5.二叉树

(g) Huffman树

邓俊辉

deng@tsinghua.edu.cn

策略与算法

❖ 自下而上构造Huffman树 //稍后可见，它的确是最优编码树之一

//贪婪策略：在构造编码树的过程中，频率低的字符优先引入

//贪心目标：在构造出来的编码树中，频率低的字符位置更低

为每个字符创建一棵单节点的树，组成森林F

按照出现频率，对所有树（非降）排序

while (F中的树不止一棵) {

取出频率最小的两棵树：T_1和T_2

将它们合并成一棵新树T，满足：

$lchild(T) = T_1$ 且 $rchild(T) = T_2$

$w(root(T)) = w(root(T_1)) + w(root(T_2))$

}

正确性？

❖ 贪婪策略？

在多数场合并不适用

不见得能得到最优解

甚至反而得到最差解 //比如，最短路径

❖ Huffman树的构造采用了贪婪策略，它是最优编码树？总是？

❖ 易见：任一指定频率的字符集，都存在对应的最优编码树

❖ 然而，最优编码树可能不止一棵

❖ 断言：Huffman树必是其中之一 //为什么？

❖ 不妨，先来考察最优编码树的特性...

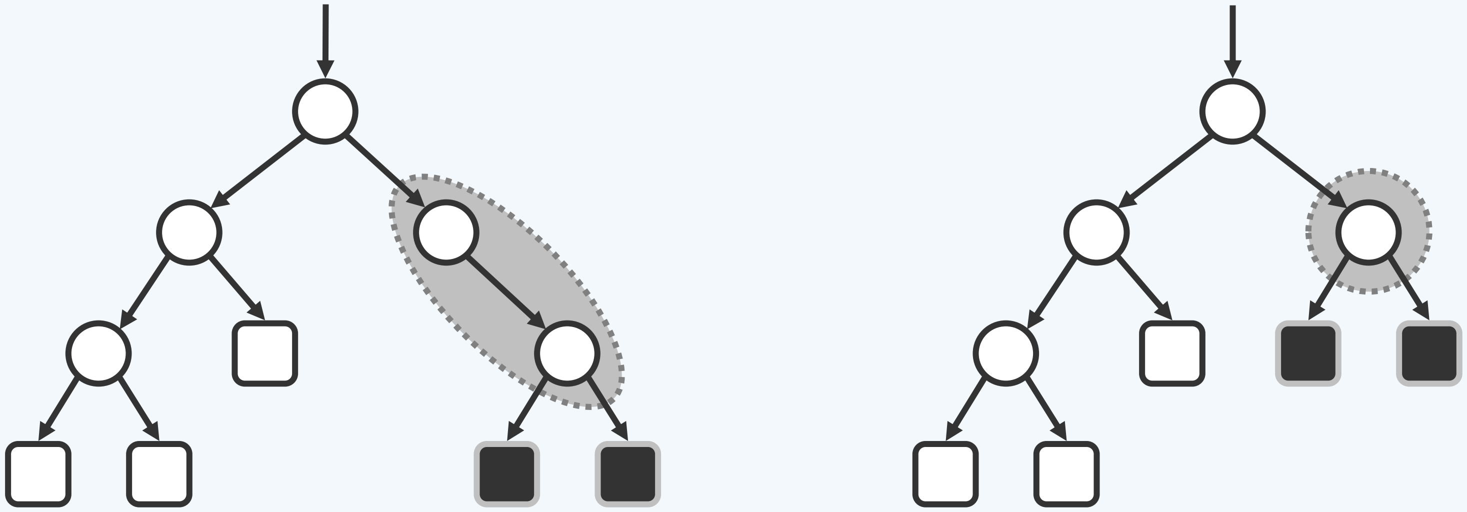
双子性

❖ 只要|Σ| > 1，最优编码树中每一内部节点都有两个孩子，亦即

节点度数均为偶数（0或2）

Huffman树必为真二叉树

❖ 否则，将1度节点替换为其唯一的孩子，则新树的wald将更小



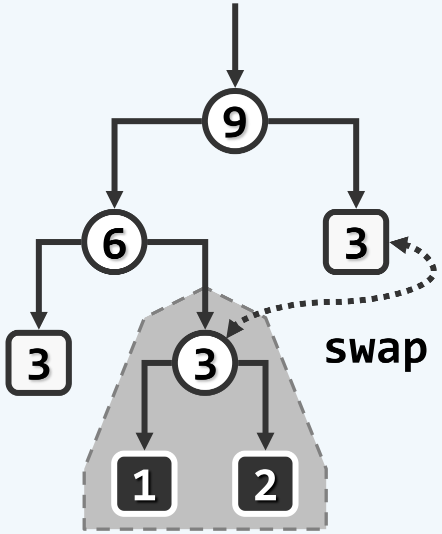
不唯一性

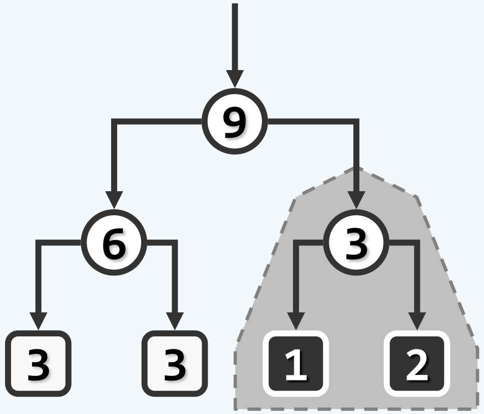
❖ 任一内部节点的左、右子树相互交换之后，wald不变

//上述算法中，左右子树的次序可以随机选取，故此...

❖ 为消除这种歧义，可以（比如）明确要求左子树的频率更低

❖ 不过，倘若它们（甚至更多节点）的频率恰好相等...



