数据结构代码-2024


```c
//单链表的数据类型定义
typedef int ElemType;
typedef struct LNode{ //定义单链表结点类型
    ElemType data;          //数据域
    struct LNode *next;     //指针域
}LNode,*LinkList;

//单链表逆序
LinkList reverseList(LinkList &L) {
    LinkList p = NULL;          // 反转后的链表的头节点
    LinkList current = L->next;  // 当前节点，初始化为链表的第一个节点
    L->next = NULL;             // 原链表的头节点将成为反转后链表的尾节点
    while (current != NULL) {
        LinkList n = current->next;  // 保存当前节点的下一个节点
        current->next = p;           // 反转当前节点的指针
        p = current;                 // 更新反转后的链表的头节点
        current = n;                 // 移动到下一个节点
    }
    L->next = p;  // 将反转后的链表挂接到原链表的头节点
    return L;     // 返回反转后的链表
}

//统计data值为x的结点数
int CountX(LinkList L, int x) {
    int count = 0;
    LinkList p = L->next;
    while (p != NULL) {
        if (p->data == x) {
            count++;
        }
        p = p->next; // 移动到下一个节点
    }
    return count;
}

//在链表第一个值为x的第一个元素前插入值为e的结点的算法
LinkList Inserty(LinkList &L, int x, int e) {
    LinkList p = L->next; // 从链表的第一个有效节点开始遍历
    while (p != NULL && p->next != NULL) {
        if (p->next->data == x) {
            LinkList s;
            s = (LinkList)malloc(sizeof(LNode));  // 为新节点分配内存
            s->data = e;                          // 设置新节点的值为 e
            s->next = p->next;                    // 新节点 s 指向 p 的下一个节点
            p->next = s;                          // 当前节点 p 指向新节点 s
            return L;
        }
        p = p->next; // 移动到下一个节点
    }
    return L;
}
```

```c
//在链表第一个值为x的第一个元素后插入值为e的结点的算法
LinkList InsetElem(LinkList &L, int x, int e) {
    LinkList p = L->next; // 从链表的第一个有效节点开始遍历
    while (p != NULL && p->next != NULL) {
        if (p->data == x) {  // 如果当前节点 p 的值等于 x
            LinkList s;                  // 创建一个新节点 s
            s = (LinkList)malloc(sizeof(LNode));  // 为新节点分配内存
            s->data = e;                          // 设置新节点的值为 e
            s->next = p->next;                    // 新节点 s 指向 p 的下一个节点
            p->next = s;                          // 当前节点 p 指向新节点 s
            return L;
        }
        p = p->next;  // 移动到下一个节点
    }
    return L;
}
```

```c
//删除单链表中元素值最大的结点
void DeleteMaxNode(LinkList &L) {
    if (L == NULL || L->next == NULL) {
        return;
    }
    LinkList maxPrev = L;  // 最大节点的前驱节点
    LinkList p = L->next;  // 当前节点
    // 遍历链表，找到最大值节点的前驱节点
    while (p->next != NULL) {
        if (p->next->data > maxPrev->next->data) {
            maxPrev = p;
        }
        p = p->next;
    }
    // 删除最大值节点
    LinkList maxNode = maxPrev->next;  // 最大值节点
    maxPrev->next = maxNode->next;     // 跳过最大值节点
    free(maxNode);                     // 释放最大值节点的内存
}
```

```c
//删除单链表中值重复的结点(排好序)
LinkList DeletRepeatElem(LinkList &L) {
    if (L == NULL || L->next == NULL) {
        return L;  // 如果链表为空或只有一个节点，直接返回
    }
    LinkList p = L->next;  // 从链表的第一个有效节点开始遍历

    while (p != NULL && p->next != NULL) {
        if (p->data == p->next->data) {  // 如果当前节点与下一个节点的值相同
            LinkList t = p->next;        // 保存重复节点
            p->next = p->next->next;     // 跳过重复节点
            free(t);                     // 释放重复节点的内存
        } else {
            p = p->next;  // 移动到下一个节点
        }
    }
    return L;  // 返回修改后的链表
}
```

```c
//删除单链表中在给定范围的元素
void DeleteRange(LinkList &L, int min, int max) {
    if (L == NULL || L->next == NULL) {
        return;
    }
    LinkList p = L;  // 当前节点的前驱节点
    while (p->next != NULL) {
        // 如果当前节点的值在 [min, max] 范围内
        if (p->next->data >= min && p->next->data <= max) {
