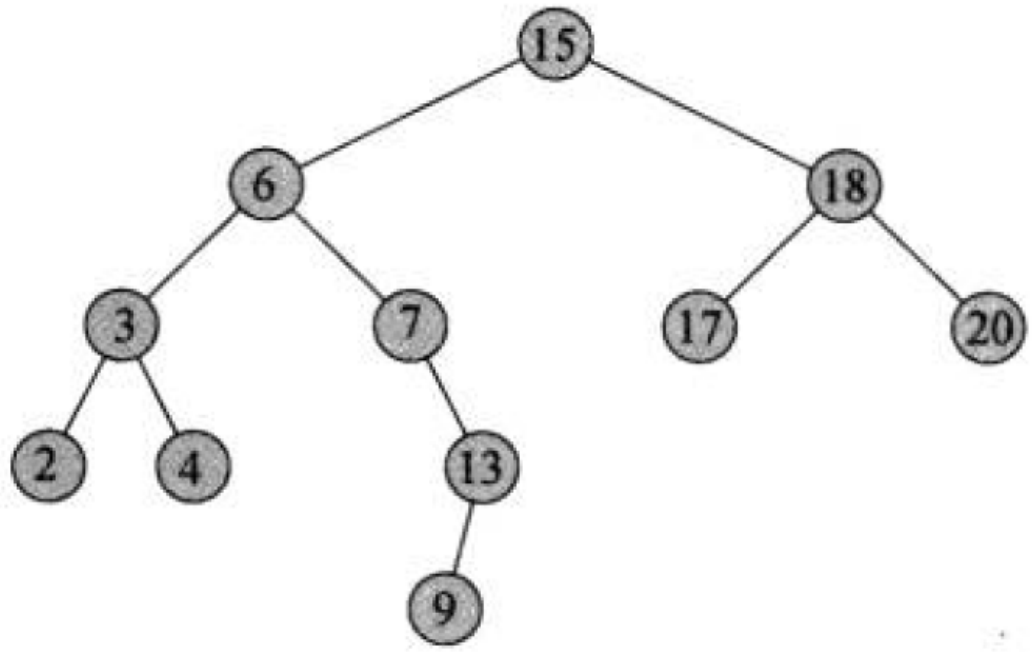
树的应用

> by ReActor

1. 二叉搜索树

定义：

设 x 是二叉搜索树中的一个结点。如果 y 是 x **左子树**中的一个结点，那么 $y.key \leq x.key$。如果y 是 x **右子树**中的一个结点，那么 $y.key \geq x.key$ 。



遍历：

中序遍历算法能够按顺序输出二叉搜索树中所有的关键字。

1.1 查询二叉搜索树

1.1.1 查找

`TREE-SEARCH` 返回一个指向关键字为k 的结点的指针；否则返回 `NULL`

递归实现：

```
BTNode TREE-SEARCH(x,k)// 递归版本
{
    if(x==NULL or k==x.key)
        return x;
    if(k < x.key)
        return TREE-SEARCH(x.left,k);
    else return TREE-SEARCH(x.right,k);
}
```

迭代实现：

```
BTNode TREE-SEARCH(x,k)
{
    while(x != NULL && k!=x.key)
    {
        if(k < x.key)
            x=x.left;
        else
            x=x.right;
    }
    return;
}
```

实现的时间复杂度为 $O(h)$，其中 h 为BST 的高度，也就是logN级别的；

1.1.2 最值查询

通常的最值查找也需要O(N)的时间，在二叉搜索树下，仍然是logN级别的复杂度；

```
TREE-MINIMUM(x){
    while(x.left != NULL)
        x=x.left;
    return x;
}
TREE-MAXIMUM(x){
    while(x.right != NULL)
        x=x.right;
    return x;
}
```

1.1.3 后继和前驱

二叉搜索树还支持快速， (logN) 查找后继、前驱结点。

我们假定所有 key 值互不相同，则一个结点 x 的后继是其值大于 $x.key$ 的，值最小的结点。

二叉搜索树的结构允许我们通过没有任何关键字的比较来确定一个结点的后继。

后继和前驱是对称的，以查找后继为例，先看伪代码：

如果后继存在，该算法返回目标结点，如果x 是树中最大的结点，则返回`NULL`

```
TREE-SUCCESSOR(x){
    if(x.right != NULL)
        return TREE-MINIMUM(x.right)
    y=x.p;
    while(y != NULL && x == y.right)
        x = y;
        y = y.p;
    return y;
}
```