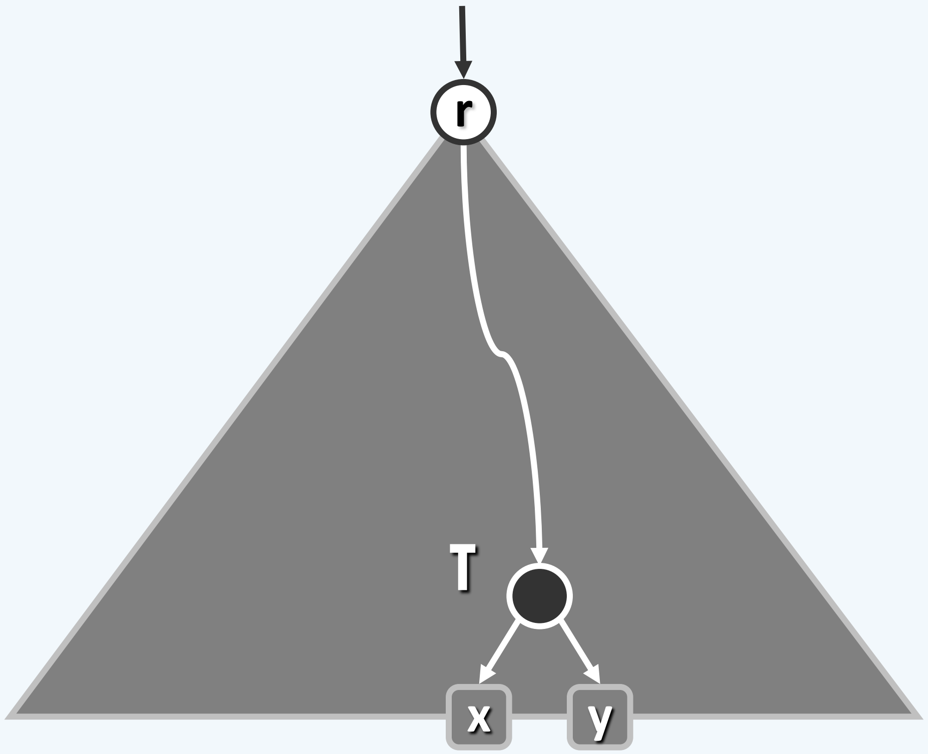


层次性

❖ 若：在字符表中，x和y是出现频率最低的两个字符

则：存在**某棵**最优编码树，x和y在其中处于最**底层**，且互为**兄弟**

❖ 为什么？

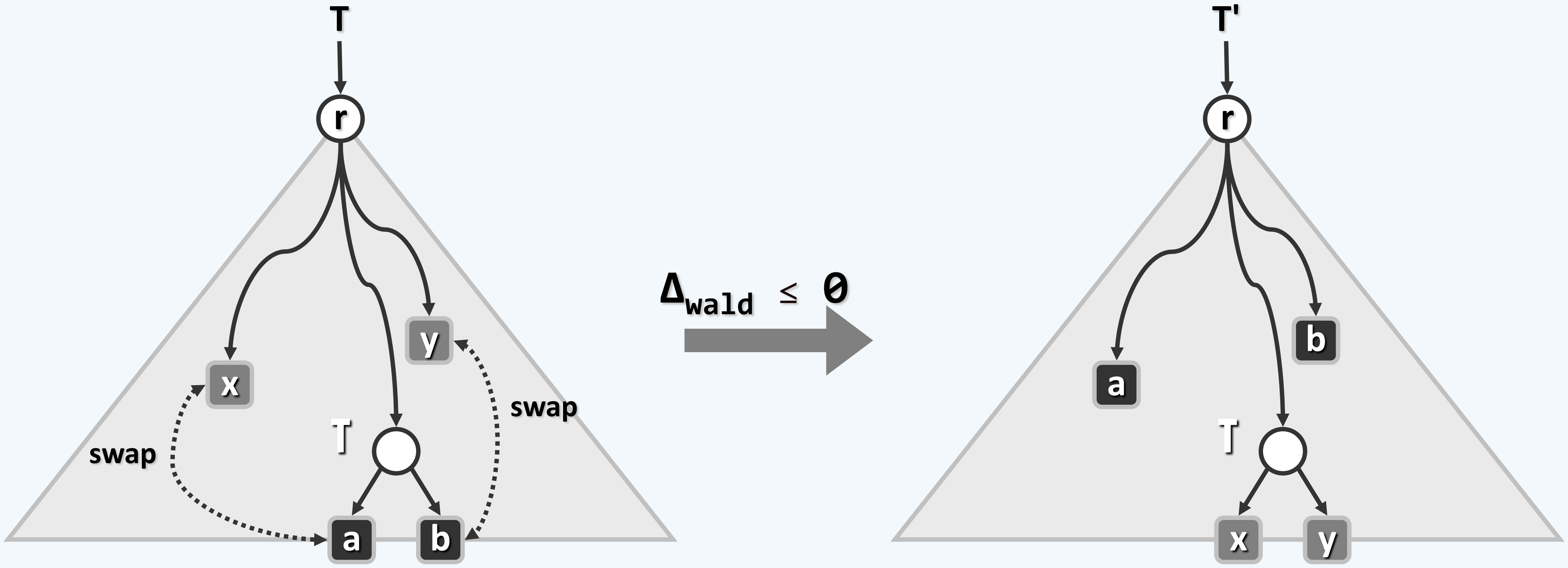


层次性

❖ 任取一棵最优编码树 //注意T的存在性

在其最底层，任取一对兄弟a和b //同样，注意其存在性

分别交换a和x、b和y后，wald绝不会增加 //正如此前已看到的



