

7-树与二叉树

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

1、目标



掌握树的基本概念

掌握二叉树的定义

掌握二叉树的创建

掌握特殊二叉树和性质

掌握二叉树的递归非递归遍历

掌握二叉树的恢复

堆

线索化二叉树

树与二叉树的转换----左孩子右兄弟法

huffman编码，压缩解压缩

2、基本概念



树是一个递归概念（有且仅有一个根节点，其余节点互不相交，每个集合本身又是一棵树）

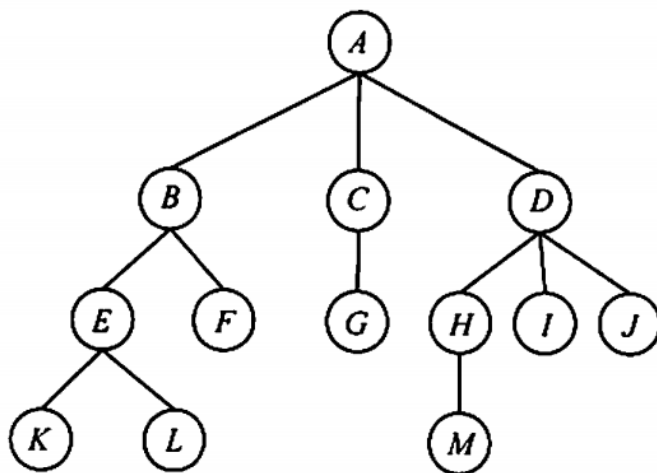
节点的度、树的度

根节点、叶子节点、终端节点

树的高度、层次

有序数、无序树

父节点、子节点



3、树的性质

 首先注意，这是树的性质，是普通树形结构，不是二叉树的性质

树具有如下最基本的性质：

- 1) 树中的结点数等于所有结点的度数之和加 1。
- 2) 度为 m 的树中第 i 层上至多有 m^{i-1} 个结点 ($i \geq 1$)。
- 3) 高度为 h 的 m 叉树至多有 $(m^h - 1)/(m - 1)$ 个结点^①。
- 4) 具有 n 个结点的 m 叉树的最小高度为 $\lceil \log_m(n(m - 1) + 1) \rceil$ 。

4、二叉树

4.1 二叉树的形态

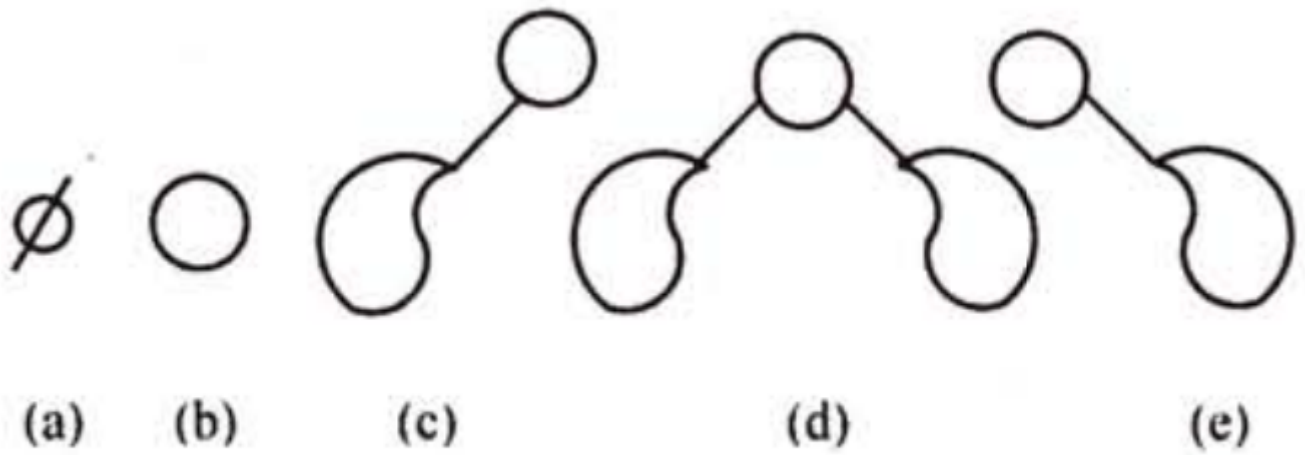
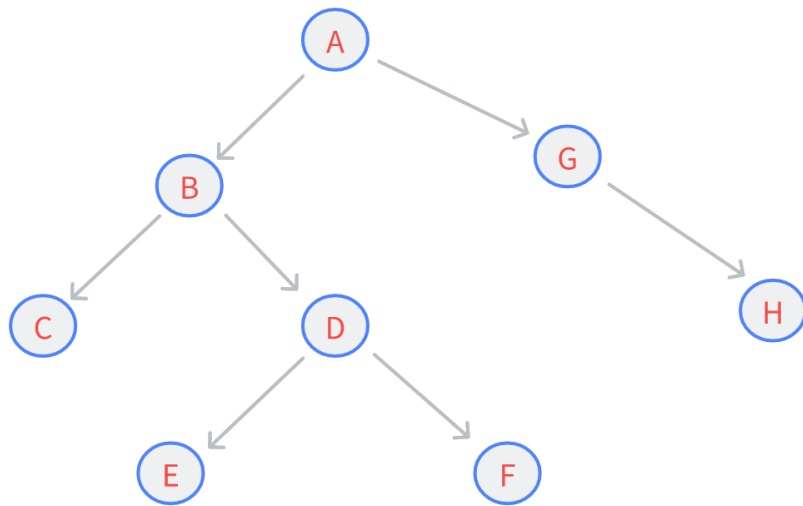


