查找

9.1 查找的基本概念、顺序查找法、折半查找法

9.1.1 查找的基本概念

平均查找次数 ASL=Σ p_i*c_i

9.1.2 顺序查找法

基本思路：从表的一端开始，顺序扫描线性表，依次将扫描到的关键字和给定值k对比，若当前扫描的关键字与k相等，则查找成功；若扫描结束后，仍未发现关键字等于k的记录，则查找失败。

算法：

```
int Search(int a[], int n, int k) {
    int i;
    for(i = 1; i<=n; ++i) {
        if(a[i] == k) {
            return i;
        }
    }
    return 0;
}
```

由以上代码克制，根据k取值不同，体现了两种查找长度，一种是成功的，另一种是失败的。ASL也分两种。

第一种：p_i=1/n, c_i=i，因为若k等于a[i]，则在扫描到a[i]之前已经进行了i-1次比较，加上最后一次，共i次则：

ASL_1=Σ i/n=(1/n)*n*(1+n)/2=(n+1)/2

第二种：k在a[]中值之外的范围内取值，则查找不成功。这时候k的取值是无限的，但是对于k的任意一个取值，其查找长度必为n。对于所有i，a[i]==k都不成立，循环n次，则：

ASL_2=n

该算法时间复杂度为O(n)

9.1.3 折半查找法

折半查找要求线性表是**有序的**。

具体步骤见书P278

算法如下：

```
int Bsearch(int R[], int low, int high, int k){
    int mid;
    while(low <= high) {
        mid = (low + high) / 2;
        if(R[mid] == k) {
            return mid;
        } else if(R[mid] > k) {
            return mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return 0;
}
```

时间复杂度为$\log_2 n$

9.1.4 分块查找

1.数据结构

索引表定义如下：

```
typedef struct {
    int key;
    int low, high;
} indexElem;
indexElm index[maxSize];
```

2.算法描述

分块查找可以氛围两步进行：

1. 确定带查找的元素属于哪一块
2. 在块内精确查找该元素

由于索引表是递增有序的，因此第一步采用二分查找，块内元素一般个数较少，因此第二部采用顺序查找即可。

分块查找实际上进行两次查找，整个算法的平均查找长度是两次查找的平均查找长度之和即**二分查找平均查找长度+顺序查找平均查找长度。**

9.2 二叉排序树与平衡二叉树

9.2.1 二叉排序树

1.二叉排序树的定义与存储结构

（1）二叉排序树（BST）的定义

二叉排序树或者是空树，或者是满足一下性质的二叉树：

- 若它的左子树不空，则左子树上所有关键字的值均小于根关键字的值。
- 若它的右子树不空，则右子树上所有关键字的值均大于跟关键字的值。
- 左右子树又各是一棵二叉排序树。

（2）二叉排序树的存储结构

```
typedef struct BTNode {
    int key;
    struct BTNode *lchild;
    struct BTNode *rchild;
} BTNode;
```

2.二叉树的基本算法

（1）查找关键字的算法

要找的关键字要么在左子树，要么在右子树，要么在根结点。根据二叉排序树的定义可知，根结点中的关键字将所有关键字分成了两份，即大于根结点中关键字的部分和小于根结点中的关键字的部分，可以将待查关键字先和根结点中的关键字比较，如果相等则查找成功；如果小宇则到左子树中查找，无需考虑右子树中的关键字；如果大于则在右子树中查找。如果来到当前树的子树根，重复以上过程；若来到了结点的空指针域，则查找失败。

代码：

```
BTNode* BSTSearch(BTNode* bt, int key) {
    if(bt == NULL)
        return NULL;
    else {
        if(bt->key == key)
            return bt;
        else if(key < bt->key)
            return BSTSearch(bt->lchild, key);
        else
            return BSTSearch(bt->rchild, key);
    }
}
```

（2）插入关键字的算法

插入一个关键字首先要找到其插入的位置。对于一个不存在于二叉排序树中的关键字，查找不成功的位置极为该关键字的插入位置。

来到空指针的时候将关键字插入。在插入过程中如果待入关键字已存在，返回0，插入不成功；如果待插入关键字不存在，则插入，返回1。

算法：

```
int BSTInsert(BTNode *&bt, int key) {
    if(bt == NULL){
        bt = (BTNode*)malloc(sizeof(BTNode));
        bt->lchild = bt->rchild = NULL;
        bt->key = key;
        return 1;
    } else {
        if(key == bt->key)
            return 0;
        else if(key < bt->key)
            return BSTInsert(bt->lchild, key);
        else
            return BSTInsert(bt->rchild, key);
    }
}
```

在二叉排序树中插入的关键字均存在新创建的叶子上，由于找到的插入位置总是在空指针域上，因此在空指针域上谅解的新结点必为叶子结点。

（3）二叉排序树的构造方法

```
void CreateBST(BTNode *&bt, int key[], int n) {
    int i;
    bt = NULL;                  //清空树
    for(i = 0; i < n; ++i) {    //调用插入函数，逐个插入关键字
        BSTInsert(bt, key[i]);
    }
}
```

（4）删除关键字的操作

假设在二叉排序树上被删除结点为p，f为其双亲结点，则删除结点p的过程分为以下3种情况：

- **p为叶子结点**。直接删除即可。

- **p只有右子树无左子树，或只有左子树无右子树**。此时只需删除p，然后将p的子树直接连接在原来p与其双亲结点f相连的指针上即可。如图：