正确性

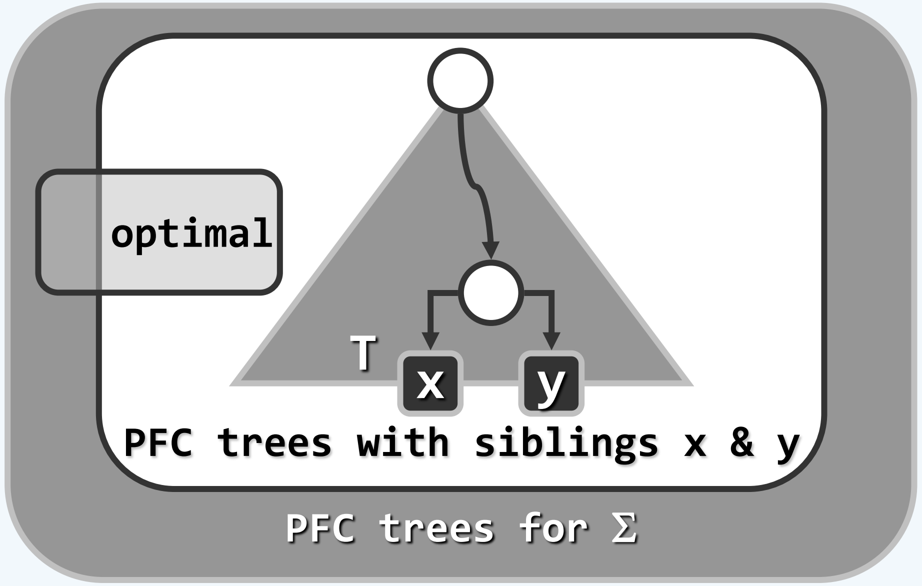
❖ Huffman（算法所生成的）编码树，的确最优！

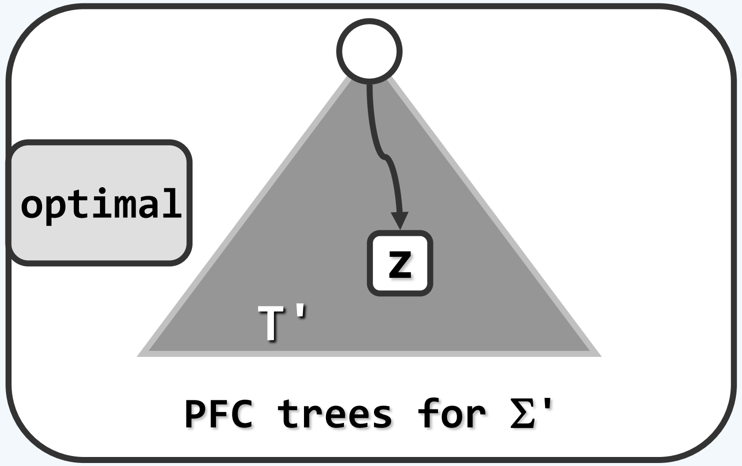
❖ 对|Σ|做归纳：|Σ| = 1时显然

设|Σ| < n时Huffman算法都能最优编码，考虑|Σ| = n的情况...

❖ 取Σ中频率最低的x和y　　//由层次性，仅考虑其互为兄弟的情形

❖ 令Σ' = (Σ\{x, y}) ∪ {z}，w(z) = w(x) + w(y)