            LinkList temp = p->next;  // 保存要删除的节点
            p->next = temp->next;     // 跳过要删除的节点
            free(temp);               // 释放内存
        } else {
            p = p->next;  // 移动到下一个节点
        }
    }
}

//两个有序单链表的合并
LinkList Merge(LinkList A, LinkList B) {
    // 创建新链表 C 的头节点
    LinkList C = (LinkList)malloc(sizeof(LNode));
    C->next = NULL;  // 初始化新链表的 next 指针为 NULL
    LinkList r = C;  // r 指向新链表的最后一个节点
    LinkList a = A->next;  // a 指向链表 A 的第一个有效节点
    LinkList b = B->next;  // b 指向链表 B 的第一个有效节点
    while (a != NULL && b != NULL) {
        if (a->data < b->data) {
            r->next = a;  // 将链表 A 的当前节点插入新链表 C
            a = a->next;  // 移动链表 A 的指针
        } else {
            r->next = b;  // 将链表 B 的当前节点插入新链表 C
            b = b->next;  // 移动链表 B 的指针
        }
        r = r->next;  // 移动新链表 C 的指针
    }
    // 处理链表 A 或链表 B 剩余的节点
    if (a != NULL) {
        r->next = a;  // 将链表 A 剩余的节点接入新链表 C
    } else {
        r->next = b;  // 将链表 B 剩余的节点接入新链表 C
    }
    return C;
}
```

```c
//二叉链表的存储结构
typedef struct BiTNode {
    int data;                       // 数据域
    struct BiTNode *lchild;         // 左孩子指针
    struct BiTNode *rchild;         // 右孩子指针
} BiTNode, *BiTree;


//判断两棵树是否相似
bool isSimilar(BiTree T1, BiTree T2) {
    // 如果两棵树都为空，则相似
    if (T1 == NULL && T2 == NULL) {
        return true;
    }
    // 如果一棵树为空，另一棵不为空，则不相似
    if (T1 == NULL || T2 == NULL) {
        return false;
    }
    // 递归判断左子树和右子树是否相似
    return isSimilar(T1->lchild, T2->lchild) && isSimilar(T1->rchild, T2->rchild);
}


//由二叉树b递归复制成另一棵二叉树t的算法
void Copy(BiTree T, BiTree *NewT) {
    if (T == NULL) {
        *NewT = NULL;  // 将新树的当前节点指针设为 NULL
        return;
    }
    *NewT = (BiTree)malloc(sizeof(BiTNode));    // 创建新节点
    (*NewT)->data = T->data;  // 将原树当前节点的数据复制到新节点
    Copy(T->lchild, &((*NewT)->lchild));  // 递归复制原树的左子树到新树的左子树
    Copy(T->rchild, &((*NewT)->rchild));  // 递归复制原树的右子树到新树的右子树
}


//计算给定二叉树中结点为x的结点个数
int countNodes(BiTree root, int x) {
    if (root == NULL) {
        return 0;
    }
    int count = 0;
    if (root->data == x) {
        count = 1;
    }
    count += countNodes(root->lchild, x);
    count += countNodes(root->rchild, x);
    return count;
}
```

```c
//输出一棵给定二叉树的度为2的结点的个数。
int countDegree2(BiTree root) {
    if (root == NULL) {
        return 0;
    }
    // 检查当前节点是否有左孩子和右孩子
    int isDegreeTwo = 0; // 标记当前节点是否为度为 2 的节点
    if (root->lchild != NULL && root->rchild != NULL) {
        isDegreeTwo = 1; // 如果既有左孩子又有右孩子，标记为 1
    }
    // 递归统计左子树中度为 2 的节点个数
    int leftCount = countDegree2(root->lchild);
    // 递归统计右子树中度为 2 的节点个数
    int rightCount = countDegree2(root->rchild);
    return isDegreeTwo + leftCount + rightCount;
}

//计算二叉树中叶子结点的数目
int CountLeaf(BiTree T) {
    if (T == NULL) {
        return 0;
    }
    // 如果当前节点是叶子节点，返回 1
    if (T->lchild == NULL && T->rchild == NULL) {
        return 1;
    }
    // 递归统计左子树和右子树的叶子节点数
    return CountLeaf(T->lchild) + CountLeaf(T->rchild);
}

//设计一个算法求二叉树中最小值的结点值
int findMinValue(BiTree root) {
    if (root == NULL) {
        return INT_MAX;
    }
    // 获取当前节点的值
    int currentValue = root->data;
    // 递归查找左子树的最小值
    int leftMin = findMinValue(root->lchild);
    // 递归查找右子树的最小值
    int rightMin = findMinValue(root->rchild);
    // 返回当前节点、左子树、右子树中的最小值
    if (leftMin < currentValue) {
        currentValue = leftMin;
    }
    if (rightMin < currentValue) {
        currentValue = rightMin;
    }
    return currentValue;
}
```

```
//编写算法在二叉树中查找数据元素值为x的结点。
BiTree findNode(BiTree root, int x) {
    if (root == NULL) {
        return NULL;
