



```

4 //单链表的数据类型定义
5 typedef int ElemType;
6 typedef struct LNode{ //定义单链表结点类型
7     ElemType data;           //数据域
8     struct LNode *next;     //指针域
9 }LNode,*LinkList;

11 //单链表逆序
12 LinkList reverseList(LinkList &L) {
13     LinkList p = NULL;      // 反转后的链表的头节点
14     LinkList current = L->next; // 当前节点，初始化为链表的第一个节点
15     L->next = NULL;         // 原链表的头节点将成为反转后链表的尾节点
16     while (current != NULL) {
17         LinkList n = current->next; // 保存当前节点的下一个节点
18         current->next = p;          // 反转当前节点的指针
19         p = current;              // 更新反转后的链表的头节点
20         current = n;              // 移动到下一个节点
21     }
22     L->next = p; // 将反转后的链表挂接到原链表的头节点
23     return L; // 返回反转后的链表
24 }

26 //统计data值为x的结点数
27 int CountX(LinkList L, int x) {
28     int count = 0;
29     LinkList p = L->next;
30     while (p != NULL) {
31         if (p->data == x) {
32             count++;
33         }
34         p = p->next; // 移动到下一个节点
35     }
36     return count;
37 }

39 //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
40 LinkList Inserty(LinkList &L, int x, int e) {
41     LinkList p = L->next; // 从链表的第一个有效节点开始遍历
42     while (p != NULL && p->next != NULL) {
43         if (p->next->data == x) {
44             LinkList s;
45             s = (LinkList)malloc(sizeof(LNode)); // 为新节点分配内存
46             s->data = e;                         // 设置新节点的值为 e
47             s->next = p->next;                  // 新节点 s 指向 p 的下一个节点
48             p->next = s;                      // 当前节点 p 指向新节点 s
49             return L;
50         }
51         p = p->next; // 移动到下一个节点
52     }
53     return L;
54 }

```

```
56 //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
57 LinkList InsetElem(LinkList &L, int x, int e) {
58     LinkList p = L->next; // 从链表的第一个有效节点开始遍历
59     while (p != NULL && p->next != NULL) {
60         if (p->data == x) { // 如果当前节点 p 的值等于 x
61             LinkList s; // 创建一个新节点 s
62             s = (LinkList)malloc(sizeof(LNode)); // 为新节点分配内存
63             s->data = e; // 设置新节点的值为 e
64             s->next = p->next; // 新节点 s 指向 p 的下一个节点
65             p->next = s; // 当前节点 p 指向新节点 s
66             return L;
67         }
68         p = p->next; // 移动到下一个节点
69     }
70     return L;
71 }
```

```
73 //删除单链表中元素值最大的结点
74 void DeleteMaxNode(LinkList &L) {
75     if (L == NULL || L->next == NULL) {
76         return;
77     }
78     LinkList maxPrev = L; // 最大节点的前驱节点
79     LinkList p = L->next; // 当前节点
80     // 遍历链表，找到最大值节点的前驱节点
81     while (p->next != NULL) {
82         if (p->next->data > maxPrev->next->data) {
83             maxPrev = p;
84         }
85         p = p->next;
86     }
87     // 删除最大值节点
88     LinkList maxNode = maxPrev->next; // 最大值节点
89     maxPrev->next = maxNode->next; // 跳过最大值节点
90     free(maxNode); // 释放最大值节点的内存
91 }
```

```
93 //删除单链表中值重复的结点(排好序)
94 LinkList DeleteRepeatElem(LinkList &L) {
95     if (L == NULL || L->next == NULL) {
96         return L; // 如果链表为空或只有一个节点，直接返回
97     }
98     LinkList p = L->next; // 从链表的第一个有效节点开始遍历
99
100    while (p != NULL && p->next != NULL) {
101        if (p->data == p->next->data) { // 如果当前节点与下一个节点的值相同
102            LinkList t = p->next; // 保存重复节点
103            p->next = p->next->next; // 跳过重复节点
104            free(t); // 释放重复节点的内存
105        } else {
106            p = p->next; // 移动到下一个节点
107        }
108    }
109    return L; // 返回修改后的链表
110 }
```

```

112 //删除单链表中在给定范围的元素
113 void DeleteRange(LinkList &L, int min, int max) {
114     if (L == NULL || L->next == NULL) {
115         return;
116     }
117     LinkList p = L; // 当前节点的前驱节点
118     while (p->next != NULL) {
119         // 如果当前节点的值在 [min, max] 范围内
120         if (p->next->data >= min && p->next->data <= max) {
121             LinkList temp = p->next; // 保存要删除的节点
122             p->next = temp->next; // 跳过要删除的节点
123             free(temp); // 释放内存
124         } else {
125             p = p->next; // 移动到下一个节点
126         }
127     }
128 }

```

```

130 //两个有序单链表的合并
131 LinkList Merge(LinkList A, LinkList B) {