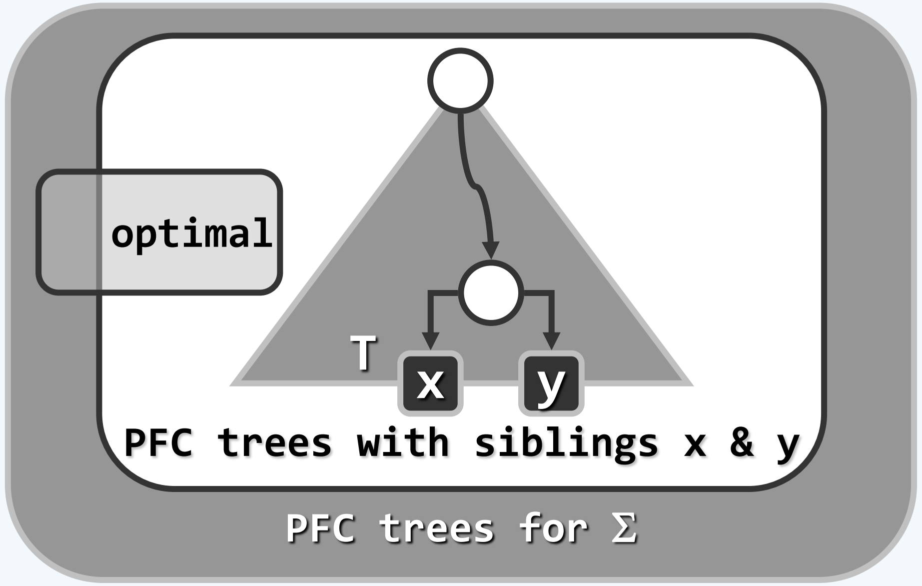
正确性

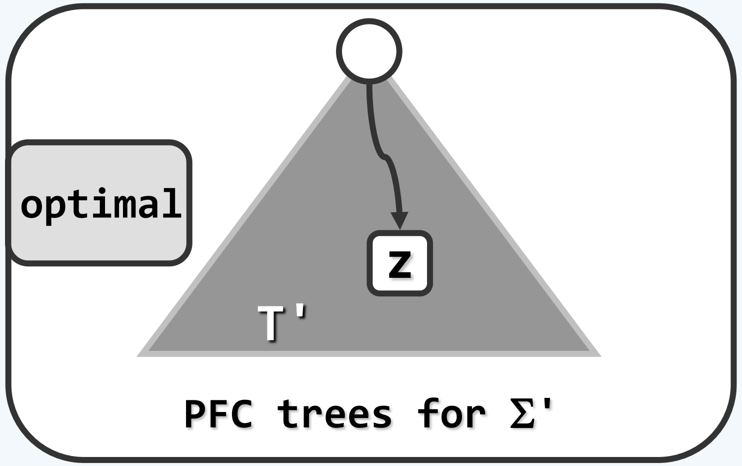
❖ 对于Σ'的任一编码树T'，只要为z添加孩子x和y，即可得到Σ的一棵编码树T，且wd(T) - wd(T') = w(x) + w(y) = w(z)

❖ 亦即，如此对应的T和T'，wd之差与T的具体形态无关

❖ 因此，只要T'是Σ'的最优编码树，则T也必是Σ的最优编码树（之一）

❖ 实际上，Huffman算法的过程，与上述归纳过程完全一致





实现：构造编码树

```
❖ #define HuffTree BinTree<HuffChar> //Huffman树，节点类型HuffChar
  typedef List<HuffTree*> HuffForest; //Huffman森林
❖ HuffTree* generateTree(HuffForest* forest) { //Huffman编码算法
    while (1 < forest->size()) { //反复迭代，直至森林中仅含一棵树
      HuffTree *T1 = minHChar(forest), *T2 = minHChar(forest);
      HuffTree* S = new HuffTree(); //创建新树，准备合并T1和T2
      S->insertAsRoot(HuffChar('^', //根节点权重，取作T1与T2之和
        T1->root()->data.weight + T2->root()->data.weight));
      S->attachAsLC(S->root(), T1); S->attachAsRC(S->root(), T2);
      forest->insertAsLast(S); //T1与T2合并后，重新插回森林
    } //assert: 循环结束时，森林中唯一的那棵树即Huffman编码树
    return forest->first()->data; //故直接返回之
  }
```

实现：搜索最小超字符

- Huffman编码的整体效率，直接决定于minHChar()的效率

  以下版本仅达到$\mathcal{O}(n)$，整体为$\mathcal{O}(n^2)$

```
HuffTree* minHChar( HuffForest * forest ) {
   ListNodePosi( HuffTree* ) p = forest->first(); //从首节点出发
   ListNodePosi( HuffTree* ) minChar = p; //记录最小树的位置及其