第一种情况， 若 x 存在非空的右子树，那么右子树中最小者为所求，通过`TREE-MINIMUM`实现；

第二种情况， 若 x 右子树为空，那么可能比 x 值更大的结点必然出现在右上侧。事实上，从 x 开始沿树而上，直到遇到由上而下左向的边，边的上端结点即为所求。

证明：

在沿树向上的过程中，在遇到第一条左向边之前，必然全为右向边，而右向边意味着源结点 x 的值，比路径上的结点，及其左子树都更大，是不可能满足要求的。设第一条左向边的上端结点为 y，出现第一条左向边则意味着：

1. x 处在 y 的左子树中，满足 $x.key \leq y.key$
2. x 处在 y 的子树中，y 的祖先的值，要么比 y 的整棵子树小(因此比x 小，不合法)，要么比 y 的值大(因此比 y 更劣)，不会产生答案
3. y 的右子树内结点均比 y 大，不可能产生答案

第一点说明了 y 的合法性，二三点说明了 y 的最优性。

为什么在第一种情况出现的时候，第二种情况不会出现呢?

1.2 插入和删除

1.2.1 插入

调用 `TREE-INSERT`，以z 结点作为输入，其中 $z.key = v, z.left = NULL, z.right = NULL$，将其插入到BST上的叶子结点位置。

```
TREE-INSERT(T,z){
    y = NULL;
    x = T.root;
    while(x != NULL){
        y = x;
        if(z.key < x.key)
            x = x.left;
        else x = x.right;
    }
    z.p = y;
    if(y == NULL)
        T.root = z;
    else if(z.key < y.key)
        y.left = z;
    else y.right = z;
}
```

过程中维护一条自上而下的边，其上下端点分别为 y, x，初始化为 `NULL` 和根节点

按通常的查找 $z.key$ 操作，找到能够不破坏BST性质的叶子结点位置，(此时 x = NULL，y 为叶子结点)，并用 y 结点替换 x；

特别处理 $x = y = NULL$ 的情况，此时BST为空，直接令 z 为根节点。

1.2.2 建树

通常采用逐点插入的方式构建二叉搜索树。

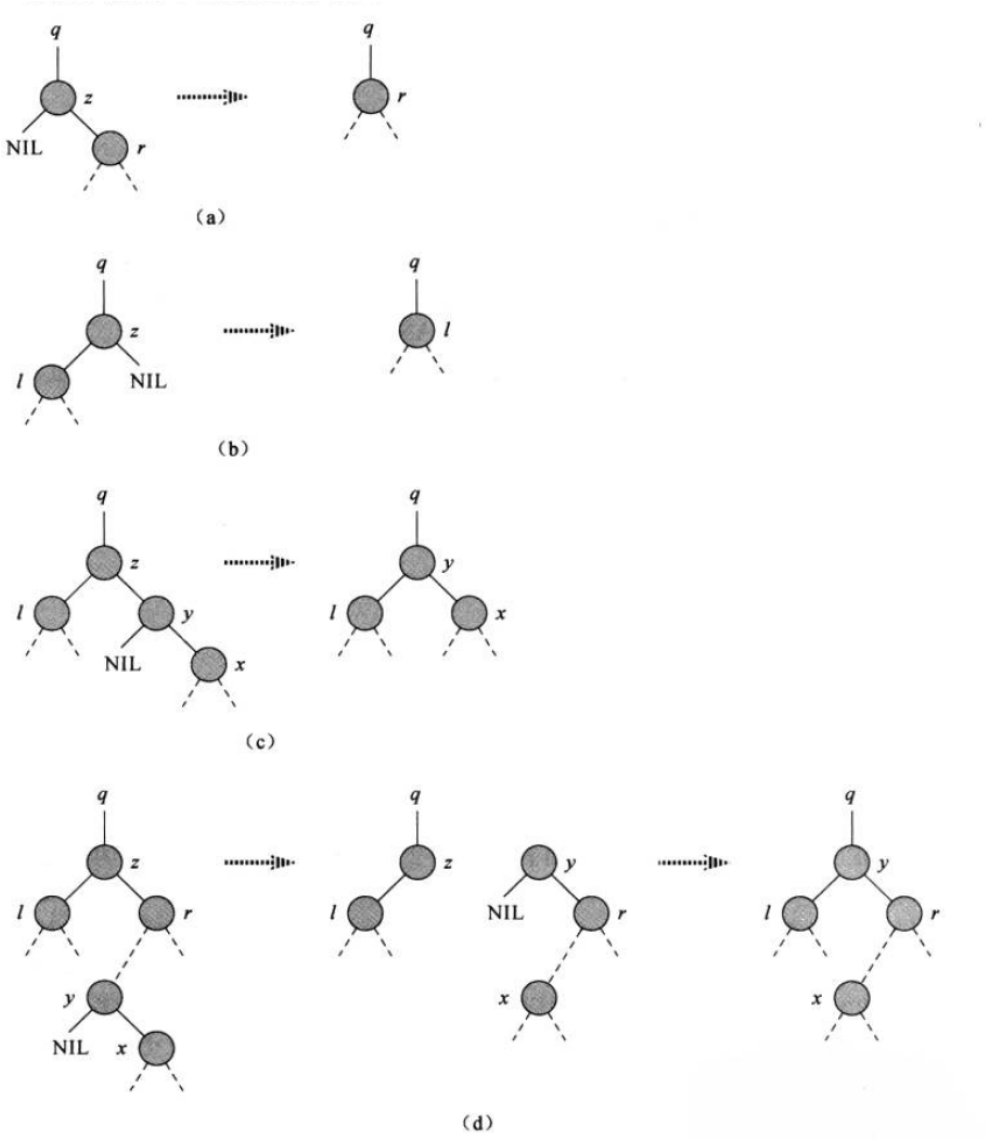
```
BTREE BUILD-TREE(datatype K[],int n)
{
    BTREE T=NULL,p;
    for(int i=0;i<n;++i)
    {
        p=(BTREE)malloc(sizeof(BTNode));
        p->data=K[i];
        p->lch=NULL;
        p->rch=NULL;
        TREE-INSERT(T,p);
    }
    return T;
}
```