图 6.3 二叉树的 5 种基本形态

4.2 二叉树的遍历

📌 先序遍历，中序遍历，后序遍历，层次遍历

无论何种遍历，永远记住，一定是先遍历左子树，再遍历右子树，在根据根节点的位置来划分先中后。



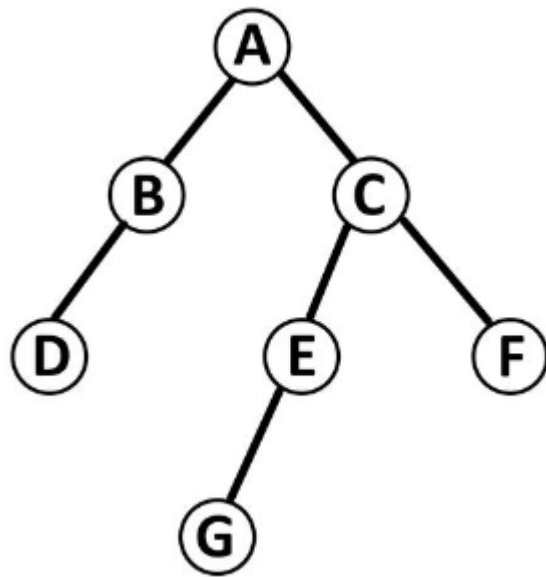
📌 VLR: ABCDEFGH

LVR: CBEDFAGH

LRV: CEFDBHGA

LEVEL: ABGCDHEF

• 练习



VLR: A B D C E G F
 LVR: D B A G E C F
 LRV: D B G E F C A
 LEVEL: A B C D E F G

非递归遍历，前中后序需要借助栈结构，层次遍历需要借助队列结构

4.3 二叉树ADT

```

1 #define BinTreeElem_Type char
2
3 //定义二叉树的节点类型
4 typedef struct BinTreeNode
5 {
6     BinTreeElem_Type data;
7     struct BinTreeNode *leftChild;
8     struct BinTreeNode *rightChild;
9 }BinTreeNode;
10
11 typedef BinTreeNode* BinTree;
12
13 //函数申明
14 void BinTreeInit(BinTree *t);
15 void BinTreeCreate_1(BinTree *t);
16 BinTree BinTreeCreate_2();
17 BinTree BinTreeCreate_3(const char *str, int *idx);
18 BinTree BinTreeCreate_4(const char *vlr, const char *lvr, int n);
19 BinTree BinTreeCreate_5(const char *lvr, const char *lr, int n);
  
```

```

20
21 //递归遍历
22 void PreOrder(BinTree t);
23 void InOrder(BinTree t);
24 void PostOrder(BinTree t);
25
26 //非递归遍历
27 void PreOrder_NoR(BinTree t);
28 void InOrder_NoR(BinTree t);
29 void PostOrder_NoR(BinTree t);
30 void LevelOrder(BinTree t);
31
32 //二叉树操作
33 size_t Size(BinTree t);
34 size_t Height(BinTree t);
35 BinTreeNode* Find(BinTree t, BinTreeElem_Type key);
36 BinTreeNode* Parent(BinTree t, BinTreeNode *p);
37 size_t LeafSize(BinTree t);
38 size_t LevelKSize(BinTree t, int k);
39 bool Equal(BinTree t1, BinTree t2);
40 BinTree Copy(BinTree t);
41 void BinTreeDestroy(BinTree *t);

```

二叉树必会题型：

- 1、前序遍历 <https://leetcode.cn/problems/binary-tree-preorder-traversal/>
- 2、中序遍历 <https://leetcode.cn/problems/binary-tree-inorder-traversal/>
- 3、后序遍历 <https://leetcode.cn/problems/binary-tree-postorder-traversal/>
- 4、二叉树最大深度 <https://leetcode.cn/problems/maximum-depth-of-binary-tree/>
- 5、相同二叉树 <https://leetcode.cn/problems/same-tree/>
- 6、对称二叉树 <https://leetcode.cn/problems/symmetric-tree/description/>
- 7、单值二叉树 <https://leetcode.cn/problems/univalued-binary-tree/description/>
- 8、翻转二叉树 <https://leetcode.cn/problems/invert-binary-tree/description/>
- 9、二叉树最近公共祖先 <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/description/>

4.4 二叉树的恢复

这是一种最常见的考题

前中后序，可以根据前序中序恢复，也可以根据中序后序恢复，但不能仅依靠前序和后序恢复

比如：前序: ABCDEFGH, 中序: CBEDFAGH

比如：中序: CBEDFAGH, 后续: CEFDBHGA



1、前序+中序构造二叉树 <https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>

2、中序+后续构造二叉树 <https://leetcode.cn/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>

4.5 特殊二叉树

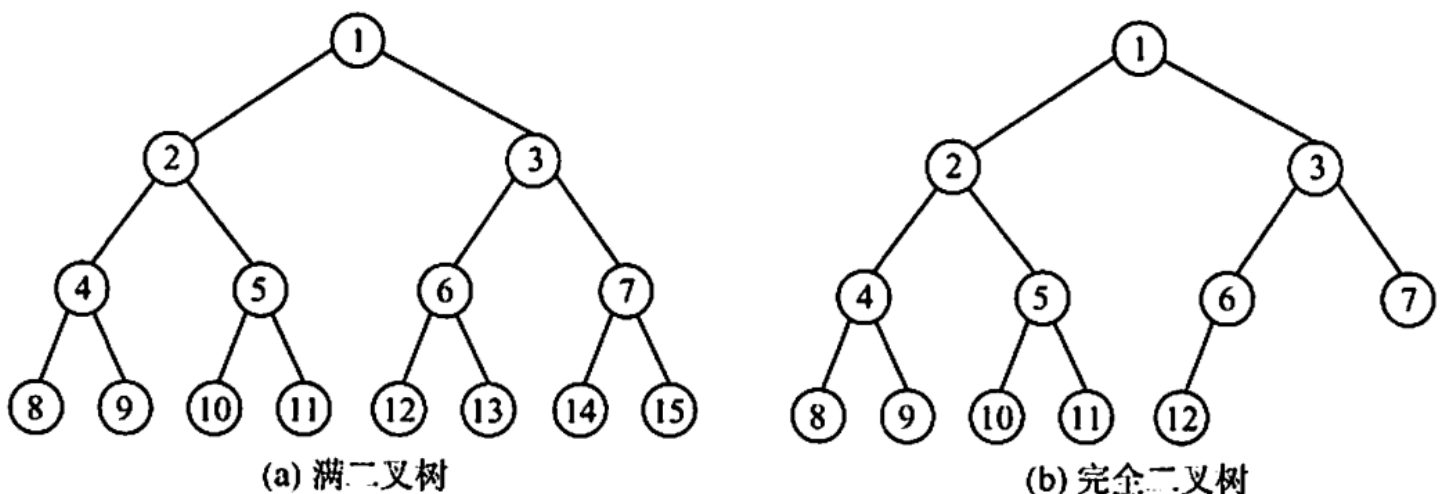


图 5.3 两种特殊形态的二叉树^①

4.6 二叉树性质



1、在二叉树的第 i 层上至多有 $2^{(i-1)}$ 个结点， $i \geq 1$

2、深度为 k 的二叉树至多有 $2^k - 1$ 个结点， $k \geq 1$

3、任何一棵二叉树 T ,如果其终端结点数为 n_0 ,度为2的结点数为 n_2 ,则 $n_0 = n_2 + 1$

4、具有 n 个结点的完全二叉树的深度为 $\log_2(n)$ 向下取整+1

5、如果对一棵有 n 个结点的完全二叉树，可以通过父节点求子女结点，也可以通过子女结点求父节点，不过要注意，根节点的编号是从0开始还是从1开始

5、线索化二叉树



二叉树空指针的个数：

在有 n 个节点的二叉树中，空指针的个数为 $n+1$ 个。

推导:叶子节点有两个空指针，度为1的节点有一个空指针，假设叶子节点数为 n_0 ,度为1的节点数为 n_1 ,则空指针数为 $2*n_0 + n_1$, 又因为 $n_0=n_2+1$, 所以 $2*n_0 + n_1 = n_0+n_1+n_2+1 = n+1$

