1 链表

1.1 单链表

```c
typedef struct LNode{
        int data;
        struct LNode *next;
}LNode, *LinkList;

//查找单链表中是否存在结点x, 若有则返回指向x结点的指针; 否则返回NULL
LNode *GetElem(LinkList L, LNode x){
        LNode *p = L->next;
        while(p != NULL && p->data != x->data)
                p = p->next;
        return p;
}

//在线性链表的任意两个数据元素a和b间插入x
void InsertNode(LinkList L, int x){
        LNode *p = L->next;
        LNode *s = (LNode*)malloc(sizeof(LNode));
        s->data = x;
        while(p != NULL && p->data != a)
                p = p->next;
        s->next = p->next;
        p->next = s;
}

//插入操作，在表 L 中的第 i 个位置插入指定元素 e
bool Insert(LinkList L, int i, int e){
        LNode *p = L;
        int j = 0;
        LNode *s = (LNode*)malloc(sizeof(LNode));
        s->data = e;
        while(j < i-1 && p != NULL){
                p = p->next;
                j++;
        }
        s->next = p->next;
        p->next = s;
        return true;
}

```

```c
//在已知p指向a, 删除结点a的后继结点b
bool Delete(LNode *p){
        if(p != NULL){
                LNode *q = p->next
                p->next = q->next;
                free(q);
                return true;
        }
        else
                return false;
}

//动态创建单链表,设线性表n个元素已存放在数组a中,建立一个单链表,head为指向其头结点的头指针
LNode *CreateLinkedList(ElemType a[], int n){
        LNode *head = (LNode*)malloc(sizeof(LNode));
        head->next = NULL;
        for(int i=n; i>0; i--){
                LNode *s = (LNode*)malloc(sizeof(LNode));
                s->data = a[i-1];
                s->next = head->next;
                head->next = s;
        }
        return head;
}
```

1.2 循环链表

```c
typedef struct LNode{
        int data;
        struct LNode *next;
}LNode, *LinkList;

bool Empty(LinkList L){
        if(L->next == L)
                return true;
        else
                return false;
}

//统计一个由head指向其头结点的循环链表中数据元素个数的算法
int Count(LNode *head){
        LNode *p = head->next;

```

```
        int num = 0;
        while(p != head){
                num++;
                p = p->next;
        }
        return num;
}
```

1.3 双向链表

```c
typedef struct LNode{
        int data;
        struct LNode *prior, *next;
}LNode, *LinkList;l

//删除指针p指向的结点
void Delete(LNode *p){
        p->prior->next = p->next;
        p->next->prior = p->prior;
        free(p);
}

//在指针p之前插入元素x
void Insert(LNode *p, int x){
        LNode *s = (LNode*)malloc(sizeof(LNode));
        s->data = x;
        s->prior = p->prior;
        p->prior->next = s;
        s->next = p;
        p->prior = s;
}
```

1.4 一元多项式相加

```c
typedef struct Node{
        int coef, exp;
        struct Node *next;
}Node;

Node *append(Node *head, int coef, int exp){
        Node *s = (Node*)malloc(sizeof(Node));

```

```
        s->coef = coef;
        s->exp = exp;

        Node *p = head;
        while(p->next != NULL)
                p = p->next;
        p->next = s;
        return head;
}

Node *add(Node *p1, Node *p2){
        head = (Node*)malloc(sizeof(Node));
        head->next = NULL;
        p1 = p1->next;
        p2 = p2->next;

        while(p1 != NULL && p2 != NULL){
                if(p1->exp > p2->exp){
                        head = append(head, p1->coef, p1->exp);
                        p1 = p1->next;
                }else if(p1->exp < p2->exp){
                        head = append(head, p2->coef, p2->exp);
                        p2 = p2->next;
                }else{
                        head = append(head, p1->coef+p2->coef, p1->exp+p
2->exp);
                        p1 = p1->next;
                        p2 = p2->next;
                }
        }
        while(p1 != NULL){
                head = append(head, p1->coef, p1->exp);
                p1 = p1->next;
        }
        while(p2 != NULL){
                head = append(head, p2->coef, p2->exp);
                p2 = p2->next;
        }
        return head;
}
```

