数据结构算法背诵

一、线性表

1. 逆转顺序表中的所有元素

   算法思想：第一个元素和最后一个元素对调，第二个元素和倒数第二个元素对调，……，依此类推。

```
void Reverse(int A[], int n)
{
    int i, t;
    for (i=0; i < n/2; i++)
    {
        t = A[i];
        A[i] = A[n-i-1];
        A[n-i-1] = t;
    }
}
```

2. 删除线性链表中数据域为 item 的所有结点

   算法思想：先从链表的第 2 个结点开始，从前往后依次判断链表中的所有结点是否满足条件，若某个结点的数据域为 item，则删除该结点。最后再回过头来判断链表中的第 1 个结点是否满足条件，若满足则将其删除。

```
void PurgeItem(LinkList &list)
{
    LinkList p, q = list;

    p = list->next;
    while (p != NULL)
    {
        if (p->data == item) {
            q->next = p->next;
            free(p);
            p = q->next;
        } else {
            q = p;
            p = p->next;
        }
    }
    if (list->data == item)
    {
        q = list;
        list = list->next;
        free(q);
    }
}
```

3. 逆转线性链表

```
void Reverse(LinkList &list)
{
    LinkList p, q, r;

    p = list;
    q = NULL;
    while (p != NULL)
    {
        r = q;
        q = p;
        p = p->next;
        q->next = r;
    }
    list = q;
}
```

4. 复制线性链表（递归）

```
LinkList Copy(LinkList lista)
{
    LinkList listb;

    if (lista == NULL)
        return NULL;
    else {
        listb = (LinkList)malloc(sizeof(LNode));
        listb->data = lista->data;
        listb->next = Copy(lista->next);
        return listb;
    }
}
```

5. 将两个按值有序排列的非空线性链表合并为一个按值有序的线性链表

```
LinkList MergeList(LinkList lista, LinkList listb)
{
    LinkList listc, p = lista, q = listb, r;

    // listc 指向 lista 和 listb 所指结点中较小者
    if (lista->data <= listb->data) {
        listc = lista;
        r = lista;
        p = lista->next;
    } else {
        listc = listb;
        r = listb;
        q = listb->next;
    }

    while (p != NULL && q != NULL)
    {
        if (p->data <= q->data) {
            r->next = p;
            r = p;
            p = p->next;
        } else {
            r->next = q;
            r = q;
            q = q->next;
        }
    }

    // 将剩余结点（即未参加比较的且已按升序排列的结点）链接到整个链表后面
    r->next = (p != NULL) ? p : q;
    return listc;
}
```

二、树

1. 二叉树的先序遍历（非递归算法）

算法思想：若 p 所指结点不为空，则访问该结点，然后将该结点的地址入栈，然后再将 p 指向其左孩子结点；若 p 所指向的结点为空，则从堆栈中退出栈顶元素（某个结点的地址），将 p 指向其右孩子结点。重复上述过程，直到 p = NULL 且堆栈为空，遍历结束。

```
#define MAX_STACK   50

void PreOrderTraverse(BTree T)
{
    BTree STACK[MAX_STACK], p = T;
    int   top = -1;

    while (p != NULL || top != -1)
    {
        while (p != NULL)
        {
            VISIT(p);
            STACK[++top] = p;
            p = p->lchild;
        }
        p = STACK[top--];
        p = p->rchild;
    }
}
```

2. 二叉树的中序遍历（非递归算法）

算法思想：若 p 所指结点不为空，则将该结点的地址 p 入栈，然后再将 p 指向其左孩子结点；若 p 所指向的结点为空，则从堆栈中退出栈顶元素（某个结点的地址）送 p，并访问该结点，然后再将 p 指向该结点的右孩子结点。重复上述过程，直到 p = NULL 且堆栈为空，遍历结束。

```
#define MAX_STACK   50

void InOrderTraverse(BTree T)
{
    BTree STACK[MAX_STACK], p = T;
    int   top = -1;

    while (p != NULL || top != -1);
    {
        while (p != NULL)
        {
            STACK[++top] = p;
            p = p->lchild;
        }
        p = STACK[top--];
        VISIT(p);
        p = p->rchild;
    }
}
```