线索二叉树概述：

线索二叉树是一种**光盘行动**，合理利用空指针，杜绝浪费，结点没有左右子树时，就让左右指针保存其他有用信息，左指针保存前驱节点，右指针保存后继节点，此时左右指针不再是单纯的左右孩子结点指针，而是相对应遍历顺序的前驱和后继节点的链接指针，相当于前驱和后继的线索信息，顾因此得名线索化的二叉树

线索二叉树作用：

利用空指针来存放前驱和后继指针后，就可以像遍历链表那样方便的遍历二叉树，加速了二叉树的前驱结点和后继结点的访问



线索二叉树的分类：

前序线索二叉树、中序线索二叉树、后续线索二叉树



图形说明：

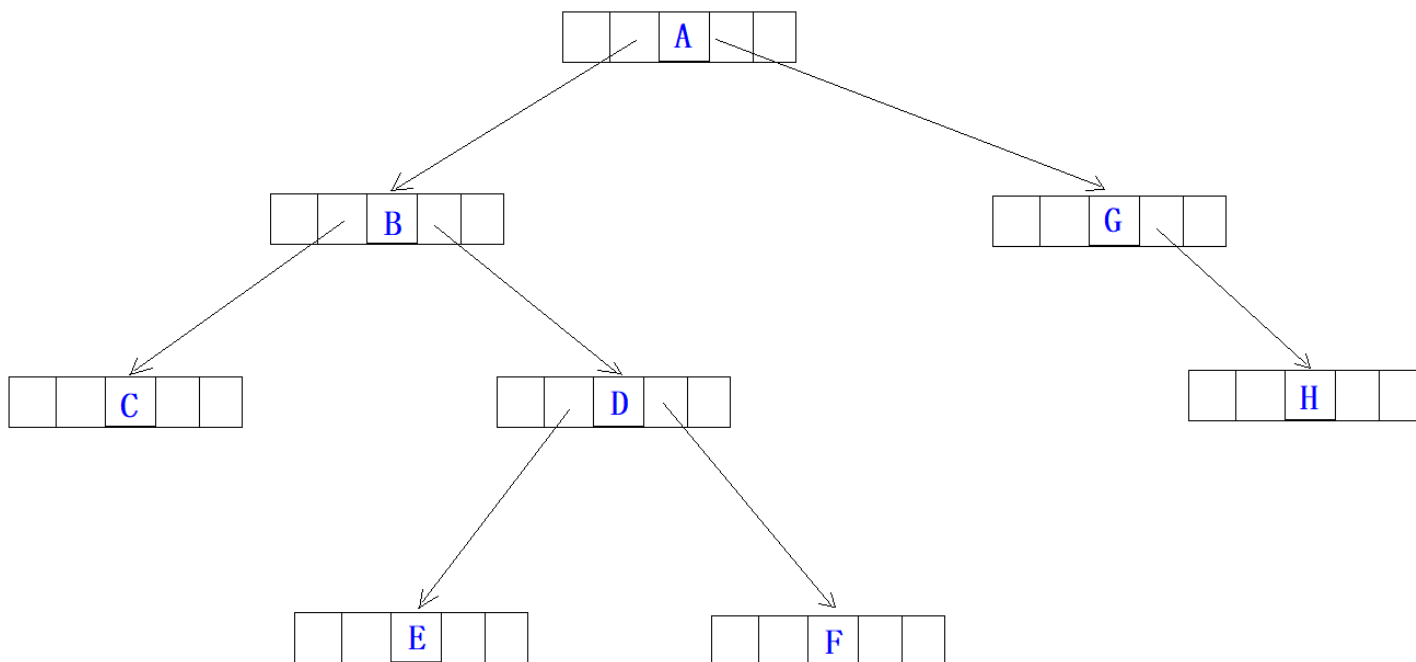