132     // 创建新链表 C 的头节点
133     LinkList C = (LinkList)malloc(sizeof(LNode));
134     C->next = NULL; // 初始化新链表的 next 指针为 NULL
135     LinkList r = C; // r 指向新链表的最后一个节点
136     LinkList a = A->next; // a 指向链表 A 的第一个有效节点
137     LinkList b = B->next; // b 指向链表 B 的第一个有效节点
138     while (a != NULL && b != NULL) {
139         if (a->data < b->data) {
140             r->next = a; // 将链表 A 的当前节点插入新链表 C
141             a = a->next; // 移动链表 A 的指针
142         } else {
143             r->next = b; // 将链表 B 的当前节点插入新链表 C
144             b = b->next; // 移动链表 B 的指针
145         }
146         r = r->next; // 移动新链表 C 的指针
147     }
148     // 处理链表 A 或链表 B 剩余的节点
149     if (a != NULL) {
150         r->next = a; // 将链表 A 剩余的节点接入新链表 C
151     } else {
152         r->next = b; // 将链表 B 剩余的节点接入新链表 C
153     }
154     return C;
155 }

```

```

157 //二叉链表的存储结构
158 typedef struct BiTNode {
159     int data;                                // 数据域
160     struct BiTNode *lchild;                 // 左孩子指针
161     struct BiTNode *rchild;                 // 右孩子指针
162 } BiTNode, *BiTree;
163
164 //判断两棵树是否相似
165 bool isSimilar(BiTree T1, BiTree T2) {
166     // 如果两棵树都为空，则相似
167     if (T1 == NULL && T2 == NULL) {
168         return true;
169     }
170     // 如果一棵树为空，另一棵不为空，则不相似
171     if (T1 == NULL || T2 == NULL) {
172         return false;
173     }
174     // 递归判断左子树和右子树是否相似
175     return isSimilar(T1->lchild, T2->lchild) && isSimilar(T1->rchild, T2->rchild);
176 }
177
178 //由二叉树b递归复制成另一棵二叉树t的算法
179 void Copy(BiTree T, BiTree *NewT) {
180     if (T == NULL) {
181         *NewT = NULL; // 将新树的当前节点指针设为 NULL
182         return;
183     }
184     *NewT = (BiTree)malloc(sizeof(BiTNode)); // 创建新节点
185     (*NewT)->data = T->data; // 将原树当前节点的数据复制到新节点
186     Copy(T->lchild, &(*NewT)->lchild); // 递归复制原树的左子树到新树的左子树
187     Copy(T->rchild, &(*NewT)->rchild); // 递归复制原树的右子树到新树的右子树
188 }
189
190 //计算给定二叉树中结点为x的结点个数
191 int countNodes(BiTree root, int x) {
192     if (root == NULL) {
193         return 0;
194     }
195     int count = 0;
196     if (root->data == x) {
197         count = 1;
198     }
199     count += countNodes(root->lchild, x);
200     count += countNodes(root->rchild, x);
201     return count;
202 }

```

```
204 //输出一棵给定二叉树的度为2的结点的个数。
205 int countDegree2(BiTTree root) {
206     if (root == NULL) {
207         return 0;
208     }
209     // 检查当前节点是否有左孩子和右孩子
210     int isDegreeTwo = 0; // 标记当前节点是否为度为 2 的节点
211     if (root->lchild != NULL && root->rchild != NULL) {
212         isDegreeTwo = 1; // 如果既有左孩子又有右孩子，标记为 1
213     }
214     // 递归统计左子树中度为 2 的节点个数
215     int leftCount = countDegree2(root->lchild);
216     // 递归统计右子树中度为 2 的节点个数
217     int rightCount = countDegree2(root->rchild);
218     return isDegreeTwo + leftCount + rightCount;
219 }
```

```
221 //计算二叉树中叶子结点的数目
222 int CountLeaf(BiTTree T) {
223     if (T == NULL) {
224         return 0;
225     }
226     // 如果当前节点是叶子节点，返回 1
227     if (T->lchild == NULL && T->rchild == NULL) {
228         return 1;
229     }
230     // 递归统计左子树和右子树的叶子节点数
231     return CountLeaf(T->lchild) + CountLeaf(T->rchild);
232 }
```

```
235 //设计一个算法求二叉树中最小值的结点值
236 int findMinValue(BiTTree root) {
237     if (root == NULL) {
238         return INT_MAX;
239     }
240     // 获取当前节点的值
241     int currentValue = root->data;
242     // 递归查找左子树的最小值
243     int leftMin = findMinValue(root->lchild);
244     // 递归查找右子树的最小值
245     int rightMin = findMinValue(root->rchild);
246     // 返回当前节点、左子树、右子树中的最小值
247     if (leftMin < currentValue) {
248         currentValue = leftMin;
249     }
250     if (rightMin < currentValue) {
251         currentValue = rightMin;
252     }
253     return currentValue;
254 }
```

```
256 //编写算法在二叉树中查找数据元素值为x的结点。
257 BiTree findNode(BiTree root, int x) {
258     if (root == NULL) {
259         return NULL;
260     }
261     // 如果当前节点的值等于 x, 返回当前节点
262     if (root->data == x) {
263         return root;
264     }
265     // 递归查找左子树
266     BiTree leftResult = findNode(root->lchild, x);
