

第四章 串

•定义：串，即字符串，是由0个或多个字符组成的有限序列

字符串：串中任意个连续的字符组成的子序列

主串：包含子串的串

字符在主串中的位置：字符在串中的序号

子串在主串中的位置：子串的第一个字符在主串中的位置

空串（长度为0） 空指针

•基本操作

- 赋值操作 StrAssign (&T, chars) ；
- 复制操作 StrCopy (&T, S) ；
- 判空操作 StrEmpty (S) ；
- 求串长 StrLength (S) ；
- 清空操作 Clear-String (&S) ；
- 销毁串 DestroyString(&S);
- 连接串 (&T, S1, S2) ；
- 定位操作 Index (S, T) ；
- 比较操作 StrCompare (S, T) ；

•串的结构

1. 串的顺序存储

```
1 // 静态数组（定长顺序）实现
2 // 静态数组（定长顺序）实现
3 #define MAXLEN 255
4 typedef char sstring[255];
5 // 每个分量存储一个字符串
6 int length; // 串的实际长度
7 // 字符串
8 // 动态数组（变分配）实现
9 typedef struct {
10     char *ch; // 按照串长分配存储区，ch指向串的基地址
11     int length; // 串的长度
12 } sstring;
13 // 字符串
14 // 字符串
15 // 字符串
16 // 字符串
17 // 字符串
18 // 字符串
```

2. 串的链式存储

```
1 // 链式存储
2 // 链式存储
3 typedef struct StringNode {
4     char ch;
5     struct StringNode *next;
6 } StringNode;
7 // 链式存储
8 // 链式存储
9 typedef struct StringNode {
10     char ch;
11     struct StringNode *next;
12 } StringNode;
13 // 链式存储
14 // 链式存储
15 // 链式存储
16 // 链式存储
```

•基本操作实现

1. 求子串：用Sub返回串S的第pos个字符起长度为len的子串

```
1 // 求子串
2 #define MAXLEN 255
3 typedef struct {
4     char ch[MAXLEN]; // 每个分量存储一个字符串
5     int length; // 串的实际长度
6 } SString;
7 bool SubString(SString &sub, SString S, int pos, int len) {
8     // 子串是否合法
9     if (pos < 1 || pos > S.length || len < 1 || len > S.length - pos + 1)
10         return false;
11     for (int i = pos; i <= pos + len; i++)
12         sub.ch[i - pos] = S.ch[i];
13     sub.length = len;
14     return true;
15 }
```

2. 比较操作：比较两个串，若s1>t，返回大于0，若s1=t，返回等于0，否则小于0

```
1 // 比较操作
2 int StrCompare(SString S, SString T) {
3     // 比较两个串，若s1>t，返回大于0，若s1=t，返回等于0，否则小于0
4     for (int i = 0; i < S.length && i < T.length; i++)
5         if (S.ch[i] != T.ch[i])
6             return S.ch[i] - T.ch[i];
7     // 如果两个串的所有字符都一致，则长度大的串更大
8     return S.length - T.length;
9 }
```

3. 定位操作：若主串S中存在与串T值相等的子串，则返回它在主串中第一次出现的位置，否则返回0

```
1 // 定位操作
2 int Index(SString S, SString T) {
3     int i = 1; // 从主串S的第一个字符起，依次与子串T的第一个字符比较
4     while (i <= S.length) {
5         if (StrCompare(S.ch + i, T.ch) == 0)
6             return i; // 返回子串T在主串S中的位置
7         i++;
8     }
9     return 0;
10 }
```

•字符串的模式匹配：在主串中找到与模式串相同的子串，并返回其所在位置

1. 朴素模式匹配算法（暴力）：将主串中所有长度为m的子串（最多对len-m+1个子串）依次与模式串对比，直到找到一个完全匹配的子串，或者所有子串都不匹配为止

```
1 // 朴素模式匹配算法（暴力）
2 int Index_Naive(SString s, SString t) {
3     int i = 1;
4     while (i <= s.length - t.length + 1) {
5         if (StrCompare(s.ch + i, t.ch) == 0)
6             return i;
7         i++;
8     }
9     return 0;
10 }
```

2. KMP算法

注意：KMP算法

```
1 // KMP算法
2 int Index_KMP(SString S, SString T, int next[]) {
3     int i = 1;
4     while (i <= S.length) {
5         if (StrCompare(S.ch + i, T.ch) == 0)
6             return i;
7         i++;
8     }
9     return 0;
10 }
```