0代表 链

1代表 线

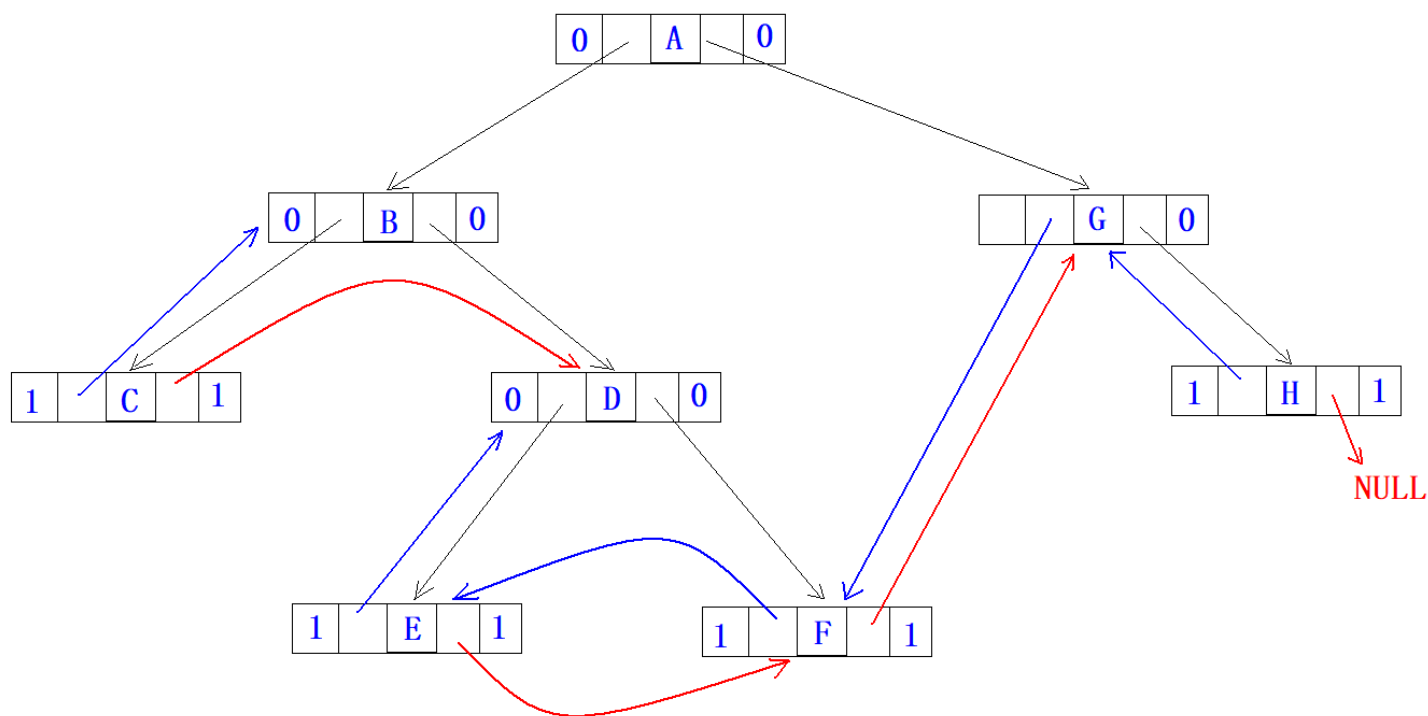
蓝色代表 前驱线索

红色代表 后继线索

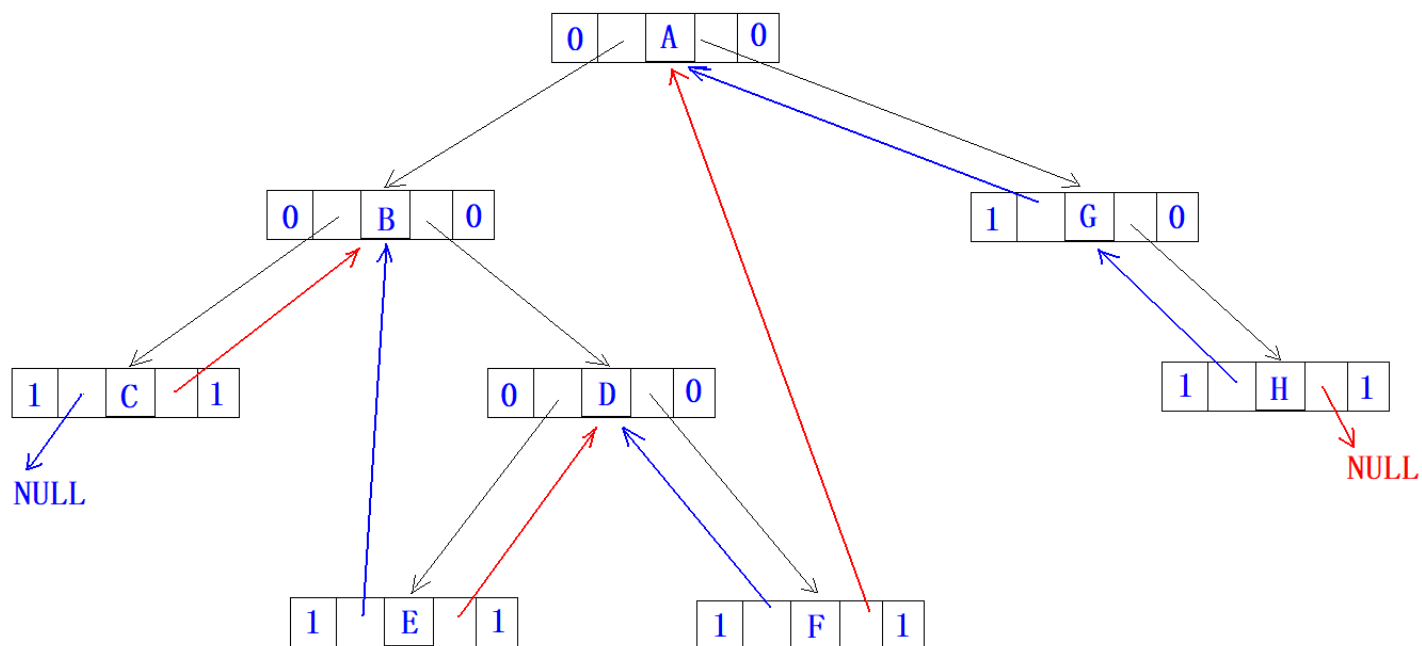
黑色代表 正常子女指针



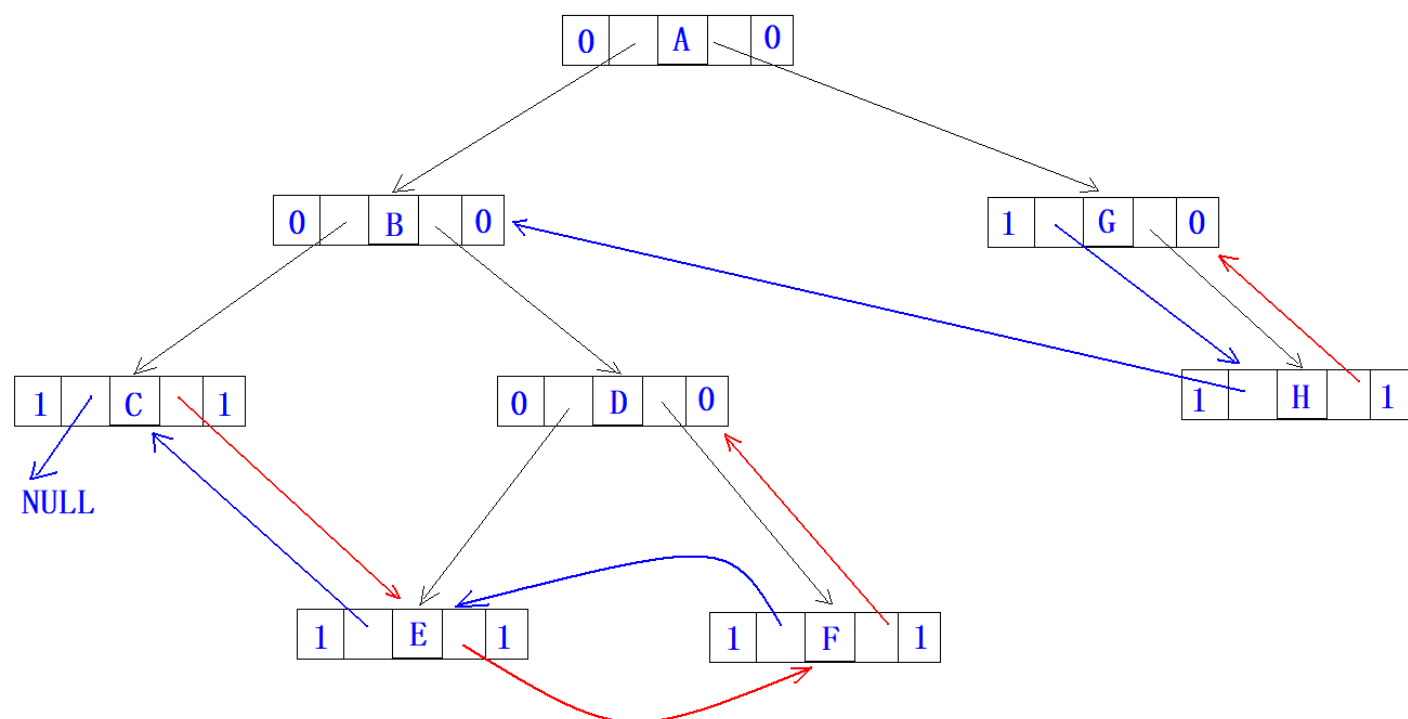
- 前序线索二叉树



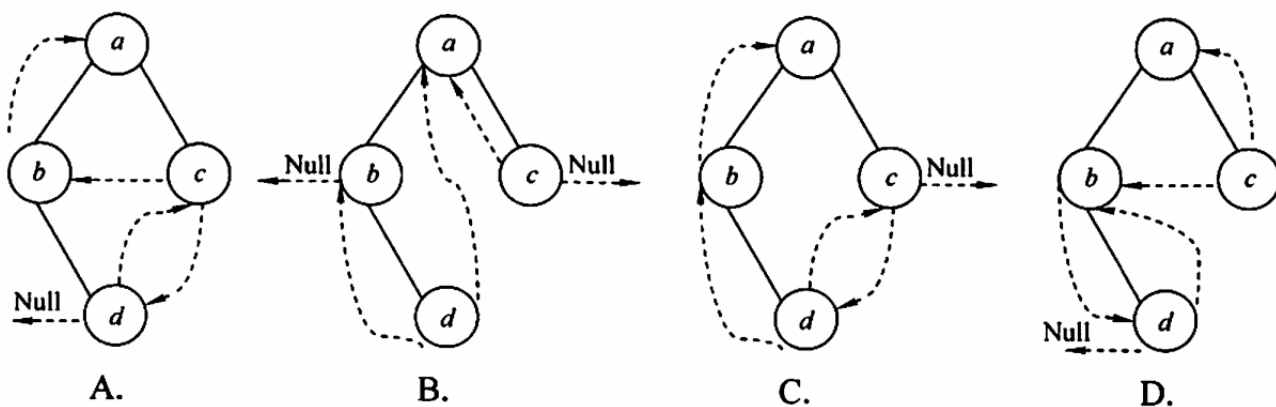
- 中序线索二叉树



• 后序线索二叉树



