

> 一研为定

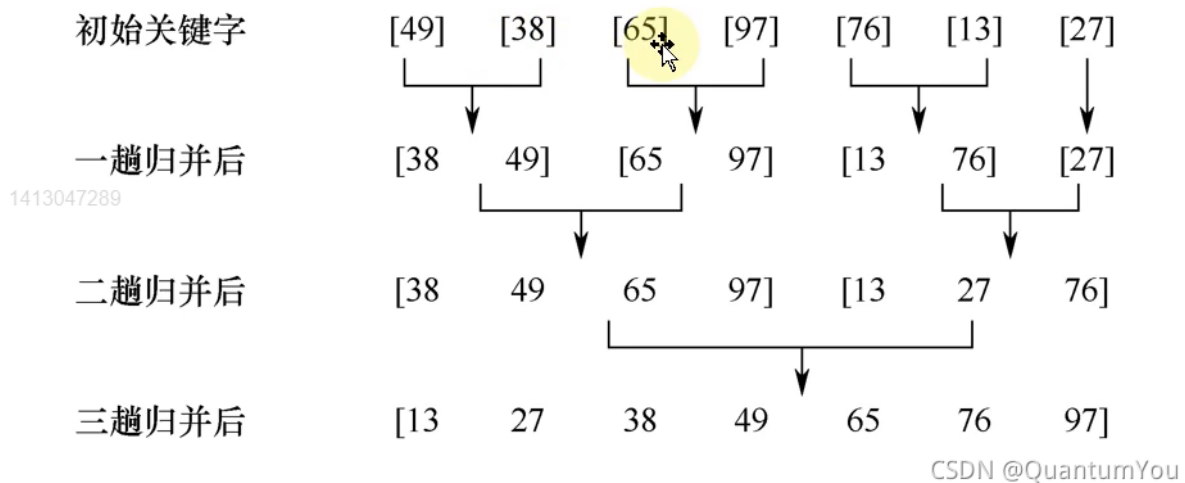
@[toc]

高级第一次 直播 归并排序

关于数组下标越界的解决办法：

```
1  if(j==1){
2      break ;    // 在边界进行判断
3  }
```

归并排序



归并排序代码

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N 7
5  typedef int ElementType;
6  //49,38,65,97,76,13,27
7  void Merge(ElementType A[],int low,int mid,int high)
8  {
9      ElementType B[N];//为了降低操作次数
10     int i,j,k;
11     for(k=low;k<=high;k++)//复制元素到B中
12         B[k]=A[k];
13     for(i=low,j=mid+1,k=i;i<=mid&& j<=high;k++)//合并两个有序数组
14     {
15         if(B[i]<=B[j])
16             A[k]=B[i++];
17         else
18             A[k]=B[j++];
19     }
20     while(i<=mid)//如果有剩余元素，接着放入即可
21         A[k++]=B[i++];
22     while(j<=high)
```

```

23     A[k++]=B[j++];
24 }
25 //归并排序不限制是两两归并，还是多个归并
26 // 1 3 5 7 9
27 // 2 4
28 // 1 2 3 4 5 7 9 主要的代码逻辑
29 void MergeSort(ElemType A[],int low,int high)//递归分割
30 {
31     if(low<high)
32     {
33         int mid=(low+high)/2;
34         MergeSort(A,low,mid);
35         MergeSort(A,mid+1,high);
36         Merge(A,low,mid,high);
37     }
38 }
39 void print(int* a)
40 {
41     for(int i=0;i<N;i++)
42     {
43         printf("%3d",a[i]);
44     }
45     printf("\n");
46 }
47
48 // 归并排序
49 int main()
50 {
51     int A[7]={49,38,65,97,76,13,27}; //数组，7个元素
52     MergeSort(A,0,6);
53     print(A);
54     system("pause");
55 }

```

各大算法时间复杂度

- **快排算法**：最坏情况，时间复杂度为 $O(n^2)$ ，即为数组本身有序的情况，解决办法使用随机数

[基数排序](#)

计数排序

高级第二次 直播 图

图

- 图G由顶点集 V 和边集 E 组成，记为 $G=(V,E)$ 其中 $V(G)$ 表示图G中顶点的有限非空集： $E(G)$ 表示图G中顶点之间的关系（边）集合

图的存储方法

- 邻接矩阵，邻接表

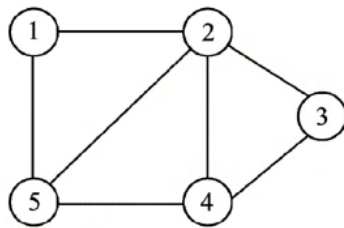
图的邻接矩阵存储结构定义如下：