//实现过程中最好改用传址调用(通过指针)，否则容易在实现过程中超时。

1.2.3 删除

从BST中删除一个结点 z 需要删除其本身，并调整与其相关结点的指针，其策略大致分为下面三种情况：

1. 如果 z 没有孩子结点，简单地将其删除，并修改其父节点，用 `NULL` 来替换 z
2. 如果 z 仅有一个孩子 y ，那么将这个孩子提升到树中 z 的位置上，并修改 z 的父节点，使其指向 y
3. 如果 z 有两个孩子，尝试用 z 的后继来替换 z
    - 若 z 的后继 y 为其右儿子，则可以直接用 y 替换 z
    - 若 z 的后继 y 并非其右儿子，先用 y 的右儿子替换 y (将后继从原位置删除)，再用 y 替换 z



简化实现：

定义过程 `TRANSPLANT` ，用一棵以 v 为根的子树，来替换一棵以 u 为根的子树：

这个过程假定 u, v 作为子树的结构都是正确的，只替换 u 的父节点上的双向指针。

```
TRANSPLANT(T,u,v){
    if(u.p == NULL)//特判根指针
        T.root = v;
    else if(u == u.p.left)
        u.p.left=v;
    else u.p.right=v;
    if(v != NULL)
        v.p=u.p;
}
```

删除：

```
TREE-DELETE(T,z){
    if(z.left == NULL)
        TRANSPLANT(T,z,z.right);
    else if(z.right == NULL)
        TRANSPLANT(T,z,z.left);
    else
    {
```

```
        y = TREE-MINIMUM(z.right)
        if(y.p != z)
        {
            TRANSPLANT(T,y,y.right);//调整构造以 y 为根的子树
            y.right = z.right;
            y.right.p = y;
        }
        TRANSPLANT(T,z,y);
        y.left = z.left;
        y.left.p = y;
    }
}
```

`TRANSPLANT` 复杂度为`O(1)`，故删除结点的复杂度等同于最值查询，为 $O(h)$

1.3 效率分析

同样的数据，构建得到的BST可能是不同的，一次查找所花费的时间，与目标结点所在的深度有关。

我们可以用**平均查找长度**来衡量查找的效率。

> **平均查找长度**(Average Search Length, ASL) 确定一个元素在树中位置所需要进行的比较次数的期望值。
>
> **内路径长度**(Internal Path Length, IPL)：所有节点的深度之和；
>
> **扩充**：当二叉树中出现空子树时，增加新的叶节点来表示空子树；
>
> **外部节点**：扩充二叉树中新增的结点；
>
> **内部节点**：扩充二叉树中原本存在的结点；
>
> **外路径长度**(External Path Length, EPL：扩充二叉树中所有外部节点的深度之和；