2 BST

```c
//二叉排序树结点
typedef struct BSTNode{
    int key;                                //数据域
    struct BSTNode  *lchild,*rchild;        //左、右孩子指针
}BSTNode,*BSTree;

//在二叉排序树中查找值为 key 的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){    //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;     //小于，则在左子树上查找
        else  T=T->rchild;              //大于，则在右子树上查找
    }
    return T;
}

//在二叉排序树中查找值为 key 的结点（递归实现）
BSTNode *BSTSearch(BSTree T,int key){
    if (T==NULL)
        return NULL;    //查找失败
    if (key==T->key)
        return T;       //查找成功
    else if (key < T->key)
        return BSTSearch(T->lchild, key);   //在左子树中找
    else
        return BSTSearch(T->rchild, key);   //在右子树中找
}
```

```c
typedef struct BSTNode{
    int key;
    struct BSTNode *lchild, *rchild;
}BSTNode, *BSTree;

BSTNode *min(BSTNode *root){
    BSTNode *p = root;
    while(p->lchild != NULL)
        p = p->lchild;
    return p;
}

//递归实现
BSTNode *BSTreeSearch(BSTNode *root, int target){
        if(root != NULL){
                if(root->key < target)
                        BSTreeSearch(root->rchild, target);
                else if(root->key > target)
```

```
                        BSTreeSearch(root->lchild, target);
        }
        return root;
}

//非递归实现
BSTNode *BSTreeSearch(BSTNode *root, int target){
        BSTNode *p = root;
        while(p != NULL && p->key != target){
                if(target < p->key)
                        p = p->lchild;
                else
                        p = p->rchild
        }
        return p;
}

//BST查找
BSTNode *Search(BSTree T, int e){
    if(e == T->key)
        return T;
    else if(T == NULL)
        return false;
    else if(e < T->key)
        Search(T->lchild, e);    //在左子树查找
    else
        Search(T->rchild, e);    //在右子树查找
}

//BST插入，非递归实现
void InsertTree(BSTNode *root, BSTNode *node){
    if(!root){
        root = node;
        root->lchild = root->rchild = NULL;
    }
    else{
        BSTNode *p, *q;
        p = root;
        while(p){
            q = p;
            if(node->key < p->key)
                p = p->lchild;
            else if(node->key > p->key)
                p = p->rchild;
        }

```

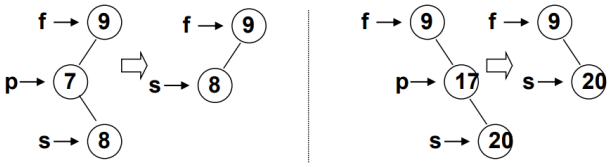
```
65          if(node→key < q→key)
66              q→lchild = node;
67          else
68              q→rchild = node;
69      }
70  }
71
72  //BST插入，递归实现
73  int InsertT(BSTNode *root, int k){
74      if(root == NULL){
75          root = (BSTNode*) malloc(sizeof(BSTNode));
76          root→key = k;
77          root→lchild = root→rchild = NULL;
78          return 1;
79      }
80      else if (k < root→key)
81          InsertT(root→lchild, k);
82      else if (k > root→key)
83          InsertT(root→rchild, k);
84      else
85          return 0;
    }
```

Deletion From A Binary Search Tree

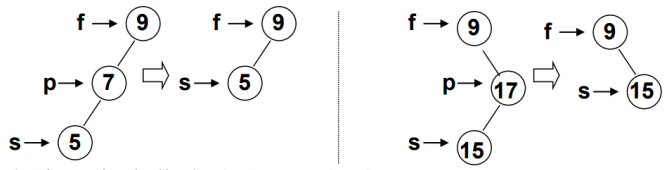
算法描述: 设p指向被删结点, f指向*p结点的双亲, s、q分别指向替代结点及其双亲. 该算法的思路是怎样找到s结点来替换p结点.

Case 1： `if (!(p->lchild) && !(p->rchild)) s = NULL;`
`if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);`

Case 2-1： `if (!(p->lchild)) s = p->rchild;`
`if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);`



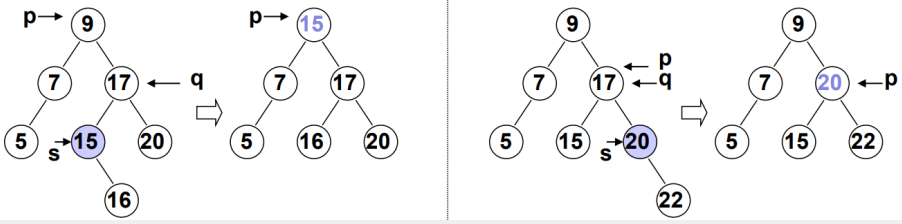
Case 2-2： `if (!(p->rchild)) s = p->lchild;`
`if (f->lchild == p) f->lchild =s; else f->rchild = s; free(p);`



上述绿颜色语句实作为公共处理部分.

Case 3: p结点有左右孩子, 则可用其前趋或后继结点s替代p. 若用后继结点, 则s应是p的右子树的最左尾结点(因为其值最小), 这时先让p结点取s结点的值(p->data=s->data); 然后再按Case2-1情况(因为此时s肯定没有左孩子)处理删除s结点. (从二叉排序树中删除一个结点的算法P231/A.9.8)

```
if (p->lchild && p->rchild) {
  q = p;  s = p->rchild;
  while (s->lchild)  { q = s; s = s->lchild; }  //确定最左尾结点
  p->data = s->data;
  if (q == p)  //特例情况: p结点的右子树没有左孩子, 如下方右图所示
    q->rchild = s->rchild;
  else q->lchild = s->rchild;
  free(s); }
```




```c
if (f→lchild == p)
        f→lchild = s;
else
        f→rchild = s;
free(p)
```