```
#define MaxVertexNum 100 //顶点数目的最大值
typedef char VertexType; //顶点的数据类型
typedef int EdgeType; //带权图中边上权值的数据类型
typedef struct{
    VertexType Vex[MaxVertexNum]; //顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵，边表
    int vexnum,arcnum; //图的当前顶点数和弧数
}MGraph;
```

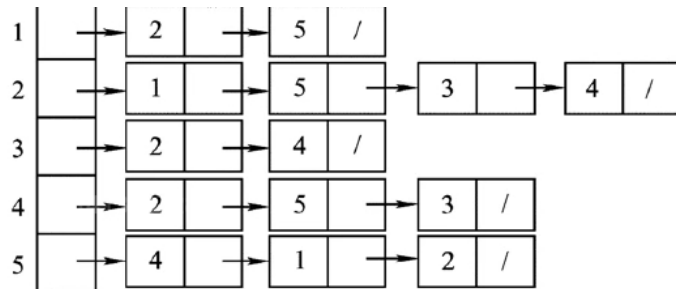
CSDN @QuantumYou

• 邻接表的定义

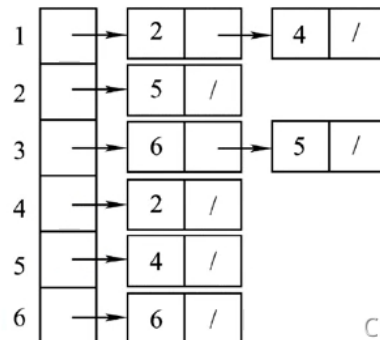
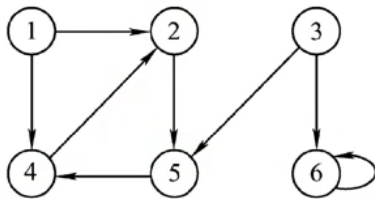
- 当一个图为稀疏图时，使用邻接矩阵法显然要浪费大量的存储空间，而图的邻接表法结合了顺序存储和链式存储方法，大大减少了这种不必要的浪费所谓邻接表，是指对图G中的每个顶点v建立一个单链表。



(a)无向图G



(b)图G的邻接表的表示



CSDN @QuantumYou

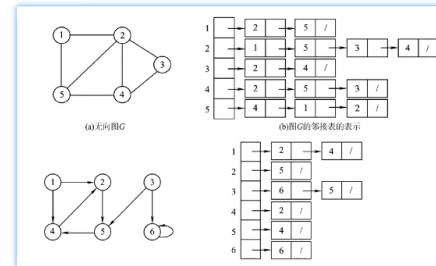
邻接表的定义

```
typedef struct _ENode // 邻接表中表对应的链表的顶点
{
    int ivex;           // 该边所指向的顶点的位置
    struct _ENode *next_edge; // 指向下一条弧的指针
}ENode, *PENode;
```

```
typedef struct _VNode // 邻接表中表的顶点
{
    char data;           // 顶点信息
    ENode *first_edge;   // 指向第一条依附该顶点的弧
}VNode;
```