3. 二叉树的后序遍历（非递归算法）

算法思想：当 p 指向某一结点时，不能马上对它进行访问，而要先访问它的左子树，因而要将此结点的地址入栈；当其左子树访问完毕后，再次搜索到该结点时（该结点地址通过退栈得到），还不能对它进行访问，还需要先访问它的右子树，所以，再一次将该结点的地址入栈。只有当该结点的右子树访问完毕后回到该结点时，才能访问该结点。为了标明某结点是否可以访问，引入一个标志变量 flag，当 flag = 0 时表示该结点暂不访问，flag = 1 时表示该结点可以访问。flag 的值随同该结点的地址一起入栈和出栈。因此，算法中设置了两个堆栈，其中 STACK1 存放结点的地址，STACK2 存放标志变量 flag，两个堆栈使用同一栈顶指针 top，且 top 的初始值为 −1。

```
#define MAX_STACK   50

void PostOrderTraverse(BTree T)
{
    BTree STACK1[MAX_STACK], p = T;
    int   STACK2[MAX_STACK], flag, top = -1;

    while (p != NULL || top != -1)
    {
        while (p != NULL) {
            STACK1[++top] = p;
            STACK2[top] = 0;
            p = p->lchild;
        }
        p = STACK1[top];
        flag = STACK2[top--];
        if (flag == 0) {
            STACK1[++top] = p;
            STACK2[top] = 1;
            p = p->rchild;
        } else {
            VISIT(p);
            p = NULL;
        }
    }
}
```

4. 二叉树的按层次遍历

算法思想：设置一个队列，首先将根结点（的地址）入队列，然后依次从队列中退出一个元素，每退出一个元素，先访问该元素所指的结点，然后依次将该结点的左孩子结点（若存在的话）和右孩子结点（若存在的话）入队列。如此重复下去，直到队列为空。

```
#define MAX_QUEUE   50

void LayeredOrderTraverse(BTree T)
{
    BTree QUEUE[MAX_QUEUE], p;
    int   front, rear;

    if (T != NULL)
    {
        QUEUE[0] = T;
        front = -1;
        rear = 0;
        while (front < rear)
        {
            p = QUEUE[++front];
            VISIT(P);
            if (p->lchild != NULL)
                QUEUE[++rear] = p->lchild;
            if (p->rchild != NULL)
                QUEUE[++rear] = p->rchild;
        }
    }
}
```

5. 建立二叉树（从键盘输入数据，先序遍历递归算法）

```
BTree CreateBT()
{
    char  ch;
    BTree T;

    sacnf("%c", &ch);
    if (ch == ' ')
        return NULL;
    else {
        T = (BTree)malloc(sizeof(BTNode));
        T->data   = ch;
        T->lchild = CreateBT();
        T->rchild = CreateBT();
        return T;
    }
}
```

6. 建立二叉树（从数组获取数据）

```
BTree CreateBT(int A[], int i, int n)
{
    BTree p;

    if (i > n)
        return NULL;
    else {
        p = (BTree)malloc(sizeof(BTNode));
        p->data = A[i];
        p->lchild = CreateBT(A, 2*i, n);
        p->rchild = CreateBT(A, 2*i+1, n);
        return p;
    }
}

T = CreateBT(A, 1, n);

------------------------------------------------------------

BTree CreateBT(int A[], int n)
{
    int    i;
    BTree *pT;

    // 对应 n 个结点申请可容纳 n 个指针变量的内存空间
    pT = (BTree *)malloc(sizeof(BTree)*n);

    // 若数组中的某个元素不等于零，则申请相应的结点空间并进行赋值
    for (i=1; i <= n; i++)
    {
        if (A[i] != 0) {
            pT[i] = (BTree)malloc(sizeof(BTNode));
            pT[i]->data = A[i];
        } else {
            pT[i] = NULL;
        }
    }

    // 修改结点的指针域的内容，使父结点指向左、右孩子结点
    for (i=1; i <= n; i++)
    {
        if (pT[i] != NULL)
        {
            pT[i]->lchild = pT[2*i];
            pT[i]->rchild = pT[2*i+1];
        }
    }
}
```