28. 【2010 统考真题】下列线索二叉树中（用虚线表示线索），符合后序线索树定义的是（ ）。



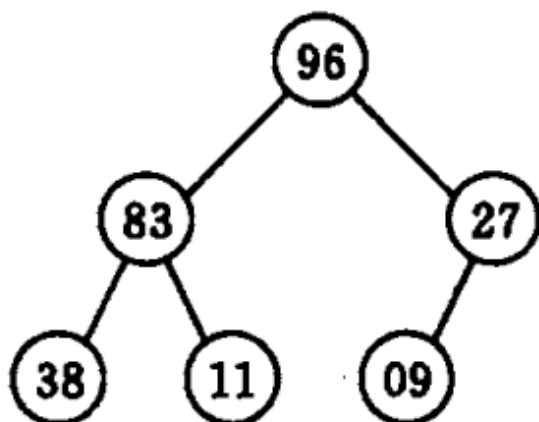
6、堆

📌 堆的概述：

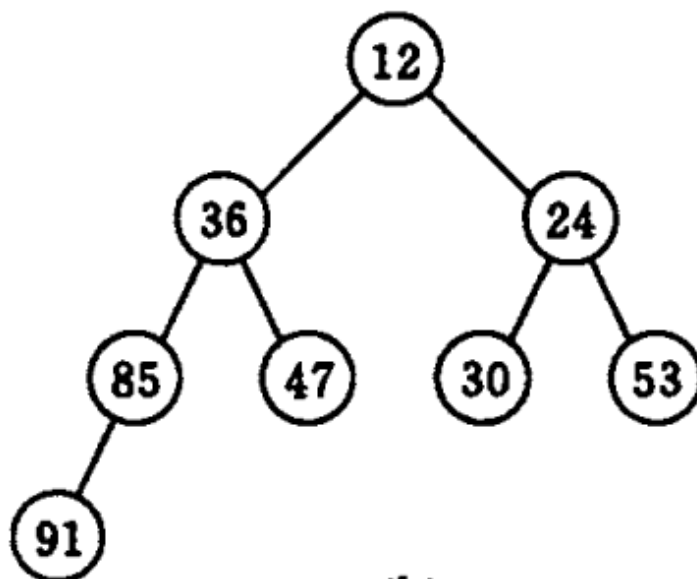
堆是一棵完全二叉树，采用数组顺序形式存储，有大小堆之分

堆又叫做优先级队列

优先级队列是完全二叉树 + 堆的规则（大小堆）

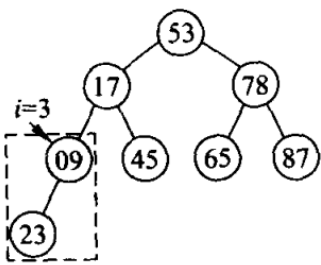


(a)

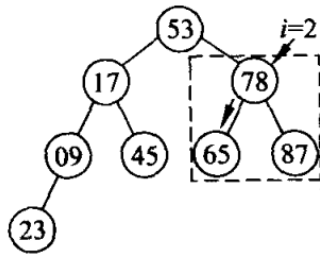


(b)