```
typedef struct _LGraph // 邻接表
{
    int vexnum;           // 图的顶点的数目
    int edgnum;           // 图的边的数目
    VNode vexs[MAX];
}LGraph;
```

图的存储代码



CSDN @QuantumYou

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <string.h>
5
6  #define MAX 100
7  #define isLetter(a) (((a)>='a')&&((a)<='z')) || (((a)>='A')&&((a)<='Z'))
8  #define LENGTH(a) (sizeof(a)/sizeof(a[0]))
9
10 // 邻接表中表对应的链表的顶点
11 typedef struct _ENode
12 {
13     int ivex;           // 该边所指向的顶点的位置,是数组的下标
14     struct _ENode *next_edge; // 指向下一条弧的指针
15 }ENode, *PENode;
16
17 // 邻接表中表的顶点
18 typedef struct _VNode
19 {
20     char data;           // 顶点信息
21     ENode *first_edge;   // 指向第一条依附该顶点的弧
22 }VNode;
23
24 // 邻接表
25 typedef struct _LGraph
26 {
27     int vexnum;           // 图的顶点的数目
28     int edgnum;           // 图的边的数目
29     VNode vexs[MAX];
30 }LGraph;
31
32 /*
33  * 返回ch在matrix矩阵中的位置
34  */
35 static int get_position(LGraph g, char ch)
```

```

36 {
37     int i;
38     for(i=0; i<g.vexnum; i++)//去顶点结构体数组中遍历每个顶点
39         if(g.vexs[i].data==ch)
40             return i;//返回的是对应顶点的下标
41     return -1;
42 }
43
44 /*
45  * 读取一个输入字符
46  */
47 static char read_char()
48 {
49     char ch;
50
51     do {
52         ch = getchar();
53     } while(!isLetter(ch));
54
55     return ch;
56 }
57
58 /*
59  * 将node链接到list的末尾
60  */
61 static void link_last(ENode *list, ENode *node)
62 {
63     ENode *p = list;
64
65     while(p->next_edge)
66         p = p->next_edge;
67     p->next_edge = node;
68 }
69
70 /*
71  * 创建邻接表对应的图(自己输入)
72  */
73 LGraph* create_lgraph()
74 {
75     char c1, c2;
76     int v, e;
77     int i, p1, p2;
78     ENode *node1, *node2;
79     LGraph* pG;
80
81     // 输入"顶点数"和"边数"
82     printf("input vertex number: ");
83     scanf("%d", &v);
84     printf("input edge number: ");
85     scanf("%d", &e);
86     if ( v < 1 || e < 1 || (e > (v * (v-1))))
87     {
88         printf("input error: invalid parameters!\n");
89         return NULL;
90     }
91
92     if ((pG=(LGraph*)malloc(sizeof(LGraph))) == NULL )
93         return NULL;

```

```

94     memset(pG, 0, sizeof(LGraph));
95
96     // 初始化"顶点数"和"边数"
97     pG->vexnum = v;
98     pG->edgnum = e;
99     // 初始化"邻接表"的顶点
100    for(i=0; i<pG->vexnum; i++)
101    {
102        printf("vertex(%d): ", i);
103        pG->vexs[i].data = read_char();
104        pG->vexs[i].first_edge = NULL;
105    }
106
107    // 初始化"邻接表"的边
108    for(i=0; i<pG->edgnum; i++)
109    {
110        // 读取边的起始顶点和结束顶点
111        printf("edge(%d): ", i);
112        c1 = read_char();
113        c2 = read_char();
114
115        p1 = get_position(*pG, c1);
116        p2 = get_position(*pG, c2);
117
118        // 初始化node1
119        node1 = (ENode*)calloc(1, sizeof(ENode));
120        node1->ivex = p2;
121        // 将node1链接到"p1所在链表的末尾"
122        if(pG->vexs[p1].first_edge == NULL)
123            pG->vexs[p1].first_edge = node1;
124        else
125            link_last(pG->vexs[p1].first_edge, node1);
126        // 初始化node2
127        node2 = (ENode*)calloc(1, sizeof(ENode));
128        node2->ivex = p1;
129        // 将node2链接到"p2所在链表的末尾"
130        if(pG->vexs[p2].first_edge == NULL)
131            pG->vexs[p2].first_edge = node2;
132        else
133            link_last(pG->vexs[p2].first_edge, node2);
134    }
135
136    return pG;
137 }
138
139 /*
140  * 创建邻接表对应的图(用已提供的数据), 无向图
141  */
142 LGraph* create_example_lgraph()
143 {
144     char c1, c2;
145     char vexs[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
146     char edges[][2] = {
147         {'A', 'C'},
148         {'A', 'D'},
149         {'A', 'F'},
150         {'B', 'C'},
151         {'C', 'D'},