7. 求二叉树的深度（递归算法）

```
int Depth(BTree T)
{
    int ldepth, rdepth;

    if (T == NULL)
        return 0;
    else {
        ldepth = Depth(T->lchild);
        rdepth = Depth(T->rchild);
        if (ldepth > rdepth)
            return ldepth+1;
        else
            return rdepth+1;
    }
}
```

8. 求二叉树的深度（非递归算法）

算法思想：对二叉树进行遍历，遍历过程中依次记录各个结点所处的层次数以及当前已经访问过的结点所处的最大层次数。每当访问到某个叶子结点时，将该叶子结点所处的层次数与最大层次数进行比较，若前者大于后者，则修改最大层次数为该叶子结点的层次数，否则不作修改。遍历结束时，所记录的最大层次数即为该二叉树的深度。本算法使用的是非递归的中序遍历算法（其它遍历顺序也可以）。

```
#define MAX_STACK   50

int Depth(BTree T)
{
    BTree STACK1[MAX_STACK], p = T;
    int   STACK2[MAX_STACK];
    int   curdepth, maxdepth = 0, top = -1;

    if (T != NULL)
    {
        curdepth = 1;
        while (p != NULL || top != -)
        {
            while (p != NULL)
            {
                STACK1[++top] = p;
                STACK2[top] = curdepth;
                p = p->lchild;
                curdepth++;
            }
            p = STACK1[top];
            curdepth = STACK2[top--];
            if (p->lchild == NULL && p->rchild == NULL)
                if (curdepth > maxdepth)
                    maxdepth = curdepth;
            p = p->rchild;
            curdepth++;
        }
    }
    return maxdepth;
}
```

9. 求结点所在层次

算法思想：采用后序遍历的非递归算法对二叉树进行遍历，遍历过程中对每一个结点判断其是否为满足条件的结点，若是满足条件的结点，则此时堆栈中保存的元素个数再加 1 即为该结点所在的层次。

```
#define MAX_STACK   50

int LayerNode(BTree T, int item)
{
    BTree STACK1[MAX_STACK], p = T;
    int   STACK2[MAX_STACK], flag, top = -1;

    while (p != NULL || top != -1)
    {
        while (p != NULL)
        {
            STACK1[++top] = p;
            STACK2[top] = 0;
            p = p->lchild;
        }
        p = STACK1[top];
        flag = STACK2[top--];
        if (flag == 0) {
            STACK1[++top] = p;
            STACK2[top] = 1;
            p = p->rchild;
        } else {
            if (p->data == item)
                return top+2;
            p = NULL;
        }
    }
}
```

10. 交换二叉树中所有结点的左右子树的位置

算法思想：按层次遍历二叉树，遍历过程中每当访问一个结点时，就将该结点的左右子树的位置对调。

```
#define MAX_QUEUE   50

void ExchangeBT(BTree T)
{
    BTree QUEUE[MAX_QUEUE], temp, p = T;
    int   front, rear;

    if (T != NULL)
    {
        QUEUE[0] = T;
        front = -1;
        rear = 0;
        while (front < rear)
        {
            p = QUEUE[++front];
            temp = p->lchild;
            p->lchild = p->rchild;
            p->rchild = temp;
            if (p->lchild != NULL)
                QUEUE[++rear] = p->lchild;
            if (p->rchild != NULL)
                QUEUE[++rear] = p->rchild;
        }
    }
}
```

11. 删除二叉树中以某个结点为根结点的子树

算法思想：先序遍历找到符合条件的结点（其它遍历方法亦可），然后删除以该结点为根结点的子树。最后把该结点的父结点的相应的指针域置为 NULL。为此，需在算法中设置一个指针变量用以指示当前结点的父结点。

```
#define MAX_STACK   50

BTree DeleteSubtree(BTree &T, int item)
{
    BTree STACK[MAX_STACK], q, p = T;
    int   top = -1;

    if (T->data == item)
    {
        DestroyBT(T);
        T = NULL;
        return NULL;
    }
    else
    {
        while (p != NULL || top != -1)
        {
            while (p != NULL)
            {
                if (p->data == item)
                {
                    if (q->lchild == p)
                        q->lchild = NULL;
                    else
                        q->rchild = NULL;
                    DestroyBT(p);
                    return T;
                }
                STACK[++top]= p;
                q = p;
                p = p->lchild;
            }
            q = STACK[top--];
            p = q->rchild;
        }
    }
}
```