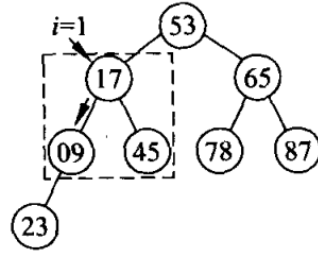
- 整体数据建堆



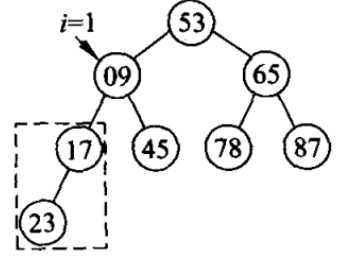
(a) 初始 $i=3$



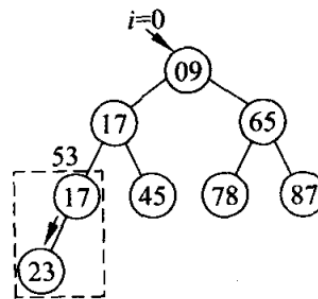
(b) $i=2$



(c) $i=1$

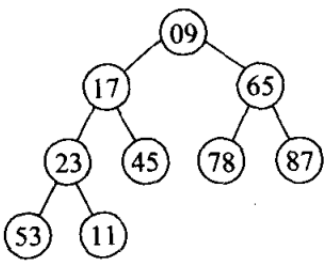


(d) $i=0$

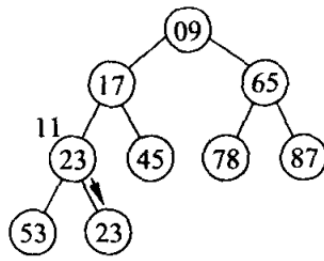


(e) 结果

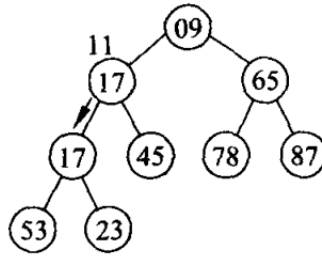
• 插入结点



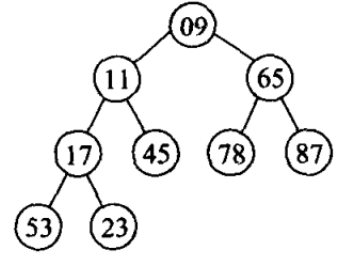
(a) 初始尾部加 11



(b) 父结点关键词 23 下降

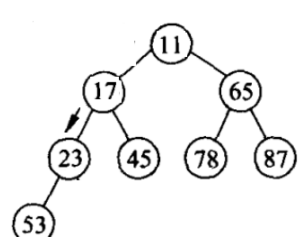
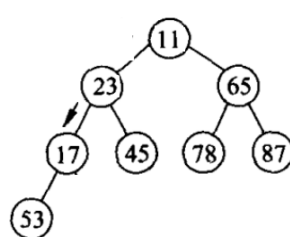
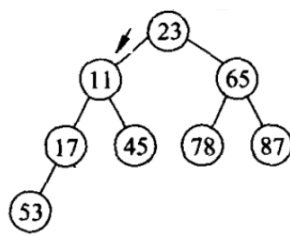
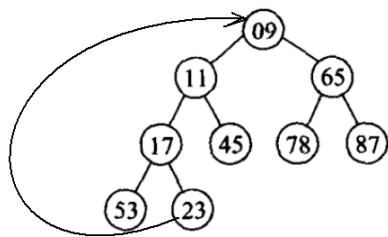



(c) 父结点关键词 17 下降



(d) 11 回填调整完成

• 删除节点



 插入数据：向上调整

删除数据：向下调整

• 堆的ADT

```
1 #define HEAP_DEFAULT_SIZE 10
```


```
2
```

```

3 #define HeapElem_Type int
4 typedef struct Heap
5 {
6     HeapElem_Type *heap;
7     size_t        capacity;
8     size_t        size;
9 }Heap;
10
11 void HeapInit(Heap *php);
12 void HeapCreate(Heap *php, HeapElem_Type ar[], int n);
13 void HeapInsert(Heap *php, HeapElem_Type v);
14 void HeapErase(Heap *php);
15 HeapElem_Type HeapTop(Heap *php);
16 size_t HeapSize(Heap *php);
17 void HeapShow(Heap *php);
18 void HeapSort(Heap *php, HeapElem_Type ar[], int n);
19 void HeapDestory(Heap* php);

```


• 堆排序

 堆排序的过程是每次将堆顶元素跟堆的最后一个元素交换（不是存储空间的最后一个元素，而是堆结构中的有效结点），再对堆进行一次向下调整，反复循环，直到堆的所有元素排序完成。

升序排序： 建立大堆

降序排序： 建立小堆

• topk问题

 topk问题是一个求前k个最值问题，可以借助堆结构进行筛选

比如，现有100000个不重复的数，求出前10个最大值或最小值

具体做法：

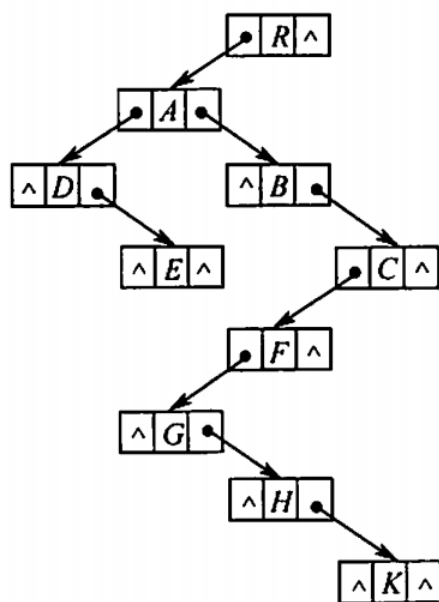
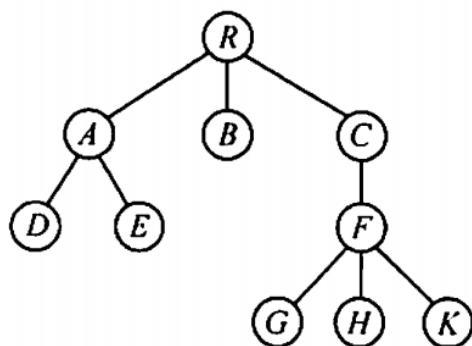
取出前10个数建立堆（求最大值建立小堆，求最小值建立大堆），再将后续的节点跟堆顶节点进行比较，满足条件就进行替换并向下调整堆结构，直到把所有数据比较完成，堆中的元素即为所求的最大值或最小值元素