```

```

152     {'E', 'G'},
153     {'F', 'G'}}};
154 int vlen = LENGTH(vexs);
155 int elen = LENGTH(edges);
156 //上面类似一个邻接矩阵存储
157 int i, p1, p2;
158 ENode *node1, *node2;
159 LGraph* pG;//pG表示图
160
161
162 if ((pG=(LGraph*)malloc(sizeof(LGraph))) == NULL )
163     return NULL;
164 memset(pG, 0, sizeof(LGraph));//就是把申请的空间内初始化为零
165
166 // 初始化"顶点数"和"边数"
167 pG->vexnum = vlen;
168 pG->edgnum = elen;
169 // 初始化"邻接表"的顶点
170 for(i=0; i<pG->vexnum; i++)
171 {
172     pG->vexs[i].data = vexs[i];
173     pG->vexs[i].first_edge = NULL;
174 }
175
176 // 初始化"邻接表"的边
177 for(i=0; i<pG->edgnum; i++)
178 {
179     // 读取边的起始顶点和结束顶点
180     c1 = edges[i][0];
181     c2 = edges[i][1];
182
183     p1 = get_position(*pG, c1);//p1对应起始顶点下标位置
184     p2 = get_position(*pG, c2);//p1对应结束顶点下标位置
185
186     // 初始化node1
187     node1 = (ENode*)calloc(1, sizeof(ENode));
188     node1->ivex = p2;
189     // 将node1链接到"p1所在链表的末尾"
190     if(pG->vexs[p1].first_edge == NULL)
191         pG->vexs[p1].first_edge = node1;
192     else
193         link_last(pG->vexs[p1].first_edge, node1);
194     // 初始化node2
195     node2 = (ENode*)calloc(1, sizeof(ENode));
196     node2->ivex = p1;
197     // 将node2链接到"p2所在链表的末尾"
198     if(pG->vexs[p2].first_edge == NULL)
199         pG->vexs[p2].first_edge = node2;
200     else
201         link_last(pG->vexs[p2].first_edge, node2);
202 }
203
204 return pG;
205 }
206
207 /*
208  * 深度优先搜索遍历图的递归实现
209  */

```

```

210 static void DFS(LGraph G, int i, int *visited)
211 {
212     ENode *node;
213
214     visited[i] = 1; //要访问当前结点了，所以打印
215     printf("%c ", G.vexs[i].data);
216     node = G.vexs[i].first_edge; //拿当前顶点的后面一个顶点
217     while (node != NULL)
218     {
219         if (!visited[node->ivex]) //只要对应顶点没有访问过，深入到下一个顶点访问
220             DFS(G, node->ivex, visited);
221         node = node->next_edge; //某个顶点的下一条边，例如B结点的下一条边
222     }
223 }
224
225 /*
226  * 深度优先搜索遍历图
227  */
228 void DFSTraverse(LGraph G)
229 {
230     int i;
231     int visited[MAX]; // 顶点访问标记
232
233     // 初始化所有顶点都没有被访问
234     for (i = 0; i < G.vexnum; i++)
235         visited[i] = 0;
236
237     printf("DFS: ");
238     //从A开始深度优先遍历
239     for (i = 0; i < G.vexnum; i++)
240     {
241         if (!visited[i])
242             DFS(G, i, visited);
243     }
244     printf("\n");
245 }
246
247 /*
248  * 广度优先搜索（类似于树的层次遍历）
249  */
250 void BFS(LGraph G)
251 {
252     int head = 0;
253     int rear = 0;
254     int queue[MAX]; // 辅助队列
255     int visited[MAX]; // 顶点访问标记
256     int i, j, k;
257     ENode *node;
258
259     //每个顶点未被访问
260     for (i = 0; i < G.vexnum; i++)
261         visited[i] = 0;
262     //从零号顶点开始遍历
263     printf("BFS: ");
264     for (i = 0; i < G.vexnum; i++) //对每个连通分量均调用一次BFS
265     {
266         if (!visited[i]) //如果没访问过，就打印，同时入队，最初是A
267         {