三、查找

1. 顺序查找的递归算法

```
int RecurSeqSearch(int A[], int n, int key, int i)
{
    if (i >= n)
        return -1;
    if (A[i] == key)
        return i;
    else
        return RecurSeqSearch(A, n, key, i+1);
}

pos = RecurSeqSearch(A, n, key, 0);
```

2. 折半查找

```
int BinSearch(int A[], int n, int key)
{
    int low=0, high=n-1, mid;

    while (low <= high)
    {
        mid = (low+high)/2;
        if (key == A[mid])
            return mid;
        if (key > A[mid])
            low = mid + 1;
        else
            high = mid – 1;
    }
    return -1;
}
```

3. 折半查找的递归算法

```
int RecurBinSearch(int A[], int low, int high, int key)
{
    int mid;

    if (low > high)
        return -1;
    else {
        mid = (low+high)/2;
        if (key == A[mid])
            return mid;
        if (key > A[mid])
            return RecurBinSearch(A, mid+1, high, key);
        else
            return RecurBinSearch(A, low, mid-1, key);
    }
}

pos = RecurBinSearch(A, 0, n-1, key);
```

4. 在按值递增排列且长度为 n 的线性表中折半查找并插入一元素

```
void BinInsert(int A[], int &n, int key)
{
    int j, low=0, high=n-1, mid;

    while (low <= high)
    {
        mid = (low+high)/2;
        if (key > A[mid])
            low = mid + 1;
        else
            high = mid – 1;
    }

    for (j=n; j > low; j--)
        A[j] = A[j-1];
    A[low] = key;
    n++;
}
```

5. 在按值递增排列且长度为 n 的线性表中折半查找值不小于 key 的最小元素

```
void BinSearch(int A[], int n, int key)
{
    int low=0, high=n-1, mid;

    while (low <= high)
    {
        mid = (low+high)/2;
        if (key == A[mid])
            return mid;
        if (key > A[mid])
            low = mid + 1;
        else
            high = mid – 1;
    }

    if (low <= n-1)
        return low;
    else
        return -1;
}
```

四、排序

1. 插入排序

算法思想：第 i 趟插入排序为：在含有 $i-1$ 个元素的有序子序列中插入一个元素，使之成为含有 i 个元素的有序子序列。在查找插入位置的过程中，可以同时后移元素。整个过程为进行 $n-1$ 趟插入，即先将整个序列的第 1 个元素看成是有序的，然后从第 2 个元素起逐个进行插入，直到整个序列有序为止。

```
void InsertSort(int A[], int n)
{
    int i, j, temp;

    for (i=1; i <= n-1; i++)
    {
        if (A[i] < A[i-1])
        {
            j = i-1;
            temp = A[i];
            while (j >= 0 && temp < A[j])
            {
                A[j+1] = A[j];
                j--;
            }
            A[j+1] = temp;
        }
    }
}
```

2. 折半插入排序

算法思想：算法同直接插入排序，只不过使用折半查找的方法来寻找插入位置。

```
void BinInsertSort(int A[], int n)
{
    int i, j, low, high, mid, temp;

    for (i=1; i <= n-1; i++)
    {
        temp = A[i];
        low  = 0;
        high = i – 1;
        while (low <= high)
        {
            mid = (low+high)/2;
            if (temp > A[mid])
                low = mid + 1;
            else
                high = mid – 1;
        }
        for (j=i; j > low; j--)
            A[j] = A[j-1];
        A[low] = temp;
    }
}
```