性质：

若有 n 个内部节点，则扩充后的二叉树有 $n+1$ 个外部节点；

且有：

$$EPL = IPL + 2n$$

在 n 个结点的二叉排序树中，若对每个数据的查找概率相等，并且对每个节点成功查找的长度为深度加一(包含与自身相等或 `NULL` 的那次比较)，则查找成功时：

$$ASL = (ISL + n)/n$$

若查找不成功，比较次数恰好为外部节点的路径长度：

$$ASL = \frac{ESL}{n} = \frac{ISL + 2n}{n}$$

在同时考虑两者的情况下，假设扩充二叉树的每个节点进行查找的概率相等，则平均查找长度为：

$$ASL = \frac{ESL + ISL + n}{n + n + 1} = \frac{2IPL + 3n}{2n + 1}$$

因此要使得BST有更高效率，或者说具有更好的平均查找长度，应该使内路径长度最小。

若我们总是尽力构造满二叉树，使得所有节点的深度都尽可能小，此时的查询复杂度为 $O(log_2 N)$ ，而最坏情况下，则可能退化到 $O(N)$

1.4 平衡二叉树

平衡搜索树是一类树，他们通过维护独特的性质，来使得运行过程中搜索树的复杂度不退化。

比较典型的有红黑树，通过给每个节点标记红色或者黑色，并且维护红黑色节点之间的相对关系，使得树的深度不超过 $2log_2(N+1)$

2. 哈夫曼树

2.1 带权路径长度

在之前的分析过程中，我们假定每个节点访问的概率都相等，但事实上，结点之间的重要性可能截然不同，有些十分重要的结点或许会被反复访问，而有些无关紧要的结点，可能几乎不被访问到。我们基于这些想法可以设计出整体上更优的数据结构。

假设图中共有 m 个结点，每个节点有权重 w_i ，深度为 d_i ，则**带权路径长度**(Weighted Path Length, WPL)为：

$$WPL=\sum_{i=1}^{m}w_i\cdot d_i$$

给定一组权值，构造出的具有最小 WPL 的二叉树，称为 **哈夫曼树**，或者 **最优二叉树。**

2.2 哈夫曼编码

2.2.1 前缀码与树

字符编码过程中，我们把一个字符与一个二进制编码对应。

假定我们需要编码一份若干字的文件，那么编码的总长度就是每个字符对应编码的长度，与他出现的频率乘积的求和。

我们怎样能使得编码的总长度更小?

