```

主函数

```
Status InsertAVL(BiTree *T,int e,Status *taller)
{                                   /* 插入新结点，树"长高"，置taller为TRUE */
    if(!*T)
    {
        *T=(BiTree)malloc(sizeof(BiTNode));
        (*T)->data=e;
        (*T)->lchild=(*T)->rchild=NULL;
        (*T)->bf=EH;
        *taller=TRUE;
    }
    else
    {
        if (e==(*T)->data)          /* 树中已存在和e有相同关键字的结点则不再插入 */
        {
            *taller=FALSE;
            return FALSE;
        }
        if (e<(*T)->data)           /* 应继续在T的左子树中进行搜索 */
        {
            if(!InsertAVL(&(*T)->lchild,e,taller)) /* 未插入 */
                return FALSE;
            if(*taller)             /* 已插入到T的左子树中且左子树"长高" */
            {
                switch((*T)->bf)/* 检查T的平衡度 */
                {
                    case LH:    /* 原本左子树比右子树高，需要作左平衡处理 */
                        LeftBalance(T);
                        *taller=FALSE;
                        break;
                    case EH:    /* 原本左、右子树等高，现因左子树增高而使树增高 */
                        (*T)->bf=LH;
                        *taller=TRUE;
                        break;
                    case RH:    /* 原本右子树比左子树高，现左、右子树等高 */
                        (*T)->bf=EH;
                        *taller=FALSE;
                        break;
                }
            }
        }
        else                        /* 应继续在T的右子树中进行搜索 */
        {
            if(!InsertAVL(&(*T)->rchild,e,taller)) /* 未插入 */
                return FALSE;
            if(*taller)             /* 已插入到T的右子树且右子树"长高" */
            {
                switch((*T)->bf)/* 检查T的平衡度 */
                {
                    case LH:    /* 原本左子树比右子树高，现左、右子树等高 */
                        (*T)->bf=EH;
                        *taller=FALSE;
                        break;
                    case EH:    /* 原本左、右子树等高，现因右子树增高而使树增高 */
                        (*T)->bf=RH;
                        *taller=TRUE;
                        break;
                    case RH:    /* 原本右子树比左子树高，需要作右平衡处理 */
                        RightBalance(T);
                        *taller=FALSE;
                        break;
                }
            }
        }
    }
    return TRUE;
}
```

实际操作

```
int i;
int a[10]={3,2,1,4,5,6,7,10,9,8};
BiTree T=NULL;
Status taller;
for(i=0;i<10;i++)
{
    InsertAVL(&T,a[i],&taller);
}
```

六. 多路查找树

每个结点的孩子数可以多于两个，且每个结点可以储存多个元素。

1. B 树

一种平衡的多路查找树，

结点最大的孩子数目称为B树的阶。

最坏查找数：$\log_{\lceil \frac{m}{2} \rceil}\left(\frac{n+1}{2}\right)+1$　　(n:关键字数，m:阶)

2. B^+ 树

结点有n棵子树 就有n个关键字

七. 散列表查找 (哈希表)

1. 定义

散列技术是记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使得每一个关键字对应一个存储位置 f(key)

f 称为散列函数，又称哈希函数。

采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为 散列表或哈希表

2. 查找步骤

∴ 散列技术既是一种存储方法，又是一种查找方法。

是面向查找的存储结构。

最适合查找与给定值相等的记录的求解问题。

当 $f(key_1)=f(key_2)$ 称为冲突。

八、散列函数的构造方法

一个好的散列函数 { 计算简单
散列地址分布均匀

1. 直接定址法

取关键字的某个线性函数值为散列地址。

$f(key) = a \times key + b$

不常用

2. 数字分析法

抽取关键字的一部分来计算散列存储位置。 适用于关键字分布均匀

3. 平方取中法

适用于：不知道关键字分布，且位数少

例：1234 → $1234^2 = 1522756$ → 227

4. 折叠法

例：98765432|10

→ 987 + 654 + 321 = 1962

→ 962

不需知道关键字分布，
适用于关键字位数较多的情况

5. 除留余数法 (最常用)

$f(key) = key \bmod p \quad (p \le m)$ (m：表长)

通常 p 为小于或等于 m 的最小质数，或不包含小于20质因子的合数。

6. 随机数法 适用于关键字长度不等

$f(key) = random(key)$

九 处理散列冲突的方法

1. 开放地址法

一旦发生冲突，寻找下一个空的散列地址，并将记录记入

称为线性 探测法

缺点：不是同义词(散列地址相同)却需争夺一个地址

二次探测法：$f_i(key) = (f(key) + d_i) \ MOD \ m$ $(d_i = 1^2, -1^2, 2^2, -2^2 \cdots q^2, -q^2 \ q \le m/2)$

随机探测法：$f_i(key) = (f(key) + d_i) \ MOD \ m$

d_i是一个随机数列

2. 再散列函数法

$f_i(key) = RH_i(key) \quad (i = 1, 2, \cdots k)$

发生冲突 换函数

3. 链地址法

指针指向 $f(key)$

4. 公共溢出区法

冲突的存在溢出表，对溢出表顺序查找

十. 散列查找的实现

散列表结构

```
#define SUCCESS 1
#define UNSUCCESS 0
#define HASHSIZE 12      /* 定义散列表长为数组的长度 */
#define NULLKEY -32768

typedef struct
{
   int *elem;            /* 数据元素存储基址，动态分配数组 */
   int count;            /* 当前数据元素个数 */
}HashTable;

                         /* 散列表表长，全局变量 */
int m=0;
```

初始化

```c
/* 初始化散列表 */
Status InitHashTable(HashTable *H)
{
    int i;
    m=HASHSIZE;
    H->count=m;
    H->elem=(int *)malloc(m*sizeof(int));
    for(i=0;i<m;i++)
        H->elem[i]=NULLKEY;
    return OK;
}
```

散列函数.

```c
/* 散列函数 */
int Hash(int key)
{
    return key % m; /* 除留余数法 */
}
```

插入

```c
/* 插入关键字进散列表 */
void InsertHash(HashTable *H,int key)
{
    int addr = Hash(key);                /* 求散列地址 */
    while (H->elem[addr] != NULLKEY)     /* 如果不为空，则冲突 */
    {
        addr = (addr+1) % m;             /* 开放定址法的线性探测 */
    }
    H->elem[addr] = key;                 /* 直到有空位后插入关键字 */
}
```

查找

```c
/* 散列表查找关键字 */
Status SearchHash(HashTable H,int key,int *addr)
{
    *addr = Hash(key);                                          /* 求散列地址 */
    while(H.elem[*addr] != key)                                 /* 如果不为空，则冲突 */
    {
        *addr = (*addr+1) % m;                                  /* 开放定址法的线性探测 */
        if (H.elem[*addr] == NULLKEY || *addr == Hash(key))     /* 如果循环回到原点 */
            return UNSUCCESS;                                   /* 则说明关键字不存在 */
    }
    return SUCCESS;
}
```

性能分析：

1. 散列函数是否均匀

2. 处理冲突的方法

3. 散列表的装填因子。