3. 冒泡排序

算法思想：首先将第 1 个元素和第 2 个元素进行比较，若前者大于后者，则两者交换位置，然后比较第 2 个元素和第 3 个元素。依此类推，直到第 $n-1$ 个元素和第 n 个元素进行过比较或交换为止。上述过程称为一趟冒泡排序，其结果是使得 n 个元素中值最大的那个元素被安排在最后一个元素的位置上。然后进行第二趟排序，即对前 $n-1$ 个元素进行同样的操作，使得前 $n-1$ 个元素中值最大的那个元素被安排在第 $n-1$ 个位置上。一般地，第 i 趟冒泡排序是从前 $n-i+1$ 个元素中的第 1 个元素开始，两两比较，若前者大于后者，则交换，结果使得前 $n-i+1$ 个元素中最大的元素被安排在第 $n-i+1$ 个位置上。显然，判断冒泡排序结束的条件是“在一趟排序中没有进行过交换元素的操作”，为此，设立一个标志变量 flag，flag = 1 表示有过交换元素的操作，flag = 0 表示没有过交换元素的操作，在每一趟排序开始前，将 flag 置为 0，在排序过程中，只要有交换元素的操作，就及时将 flag 置为 1。因为至少要执行一趟排序操作，故第一趟排序时，flag = 1。

```
void BubbleSort(int A[], int n)
{
    int i, j, temp, flag = 1;

    for (i=n-1; i >= 1 && flag == 1; i--)
    {
        flag = 0;
        for (j=0; j < i; j++)
        {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                flag = 1;
            }
        }
    }
}
```

4. 选择排序

算法思想：第 i 趟排序从序列的后 $n-i+1$（$i = 1, 2, \cdots, n-1$）个元素中选择一个值最小的元素与该个元素的第 1 个元素交换位置，即与整个序列的第 i 个元素交换。依此类推，直到 $i = n-1$ 为止。也就是说，每一趟排序从从未排好序的那些元素中选择一个值最小的元素，然后将其与这些未排好序的元素中的第 1 个元素交换位置。

```
void SelectSort(int A[], int n)
{
    int i, j, min, temp;

    for (i=0; i < n; i++)
    {
        min = i;
        for (j=i+1; j < n; j++)
        {
            if (A[min] > A[j])
                min = j;
        }
        if (min != i)
        {
            temp = A[min];
            A[min] = A[i];
            A[i] = temp;
        }
    }
}
```

5. 快速排序

算法思想：在参加排序的序列中任意选择一个元素（通常称为分界元素或基准元素），把小于或等于分界元素的所有元素都移到分界元素的前面，把大于分界元素的所有元素都移到分界元素的后面，这样，当前参加排序的序列就被划分成前后两个子序列，其中前一个子序列中的所有元素都小于后一个子序列的所有元素，并且分界元素正好处于排序的最终位置上。然后分别对这两个子序列递归地进行上述排序过程，直到所有元素都处于排序的最终位置上，排序结束。

```
void QuickSort(int A[], int n)
{
    QSort(A, 0, n-1);
}

void QSort(int A[], int low, int high)
{
    int pivotloc;

    if (low < high)
    {
        pivot = Partition(A, low, high);
        QSort(A, low, pivotloc-1);
        QSort(A, pivotloc+1, high);
    }
}

int Partition(int A[], int low, int high)
{
    int pivot;

    pivot = A[low];
    // 从线性表的两端交替地向中间扫描
    while (low < high)
    {
        while (low < high && A[high] >= pivot)
            high--;
        A[low] = A[high];
        while (low < high && A[low] <= pivot)
            low++;
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}
```

6. 堆排序

```
void HeapSort(int A[], int n)
{
    int i, temp;

    // 建立大顶堆
    for (i = n/2; i >= 1; i--)
        HeapAdjust(A,i,n);

    for (i = n-1; i >= 1; i--)
    {
        temp   = A[1];
        A[1]   = A[i+1];
        A[i+1] = temp;
        // 将 A[1..i] 重新调整为大顶堆
        HeapAdjust(A,1,i);
    }
}

void HeapAdjust(int A[], int low, int high)
{
    int i, temp;

    temp = A[low];
    for (i=2*low; i <= high; i=i*2)
    {
        // 令 i 为关键字较大的记录的下标
        if (i < high && A[i] < A[i+1])
            i++;
        if (temp >= A[i])
            break;
        else {
            A[low] = A[i];
            low = i;
        }
    }
    A[low] = temp;      // 插入
}
```