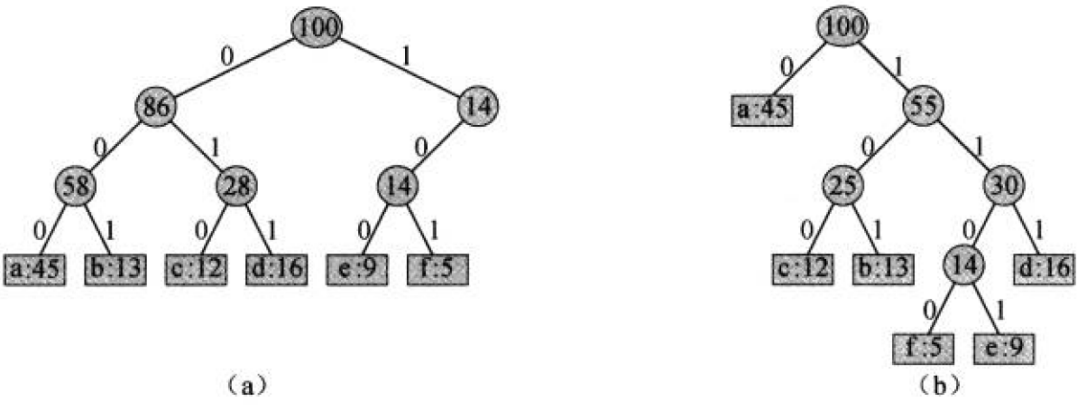
变长编码：赋予高频字符更短的字码，能达到比定长编码更好的压缩率。

前缀码：在变长编码的前提下，如果我们保证每个字符对应的编码不是任何其他字符的前缀，能极大地方便解码的过程。

并且事实上，前缀码能够保证达到最高数据压缩率，这里我们了解这个性质就好。

用二叉树表示前缀码：

如果0代表着转向左儿子，1代表转向右儿子，我们会得到一颗二叉树，每个叶子结点对应一个字符。



我们给每个节点自身都附加一个权值，为其所被访问到的频率，就得到一棵由特定的前缀码对应的树。

> 注意，这里得到的树并非BST，权值之间并不要求满足BST的性质。

满二叉树的另一个定义：

> **满二叉树**：每个非叶子结点都有两个孩子结点。

而文件的最优编码方案总是对应一棵满二叉树。

注意到这和我们之前分析BST效率时定义的内、外部结点很相似。如果把叶子结点当做外部节点，分支节点当做内部节点，

同样会有：内部节点+1 = 外部节点 的性质。

给定一棵前缀码对应的树 T ，编码文件所需的长度(代价)即为 $B(T) = \sum c.freq \cdot d_T(c)$

哈夫曼设计了一个贪心算法来构造前缀码，通过这种方式构造出来的，称作**哈夫曼编码**。

2.2.2 构造哈夫曼编码

我们先给出构造的算法，再从贪心的角度证明算法的正确性。

> **最小优先队列**：一种特化的队列，其队头总是队列中关键字最小的元素，也称为小根堆。

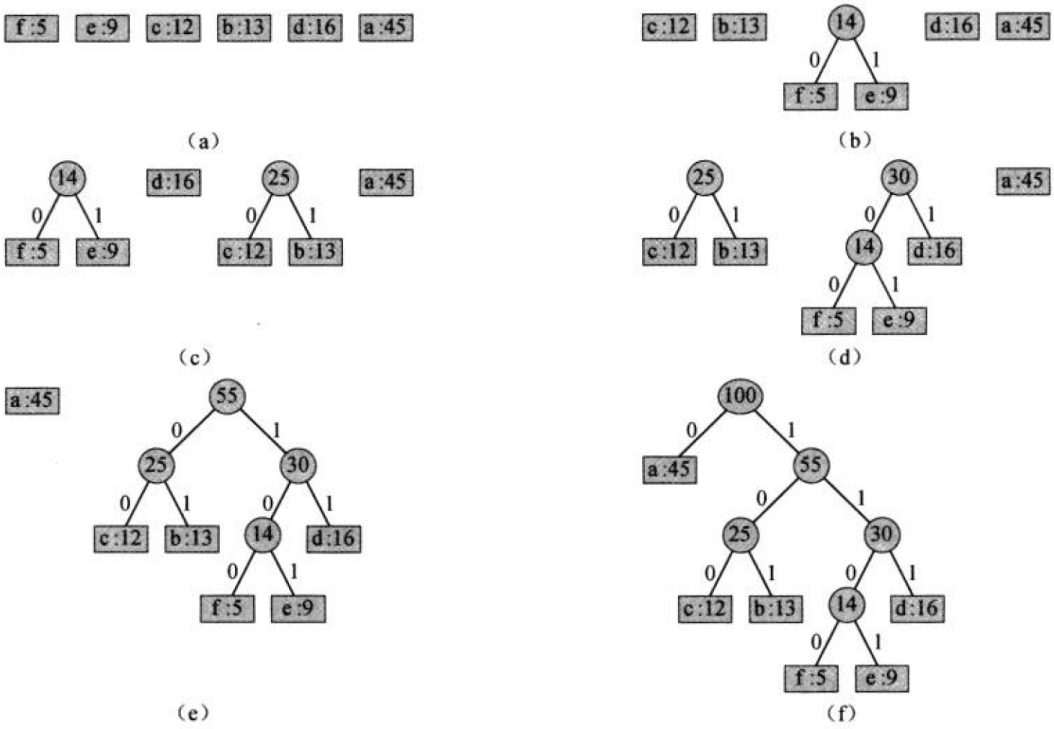
设 C 为字符集合，将字符 $c \in C$ 看做对象，并且具有属性值 $c.freq$ 描述其出现的频率。

算法从$|C|$ 个叶节点开始，自底向上，经过 $|C| - 1$ 次合并，创建出最终的哈夫曼树。

2.2.2.1 算法流程：

1. 初始化以 $c.freq$ 为关键字的最小优先队列Q ，将所有 C 中字符加入Q 中；
2. 取出 Q 中频率最低的两个对象，将其合并，合并后的对象频率为二者的频率之和，并且为两个对象的父亲；
3. 将合并后的对象加入 Q 中；
4. 回到步骤 2

```
HUFFMAN(C){
    Q=C;
    for(int i=1;i<n;++i)
    {
        allocate a new node z;
        z.left = x = EXTRACT-MIN(Q);// EXTRACT-MIN(Q)取出Q中最小元素，并将其出队；
        z.right = y = EXTRACT-MIN(Q);
        z.freq = x.freq + y.freq;
        INSERT(Q,z);
    }
    return EXTRACT-MIN(Q);//此时返回的是根节点；
}
```



2.2.2.2 算法证明：

这部分内容依赖于算法拓展的第一部分贪心算法，接下来课程讲解的最小生成树等图论算法也会用到，非常建议温习一下。

贪心选择：

总是将频率更低的结点置于更深层。

最优子结构：

证明合并后得到的哈夫曼树等价于合并前的。

2.3 自适应(动态)哈夫曼树

在原哈夫曼树的构建过程中，我们预先知道了文本中各字符的出现频率。

自适应哈夫曼算法采用了先编码，后调整编码树的方案，可以解决未知符号频率的问题。