267     if (leftResult != NULL) {
268         return leftResult; // 如果在左子树中找到, 直接返回
269     }
270     // 递归查找右子树
271     BiTree rightResult = findNode(root->rchild, x);
272     if (rightResult != NULL) {
273         return rightResult; // 如果在右子树中找到, 直接返回
274     }
275     // 如果左右子树都没有找到, 返回 NULL
276     return NULL;
277 }
```

```
279 //求二叉树高度
280 int getHeight(BiTree root) {
281     if (root == NULL) {
282         return 0;
283     }
284     // 递归计算左子树的高度
285     int leftTreeHeight = getHeight(root->lchild);
286     // 递归计算右子树的高度
287     int rightTreeHeight = getHeight(root->rchild);
288     // 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)
289     if (leftTreeHeight > rightTreeHeight) {
290         return leftTreeHeight + 1;
291     } else {
292         return rightTreeHeight + 1;
293     }
294 }
```

```
296 //二叉排序树的类型定义
297 typedef struct BSTNode {
298     int data;                                // 数据域
299     struct BSTNode *lchild;                // 左孩子指针
300     struct BSTNode *rchild;                // 右孩子指针
301 } BSTNode, *BSTree;
302
303 //二叉排序树的查找
304 BSTree searchBST(BSTree root, int key) {
305     // 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
306     if (root == NULL || root->data == key) {
307         return root;
308     }
309     // 如果 key 小于当前节点的值, 递归查找左子树
310     if (key < root->data) {
311         return searchBST(root->lchild, key);
312     }
313     // 如果 key 大于当前节点的值, 递归查找右子树
314     return searchBST(root->rchild, key);
315 }
316
317 //统计二叉树中key值大于给定值的结点数
318 int countGreater(BiTTree root, int key) {
319     if (root == NULL) {
320         return 0;
321     }
322     int count = 0;
323     // 如果当前节点的值大于 key, 计数器加 1
324     if (root->data > key) {
325         count = 1;
326     }
327     // 递归统计左子树中大于 key 的节点数
328     count += countGreater(root->lchild, key);
329     // 递归统计右子树中大于 key 的节点数
330     count += countGreater(root->rchild, key);
331     return count;
332 }
```

```
334 //有序表的折半查找
335 int binarySearch(int a[], int n, int key) {
336     int low = 0;                      // 查找区间的起始位置
337     int high = n - 1;                 // 查找区间的结束位置
338     while (low <= high) {
339         int mid = (high + low) / 2;   // 计算中间位置
340         if (a[mid] == key) {
341             return mid; // 找到目标值，返回下标
342         } else if (a[mid] < key) {
343             low = mid + 1; // 目标值在右半部分
344         } else {
345             high = mid - 1; // 目标值在左半部分
346         }
347     }
348     return -1; // 未找到目标值，返回 -1
349 }
```

```
351 //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
352 #define MAX_VERTEX 100
353 // 定义邻接表的边节点
354 typedef struct EdgeNode {
355     int adjvex;           // 邻接顶点下标
356     struct EdgeNode *next; // 指向下一个邻接顶点
357 } EdgeNode;
358
359 // 定义顶点节点
360 typedef struct VertexNode {
361     int data;             // 顶点数据（可选）
362     EdgeNode *firstEdge; // 指向第一个邻接顶点
363 } VertexNode, AdjList[MAX_VERTEX];
364
365 // 定义图结构
366 typedef struct {
367     AdjList adjList;      // 邻接表
368     int vertexNum;        // 顶点数
369     int edgeNum;          // 边数
370 } Graph;
371
372 // DFS 遍历
373 void DFS(Graph *G, int v, bool visited[]) {
374     visited[v] = true; // 标记当前节点为已访问
375
376     // 遍历当前节点的所有邻接节点
377     EdgeNode *p = G->adjList[v].firstEdge;
378     while (p != NULL) {
379         if (!visited[p->adjvex]) {
380             DFS(G, p->adjvex, visited); // 递归访问未访问的邻接节点
381         }
382         p = p->next;
383     }
384 }
385
386 // 判断图是否连通
387 bool isConnected(Graph *G) {
388     if (G->vertexNum == 0) {
389         return true; // 空图默认连通
390     }
391     // 初始化访问标记数组
392     bool visited[MAX_VERTEX];
393     for (int i = 0; i < G->vertexNum; i++) {
394         visited[i] = false;
395     }
396     // 从第一个顶点开始 DFS 遍历
397     DFS(G, 0, visited);
398     // 检查是否所有顶点都被访问过
399     for (int i = 0; i < G->vertexNum; i++) {
400         if (!visited[i]) {
401             return false; // 如果有未访问的顶点，图不连通
402         }
403     }
404     return true; // 所有顶点都被访问过，图连通
405 }
```