第五章 树和二叉树

•基本概念

- 结点——树中任意一个结点，由数据域和指针域组成。数据域存放数据，指针域存放指向其他结点的指针。
- 树根：每个结点的父结点的父结点。
- 非空树：至少有一个结点的树。
- 叶子结点：没有后继的结点称为“叶子结点”。
- 分支结点：有后继的结点称为“分支结点”。
- 度：一个结点的子结点的个数。
- 每个结点可以有0个或多个后继。

树的相关概念

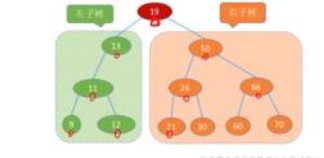
- 根结点：树中只有一个结点的树。
- 叶子结点：度为0的结点。
- 内部结点：度大于0的结点。
- 度为m的结点：具有m个子结点的结点。
- 度为m的树：所有结点的度都不大于m的树。

•二叉树（有序树）的基本概念

- 二叉树是n（n>0）个结点的有限集合：
- 1. 或者为空树，即n=0。
- 2. 或者由一个根结点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。左右子树分别也是一棵二叉树。
- 3. 特点：a. 每个结点最多有两棵子树，左右子树不能颠倒。

几个特殊的二叉树

- 1. 满二叉树：一棵高度为h，且含有2^h-1个结点的二叉树。
 - a. 只有最后一层又叶子结点。
 - b. 不存在度为1的结点。
- 2. 完全二叉树：按层次从上到下，除最后一层外，每一层上的所有结点都只有左孩子或右孩子。
 - a. 只有最后一层的叶子结点。
 - b. 最多只有一个度为1的结点。
 - c. 按层次从上到下，除最后一层外，每一层上的所有结点都只有左孩子或右孩子。
 - d. 如果最后一层的叶子结点是左孩子，则其父结点是左孩子；如果是右孩子，则其父结点是右孩子。
- 3. 二叉排序树：一棵二叉树或者是一棵空树，或者是具有如下性质的树：
 - a. 左子树的所有结点的关键字都小于根结点的关键字。
 - b. 右子树的所有结点的关键字都大于根结点的关键字。
 - c. 左子树和右子树又分别是一棵二叉排序树。



4. 平衡二叉树：树上任一点的**左子树和右子树的高度之差**不超过1

二叉树常用性质：

1. 设非空二叉树中度为0, 1和2的结点个数分别为 n_0 , n_1 和 n_2 ，则 $n_0=n_2+1$
2. 二叉树第 i 层之多有 2^{i-1} 个结点， m 叉树第 i 层最多有 m^{i-1} 个结点
3. 高度为 n 个二叉树至多有 2^n-1 个结点满二叉树高度为 n 时 m 叉树至多有 $(m^n-h-1)/(m-1)$ 个结点

二叉树的存储结构

```
1 //顺序存储
2
3 #define MaxSize 100
4 struct TreeNode{
5     ElemType value; // 结点中的数据元素
6     bool isEmpty; // 结点是否为空
7 };
8
9 TreeNode t[MaxSize];
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

二叉树的先/中/后/层序遍历

```
1 typedef struct BitNode{
2     ElemType value;
3     struct BitNode *lchild,*rchild;
4 }BitNode,*BitTree;
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

• 求树的深度

```
1 //求树的深度
2
3 int treeDepth(BitTree T){
4     if(T==NULL){
5         return 0;
6     }
7     else{
8         int l = treeDepth(T->lchild);
9         int r = treeDepth(T->rchild);
10        // 树的高度=左子树高度, 右子树高度 +1;
11        return 1+max(l,r);
12    }
13 }
```

• 层序遍历

```
1 //顺序遍历
2
3 //二叉树的结点
4 typedef struct BitNode{
5     char data;
6     struct BitNode *lchild,*rchild;
7 }BitNode,*BitTree;
8
9 //链式双链表结点
10 typedef struct LinkNode{
11     BitNode *data;
12     struct LinkNode *next;
13 }LinkNode;
14
15 typedef struct{
16     LinkNode *front,*rear; // 队头队尾
17 }LinkQueue;
18
19 //初始化队列操作 (带头结点)
20 void InitQueue(LinkQueue *Q){
21     Q->front = Q->rear = (LinkNode*)malloc(sizeof(LinkNode));
22     Q->front->next=NULL;
23 }
24
25 //判断队列是否为空 (带头结点)
26 bool IsEmpty(LinkQueue *Q){
27     if(Q->front == Q->rear){
28         return true;
29     }
30     else{
31         return false;
32     }
33 }
34
35 //入队 (带头结点)
36 void EnQueue(LinkQueue *Q,ElemType x){
37     LinkNode *p=(LinkNode*)malloc(sizeof(LinkNode));
38     p->data = x;
39     p->next = NULL;
40     Q->rear->next = p; // 新结点插入到...之后
41     Q->rear = p; // 修改队尾指针
42 }
43
44 void LevelOrder(BitTree T){
45     LinkQueue Q; // 初始化链队列
46     InitQueue(Q);
47     BitTree p;
48     EnQueue(Q,T); // 将根结点入队
49     while(!IsEmpty(Q)){ // 队列不为空时循环
50         DeQueue(Q,p); // 队头结点出队
51         visit(p); // 访问队头结点
52         if(p->lchild!=NULL){
53             EnQueue(Q,p->lchild); // 左孩子入队
54         }
55         if(p->rchild!=NULL){
56             EnQueue(Q,p->rchild); // 右孩子入队
57         }
58     }
59 }
```