    }
    // 如果当前节点的值等于 x，返回当前节点
    if (root->data == x) {
        return root;
    }
    // 递归查找左子树
    BiTree leftResult = findNode(root->lchild, x);
    if (leftResult != NULL) {
        return leftResult; // 如果在左子树中找到，直接返回
    }
    // 递归查找右子树
    BiTree rightResult = findNode(root->rchild, x);
    if (rightResult != NULL) {
        return rightResult; // 如果在右子树中找到，直接返回
    }
    // 如果左右子树都没有找到，返回 NULL
    return NULL;
}

//求二叉树高度
int getHeight(BiTree root) {
    if (root == NULL) {
        return 0;
    }
    // 递归计算左子树的高度
    int leftTreeHeight = getHeight(root->lchild);
    // 递归计算右子树的高度
    int rightTreeHeight = getHeight(root->rchild);
    // 比较左子树和右子树的高度，取较大值，然后加 1（当前节点的高度）
    if (leftTreeHeight > rightTreeHeight) {
        return leftTreeHeight + 1;
    } else {
        return rightTreeHeight + 1;
    }
}
```

```c
//二叉排序树的类型定义
typedef struct BSTNode {
    int data;                       // 数据域
    struct BSTNode *lchild;         // 左孩子指针
    struct BSTNode *rchild;         // 右孩子指针
} BSTNode, *BSTree;

//二叉排序树的查找
BSTree searchBST(BSTree root, int key) {
    // 如果当前节点为空，或者当前节点的值等于 key，返回当前节点
    if (root == NULL || root->data == key) {
        return root;
    }
    // 如果 key 小于当前节点的值，递归查找左子树
    if (key < root->data) {
        return searchBST(root->lchild, key);
    }
    // 如果 key 大于当前节点的值，递归查找右子树
    return searchBST(root->rchild, key);
}

//统计二叉树中key值大于给定值的结点数
int countGreater(BiTree root, int key) {
    if (root == NULL) {
        return 0;
    }
    int count = 0;
    // 如果当前节点的值大于 key，计数器加 1
    if (root->data > key) {
        count = 1;
    }
    // 递归统计左子树中大于 key 的节点数
    count += countGreater(root->lchild, key);
    // 递归统计右子树中大于 key 的节点数
    count += countGreater(root->rchild, key);
    return count;
}
```

```c
//有序表的折半查找
int binarySearch(int a[], int n, int key) {
    int low = 0;                    // 查找区间的起始位置
    int high = n - 1;               // 查找区间的结束位置
    while (low <= high) {
        int mid = (high + low) / 2; // 计算中间位置
        if (a[mid] == key) {
            return mid; // 找到目标值，返回下标
        } else if (a[mid] < key) {
            low = mid + 1; // 目标值在右半部分
        } else {
            high = mid - 1; // 目标值在左半部分
        }
    }
    return -1; // 未找到目标值，返回 -1
}
```

```c
//假设图G采用邻接表存储，设计一个算法，判断无向图G是否连通
#define MAX_VERTEX 100
// 定义邻接表的边节点
typedef struct EdgeNode {
    int adjvex;                 // 邻接顶点下标
    struct EdgeNode *next;      // 指向下一个邻接顶点
} EdgeNode;

// 定义顶点节点
typedef struct VertexNode {
    int data;                   // 顶点数据（可选）
    EdgeNode *firstEdge;        // 指向第一个邻接顶点
} VertexNode, AdjList[MAX_VERTEX];


// 定义图结构
typedef struct {
    AdjList adjList;            // 邻接表
    int vertexNum;              // 顶点数
    int edgeNum;                // 边数
} Graph;


// DFS 遍历
void DFS(Graph *G, int v, bool visited[]) {
    visited[v] = true; // 标记当前节点为已访问

    // 遍历当前节点的所有邻接节点
    EdgeNode *p = G->adjList[v].firstEdge;
    while (p != NULL) {
        if (!visited[p->adjvex]) {
            DFS(G, p->adjvex, visited); // 递归访问未访问的邻接节点
        }
        p = p->next;
    }
}

// 判断图是否连通
bool isConnected(Graph *G) {
    if (G->vertexNum == 0) {
        return true; // 空图默认连通
    }
    // 初始化访问标记数组
    bool visited[MAX_VERTEX];
    for (int i = 0; i < G->vertexNum; i++) {
        visited[i] = false;
    }
    // 从第一个顶点开始 DFS 遍历
    DFS(G, 0, visited);
    // 检查是否所有顶点都被访问过
    for (int i = 0; i < G->vertexNum; i++) {
        if (!visited[i]) {
            return false; // 如果有未访问的顶点，图不连通
        }
    }
    return true; // 所有顶点都被访问过，图连通
}
```