 最小K个数 <https://leetcode.cn/problems/smallest-k-lcci/description/>

7、树和森林与二叉树的转换

📌 最常用，最好用的方法就是：左孩子右兄弟表示法

能够掌握树到二叉树的转换，同时也需要能够掌握二叉树到树形结构的转换



8、huffman树

📌 huffman树是带权路径WPL（Weighed Path Length）最小的树，也称最优二叉树

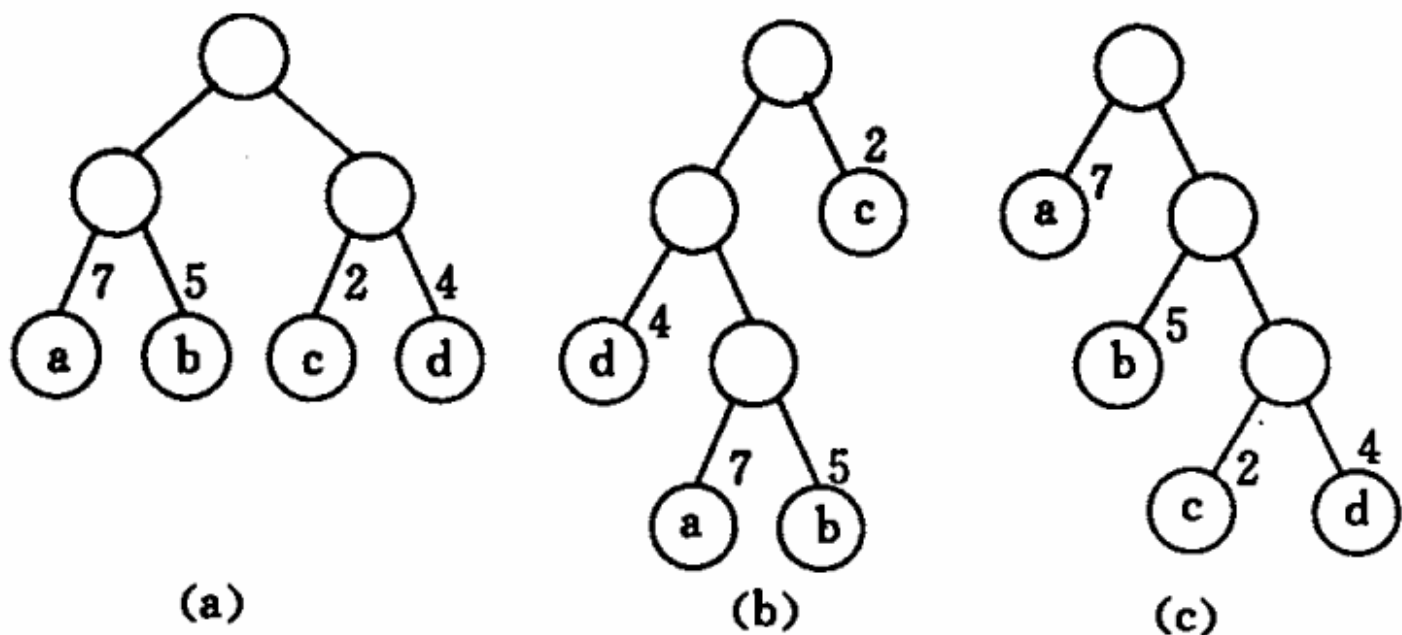



图 6.22 具有不同带权路径长度的二叉树


8.1 huffman树构造步骤

给定 n 个权值分别为 w_1, w_2, \dots, w_n 的结点，构造哈夫曼树的算法描述如下：

- 1) 将这 n 个结点分别作为 n 棵仅含一个结点的二叉树，构成森林 F 。
- 2) 构造一个新结点，从 F 中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和。
- 3) 从 F 中删除刚才选出的两棵树，同时将新得到的树加入 F 中。
- 4) 重复步骤 2) 和 3)，直至 F 中只剩下一棵树为止。

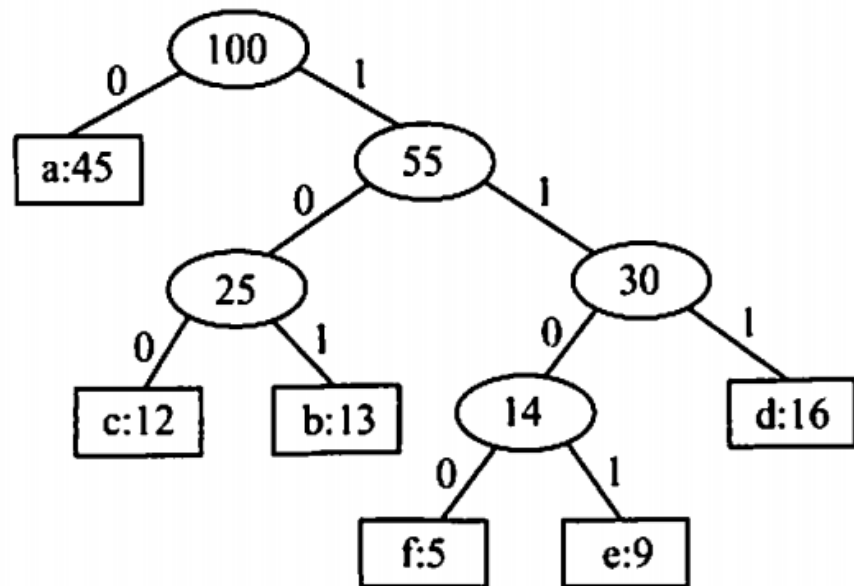
 例如：{11, 5, 6, 8, 2}; 构造一棵huffman树

8.2 huffman编码

 将huffman树的左树编码为0，右树编码为1，则每一个叶子结点将得到唯一的编码，即为huffman编码

各字符编码为

a:0
b:101
c:100
d:111
e:1101
f:1100



8.3 huffman压缩原理

以字符串中每个字符出现的次数为权值构建Huffman树

从根节点开始，左分支为0，右分支为1，如上图

所有权值节点都在叶子结点位置，遍历每条到叶子结点的路径获取字符的编码



举个例子：ABBBCCCCDDDDDDDD

Huffman编码：

A:100

B:101

C:11

D:0

压缩原理是：一个字符占一个字节，现在用二进制编码代替之后，一个字符只占三位，也就是说一个字节可以表示两三个字符，所以说一次压缩，就会节省很多字节，也就起到了压缩的作用。

