哈夫曼树

赫夫曼树，别名“哈夫曼树”、“最优树”以及“最优二叉树”。

路径：在一棵树中，一个结点到另一个结点之间的通路，称为路径。
路径长度：在一条路径中，每经过一个结点，路径长度都要加 1 。例如在一棵树中，规定根结点所在层数为1层，那么从根结点到第 i 层结点的路径长度为 i - 1 。
结点的权：给每一个结点赋予一个新的数值，被称为这个结点的权。
结点的带权路径长度：指的是从根结点到该结点之间的路径长度与该结点的权的乘积。
树的带权路径长度为树中所有叶子结点的带权路径长度之和。通常记作“WPL”。

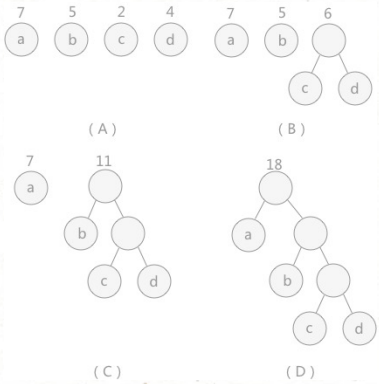
什么是哈夫曼树

当用 n 个结点（都做叶子结点且都有各自的权值）试图构建一棵树时，如果构建的这棵树的带权路径长度最小，称这棵树为“最优二叉树”，有时也叫“赫夫曼树”或者“哈夫曼树”。
在构建哈弗曼树时，要使树的带权路径长度最小，只需要遵循一个原则，那就是：权重越大的结点离树根越近。

构建哈夫曼树

对于给定的有各自权值的 n 个结点，构建哈夫曼树有一个行之有效的办法：

1. 在 n 个权值中选出两个最小的权值，对应的两个结点组成一个新的二叉树，且新二叉树的根结点的权值为左右孩子权值的和；
2. 在原有的 n 个权值中删除那两个最小的权值，同时将新的权值加入到 n–2 个权值的行列中，以此类推；
3. 重复 1 和 2 ，直到所以的结点构建成了一棵二叉树为止，这棵树就是哈夫曼树。



（A）给定了四个结点a，b，c，d，权值分别为7，5，2，4；第一步如（B）所示，找出现有权值中最小的两个，2 和 4 ，相应的结点 c 和 d 构建一个新的二叉树，树根的权值为 2 + 4 = 6，同时将原有权值中的 2 和 4 删掉，将新的权值 6 加入；进入（C），重复之前的步骤。直到（D）中，所有的结点构建成了一个全新的二叉树，这就是哈夫曼树。

哈弗曼树中结点结构

构建哈夫曼树时，首先需要确定树中结点的构成。由于哈夫曼树的构建是从叶子结点开始，不断地构建新的父结点，直至树根，所以结点中应包含指向父结点的指针。但是在使用哈夫曼树时是从树根开始，根据需求遍历树中的结点，因此每个结点需要有指向其左孩子和右孩子的指针。

```
class Node {
    data = null;
    weight = null; // 结点权重
    //父结点、左孩子、右孩子在数组中的位置下标
    left = null;
    right = null;
    parent = null;
}
```

哈弗曼树中的查找算法

构建哈夫曼树时，需要每次根据各个结点的权重值，筛选出其中值最小的两个结点，然后构建二叉树。

查找权重值最小的两个结点的思想是：从树组起始位置开始，首先找到两个无父结点的结点（说明还未使用其构建成树），然后和后续无父结点的结点依次做比较，有两种情况需要考虑：

- 如果比两个结点中较小的那个还小，就保留这个结点，删除原来较大的结点；
- 如果介于两个结点权重值之间，替换原来较大的结点；

构建哈弗曼树

```
class Node {
  constructor(weight) {
    this.weight = weight // 结点权重
    //父结点、左孩子、右孩子在数组中的位置下标
    this.left = null
    this.right = null
    this.parent = null
  }
}

class CreateHuffmanTree {
  constructor(arr) {
    this.resolve = [];
    this.tree_node = this.init_tree(arr);
  }
  init_tree(ary) {
    const _arr = ary;
    for (let i = 0; i < _arr.length; i++) {
      const node = new Node(_arr[i]);
      this.insert_node(node);
    }
    const resolve = this.resolve;
    while (resolve.length !== 1) {
      const first_node = resolve.shift();
      const se_node = resolve.shift();
      const weight = first_node.weight + se_node.weight;
      const new_node = new Node(weight);
      new_node.left = first_node;
      new_node.right = se_node;
      first_node.parent = new_node;
      se_node.parent = new_node;
      this.insert_node(new_node);
    }
    return resolve[0];
  }
  insert_node(node) {
    const resolve = this.resolve;
    if (resolve.length) {
```

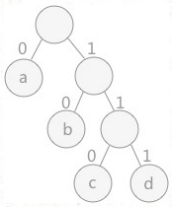
```
      for (let i = 0; i < resolve.length; i++) {
        if (resolve[i].weight > node.weight) {
          resolve.splice(i, 0, node)
          return;
        }
      }
    }
    resolve.push(node);
  }
}
```

哈夫曼编码

哈夫曼编码就是在哈夫曼树的基础上构建的，这种编码方式最大的优点就是用最少的字符包含最多的信息内容。

根据发送信息的内容，通过统计文本中相同字符的个数作为每个字符的权值，建立哈夫曼树。对于树中的每一个子树，统一规定其左孩子标记为 0 ，右孩子标记为 1 。这样，用到哪个字符时，从哈夫曼树的根结点开始，依次写出经过结点的标记，最终得到的就是该结点的哈夫曼编码。

文本中字符出现的次数越多，在哈夫曼树中的体现就是越接近树根。编码的长度越短。



符 a 用到的次数最多，其次是字符 b 。字符 a 在哈夫曼编码是 0 ，字符 b 编码为 10 ，字符 c 的编码为 110 ，字符 d 的编码为 111。

使用程序求哈夫曼编码有两种方法：

1. 从叶子结点一直找到根结点，逆向记录途中经过的标记。例如，字符 c 的哈夫曼编码从结点 c 开始一直找到根结点，结果为：0 1 1，所以字符 c 的哈夫曼编码为：1 1 0（逆序输出）。
2. 从根结点出发，一直到叶子结点，记录途中经过的标记。例如，字符 c 的哈夫曼编码，就从根结点开始，依次为：1 1 0。

```
//从根结点出发
get_code = function () {
  var get_code_from_tree = function (node, dict, code_str) {
    if (!node.left && !node.right) {
      // 页节点
      dict[node.weight] = code_str;
      return;
    }

    if (node.left) {
      get_code_from_tree(node.left, dict, code_str + "0");
    }
    if (node.right) {
      get_code_from_tree(node.right, dict, code_str + "1");
    }
  };
  // 获得最终的变长编码
  var code_dict = {};
  get_code_from_tree(this.root, code_dict, "");
  return code_dict;
```

```
21  };
```