```



```

268         visited[i] = 1; //标记已经访问过
269         printf("%c ", G.vexs[i].data);
270         queue[rear++] = i; // 入队列
271     }
272     while (head != rear) //第一个进来的是A，遍历A的每一条边
273     {
274         j = queue[head++]; // 出队列
275         node = G.vexs[j].first_edge;
276         while (node != NULL)
277         {
278             k = node->ivex;
279             if (!visited[k])
280             {
281                 visited[k] = 1;
282                 printf("%c ", G.vexs[k].data);
283                 queue[rear++] = k; //类似于树的层次遍历，遍历到的同时入队
284             }
285             node = node->next_edge;
286         }
287     }
288 }
289 printf("\n");
290 }
291
292 /*
293  * 打印邻接表图
294  */
295 void print_lgraph(LGraph G)
296 {
297     int i;
298     ENode *node;
299
300     printf("List Graph:\n");
301     for (i = 0; i < G.vexnum; i++) //遍历所有的顶点
302     {
303         printf("%d(%c): ", i, G.vexs[i].data);
304         node = G.vexs[i].first_edge;
305         while (node != NULL) //把每个顶点周围的结点都输出一下
306         {
307             printf("%d(%c) ", node->ivex, G.vexs[node->ivex].data);
308             node = node->next_edge;
309         }
310         printf("\n");
311     }
312 }
313
314 /*
315  * 创建邻接表对应的图(有向图)
316  */
317 LGraph* create_example_lgraph_directed()
318 {
319     char c1, c2;
320     char vexs[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
321     char edges[][2] = {
322         {'A', 'B'},
323         {'B', 'C'},
324         {'B', 'E'},
325         {'B', 'F'},
326         {'C', 'E'},

```

```

326         {'D', 'C'},
327         {'E', 'B'},
328         {'E', 'D'},
329         {'F', 'G'}};
330     int vlen = LENGTH(vexs);
331     int elen = LENGTH(edges);
332     int i, p1, p2;
333     ENode *node1;
334     LGraph* pG;
335
336
337     if ((pG=(LGraph*)malloc(sizeof(LGraph))) == NULL )
338         return NULL;
339     memset(pG, 0, sizeof(LGraph));
340
341     // 初始化"顶点数"和"边数"
342     pG->vexnum = vlen;
343     pG->edgnum = elen;
344     // 初始化"邻接表"的顶点
345     for(i=0; i<pG->vexnum; i++)
346     {
347         pG->vexs[i].data = vexs[i];
348         pG->vexs[i].first_edge = NULL;
349     }
350
351     // 初始化"邻接表"的边
352     for(i=0; i<pG->edgnum; i++)
353     {
354         // 读取边的起始顶点和结束顶点
355         c1 = edges[i][0];
356         c2 = edges[i][1];
357
358         p1 = get_position(*pG, c1);
359         p2 = get_position(*pG, c2);
360         // 初始化node1
361         node1 = (ENode*)calloc(1, sizeof(ENode));
362         node1->ivex = p2;
363         // 将node1链接到"p1所在链表的末尾"
364         if(pG->vexs[p1].first_edge == NULL)
365             pG->vexs[p1].first_edge = node1;
366         else
367             link_last(pG->vexs[p1].first_edge, node1);
368     }
369
370     return pG;
371 }
372 //图的创建, 打印, 广度优先遍历, 深度优先遍历
373 //有向图
374 void main()
375 {
376     LGraph* pG;
377
378     // 无向图自定义"图"(自己输入数据, 输入的方法可以参考create_example_lgraph初始化的数据)
379     //pG = create_lgraph();
380     //无向图的创建, 采用已有的"图"
381     //pG = create_example_lgraph();
382     //有向图的创建

```

```
383     pG = create_example_lgraph_directed();
384     // 打印图
385     print_lgraph(*pG);
386     DFSTraverse(*pG); //深度优先遍历
387     BFS(*pG); //广度优先遍历
388     system("pause");
389 }
```