8.4 huffman编码实现

```
1 #ifndef _HUFFMAN_H_
2 #define _HUFFMAN_H_
3
4 #define Huff_Elem_Type int
5
6 typedef struct HuffNode
7 {
8     Huff_Elem_Type data;
9     struct HuffNode *leftChild;
10    struct HuffNode *rightChild;
11    char code[10];
```

```

12 }HuffNode;
13
14 typedef HuffNode* HuffManTree;
15
16 ///////////////////////////////////////////////////
17 //小堆
18 #define MinHeap_Elem_Type    HuffNode*
19 #define MinHeap_Default_Size 10
20 typedef struct MinHeap
21 {
22     MinHeap_Elem_Type *heap;
23     size_t             capacity;
24     size_t             size;
25 }MinHeap;
26
27 void MinHeapSwap(MinHeap_Elem_Type *a, MinHeap_Elem_Type *b)
28 {
29     MinHeap_Elem_Type tmp = *a;
30     *a = *b;
31     *b = tmp;
32 }
33
34 bool MinHeapFull(MinHeap *php)
35 {
36     return php->size >= php->capacity;
37 }
38 bool MinHeapEmpty(MinHeap *php)
39 {
40     return php->size == 0;
41 }
42
43 void _MinAdjustUp(MinHeap *php, int start)
44 {
45     int j = start;
46     int i = (j-1) / 2;
47
48     while(j > 0)
49     {
50         if(php->heap[j]->data < php->heap[i]->data)
51         {
52             MinHeapSwap(&php->heap[j], &php->heap[i]);
53             j = i;
54             i = (j-1) / 2;
55         }
56         else
57             break;
58     }

```



```

59 }
60
61 void _MinAdjustDown(MinHeap *php, int start)
62 {
63     int i = start;
64     int j = 2 * i + 1; //左子树
65
66     while(j < php->size)
67     {
68         if(j+1<php->size && php->heap[j+1]->data<php->heap[j]->data) //
69             j++;
70         if(php->heap[i]->data > php->heap[j]->data)
71         {
72             MinHeapSwap(&php->heap[i], &php->heap[j]);
73             i = j;
74             j = 2 * i + 1;
75         }
76         else
77             break;
78     }
79 }
80
81 void MinHeapInit(MinHeap *php)
82 {
83     php->heap = (MinHeap_Elem_Type*)malloc(sizeof(MinHeap_Elem_Type) * MinHe
84     assert(php->heap != NULL);
85     php->capacity = MinHeap_Default_Size;
86     php->size = 0;
87 }
88 void MinHeapPush(MinHeap *php, MinHeap_Elem_Type v)
89 {
90     if(!MinHeapFull(php))
91     {
92         php->heap[php->size] = v;
93
94         //向上调整
95         _MinAdjustUp(php, php->size);
96
97         php->size++;
98     }
99 }
100
101 void HeapPop(MinHeap *php)
102 {
103     if(!MinHeapEmpty(php))
104     {
105         php->heap[0] = php->heap[php->size - 1];

```

```

106         php->size--;
107
108         //向下调整
109         _MinAdjustDown(php, 0);
110     }
111 }
112
113 MinHeap_Elem_Type HeapTop(MinHeap *php)
114 {
115     assert(!MinHeapEmpty(php));
116     return php->heap[0];
117 }
118
119 void MinHeapDestory(MinHeap* php)
120 {
121     free(php->heap);
122     php->heap = NULL;
123     php->capacity = php->size = 0;
124 }
125 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
126
127 void SelectMinVal(MinHeap *mhp, HuffNode **first_min, HuffNode **second_min)
128 {
129     *first_min = HeapTop(mhp);
130     HeapPop(mhp);
131     *second_min = HeapTop(mhp);
132     HeapPop(mhp);
133 }
134
135 HuffNode* MergeTree(HuffNode *first_min, HuffNode *second_min)
136 {
137     HuffNode *root = (HuffNode*)malloc(sizeof(HuffNode));
138     root->data = first_min->data + second_min->data;
139     root->leftChild = first_min;
140     root->rightChild = second_min;
141     return root;
142 }
143
144 void _AddHuffCode(HuffNode *node, const char *code)
145 {
146     strcat(node->code, code);
147     if(node->leftChild==NULL && node->rightChild==NULL)
148         return;
149     _AddHuffCode(node->leftChild, code);
150     _AddHuffCode(node->rightChild, code);
151 }
152 void AddHuffCode(HuffNode *root)

```

```

153 {
154     if(root->leftChild==NULL && root->rightChild==NULL)
155         return;
156     _AddHuffCode(root->leftChild, "0");
157     _AddHuffCode(root->rightChild, "1");
158 }
159
160 void ReverseHuffCode(HuffNode *node)
161 {
162     if(node->leftChild==NULL && node->rightChild==NULL)
163     {
164         int left = 0, right = strlen(node->code)-1;
165         while(left < right)
166         {
167             char tmp = node->code[left];
168             node->code[left] = node->code[right];
169             node->code[right] = tmp;
170             left++;
171             right--;
172         }
173     }
174     else
175     {
176         ReverseHuffCode(node->leftChild);
177         ReverseHuffCode(node->rightChild);
178     }
179 }
180
181 HuffManTree CreateHuffManTree(Huff_Elem_Type ar[], int n)
182 {
183     MinHeap mhp;
184     MinHeapInit(&mhp);
185
186     for(int i=0; i<n; ++i)
187     {
188         HuffNode *node = (HuffNode*)malloc(sizeof(HuffNode));
189         memset(node, 0, sizeof(HuffNode)); //code / leftChild / rightChild
190         node->data = ar[i];
191         MinHeapPush(&mhp, node);
192     }
193
194     HuffNode *first_min, *second_min, *root;
195     for(int i=0; i<n-1; ++i)
196     {
197         SelectMinVal(&mhp, &first_min, &second_min);
198         root = MergeTree(first_min, second_min);
199     }

```

```

200         AddHuffCode(root);
201
202         MinHeapPush(&mhp, root);
203     }
204     MinHeapDestory(&mhp);
205
206     ReverseHuffCode(root);
207     return root;
208 }
209
210 void ShowHuffCode(HuffNode *root)
211 {
212     if(root->leftChild==NULL && root->rightChild==NULL)
213         printf("%d : %s\n", root->data, root->code);
214     else
215     {
216         ShowHuffCode(root->leftChild);
217         ShowHuffCode(root->rightChild);
218     }
219 }
220
221 #endif /* _HUFFMAN_H_ */
222
223 int main()
224 {
225     int ar[] = {11, 5, 6, 8, 2};
226     int n = sizeof(ar) / sizeof(ar[0]);
227
228     HuffManTree hmt = NULL;
229
230     hmt = CreateHuffManTree(ar, n);
231
232     ShowHuffCode(hmt);
233
234     return 0;
235 }

```