• 线索二叉树

1. 线索二叉树的存储结构

```
1 //二叉树的结点 (链式存储)
2 typedef struct BitNode{
3     ElemType data;
4     struct BitNode *lchild,*rchild;
5 }
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
```

```
3. 中序线索化
// 中序线索化
// 全局变量pre，指向当前访问结点的前驱
ThreadNode *pre = NULL;

// 线索二叉树结点
typedef struct ThreadNode{
    ElemType data;
    struct ThreadNode *lchild,*rchild;
    int ltag,rtag;//左右线索标志
}ThreadNode,*ThreadTree;

// 中序遍历二叉树，一边遍历一边线索化
void InThread(ThreadTree T){
    if(T!=NULL){
        InThread(T->lchild);//中序遍历左子树
        visit(T);//访问结点
        InThread(T->rchild);//中序遍历右子树
    }
}

void visit(ThreadNode *q){
    if(q->lchild==NULL){//左子树为空，建立前驱线索
        q->lchild=pre;
        q->ltag = 1;
    }
    if(pre!=NULL&&pre->rchild==NULL){
        pre->rchild=q;//建立前驱结点的后继线索
        pre->rtag = 1;
    }
    pre = q;
}

// 中序线索化二叉树
void CreateInThread(ThreadTree T){
    pre = NULL;
    if(T!=NULL){//非空二叉树才能线索化
        InThread(T);
        if(pre->rchild==NULL)
            pre->rtag=1;//处理遍历后的最后一个结点
    }
}
```

```
4. 先序遍历线索化
// 先序线索化
// 全局变量pre，指向当前访问结点的前驱
ThreadNode *pre = NULL;

// 线索二叉树结点
typedef struct ThreadNode{
    ElemType data;
    struct ThreadNode *lchild,*rchild;
    int ltag,rtag;//左右线索标志
}ThreadNode,*ThreadTree;

// 先序遍历二叉树，一边遍历一边线索化
void PreThread(ThreadTree T){
    if(T!=NULL){
        visit(T);//访问结点
        if(T->ltag==0){//lchild不是空结点
            PreThread(T->lchild);//先序遍历左子树
        }
        PreThread(T->rchild);//先序遍历右子树
    }
}

void visit(ThreadNode *q){
    if(q->lchild==NULL){//左子树为空，建立前驱线索
        q->lchild=pre;
        q->ltag = 1;
    }
    if(pre!=NULL&&pre->rchild==NULL){
        pre->rchild=q;//建立前驱结点的后继线索
        pre->rtag = 1;
    }
    pre = q;
}

// 先序线索化二叉树
void CreatePreThread(ThreadTree T){
    pre = NULL;
    if(T!=NULL){//非空二叉树才能线索化
        PreThread(T);
        if(pre->rchild==NULL)
            pre->rtag=1;//处理遍历后的最后一个结点
    }
}
```

```
5. 后序线索化
// 后序线索化
void PostThread(ThreadTree p,ThreadTree &pre){
    if(p!=NULL){
        PostThread(p->rchild,pre);//递归，线索化右子树
        PostThread(p->lchild,pre);//递归，线索化左子树
        if(p->lchild==NULL){//左子树为空，建立前驱线索
            p->lchild=pre;
            p->ltag=1;
        }
        if(pre!=NULL&&pre->rchild==NULL){
            pre->rchild=p;
            pre->rtag=1;
        }
        pre=p;
    }
}

void CreatePostThread(ThreadTree T){
    ThreadTree pre = NULL;
    if(T!=NULL){
        PostThread(T,pre);//线索化二叉树
        if(pre->rchild==NULL)//处理遍历后的最后一个结点
            pre->rtag=1;
    }
}
```


第五章 树和二叉树

•基本概念



空树——结点数为0的树

非空树

- 有且仅有一个根节点；c
- 没有后继的结点称为“叶子节点”（或终端结点）；
- 有后继的结点称为“分支节点”（或非终端结点）；
- 除了根节点外，任何一个结点都有且仅有一个前驱
- 每个结点可以有0个或多个后继

•基本术语