   int minWeight = p->data->root()->data.weight; //对应的权重
   while (forest->valid(p = p->succ)) //遍历所有节点
      if( minWeight > p->data->root()->data.weight ) { //如必要
         minWeight = p->data->root()->data.weight; minChar = p; //则更新记录
      }
   return forest->remove( minChar ); //从森林中摘除该树，并返回
}
```

实现：构造编码表

```
❖ #include "../Hashtable/Hashtable.h" //用HashTable（第9章）实现
  typedef Hashtable< char, char* > HuffTable; //Huffman编码表
❖ static void generateCT //通过遍历获取各字符的编码
     ( Bitmap* code, int length, HuffTable* table, BinNodePosi(HuffChar) v) {
     if ( IsLeaf(*v) ) //若是叶节点（还有多种方法可以判断）
        { table->put( v->data.ch, code->bits2string(length) ); return; }
     if ( HasLChild(*v) ) //Left = 0，深入遍历
        { code->clear(length); generateCT(code, length + 1, table, v->lChild); }
     if ( HasRChild(*v) ) //Right = 1
        { code->set(length);   generateCT(code, length + 1, table, v->rChild); }
  } //总体O(n)
```

改进

❖ 方案1　　$\mathcal{O}(n^2)$

初始化时，通过**排序**得到一个**非升**序**向量**　　$\mathcal{O}(n\log n)$

每次（从**后**端）取出频率最低的两个节点　　$\mathcal{O}(1)$

将合并得到的新树插入向量，并保持有序　　$\mathcal{O}(n)$

❖ 方案2　　$\mathcal{O}(n^2)$

初始化时，通过**排序**得到一个**非降**序**列表**　　$\mathcal{O}(n\log n)$

每次（从**前**端）取出频率最低的两个节点　　$\mathcal{O}(1)$

将合并得到的新树插入列表，并保持有序　　$\mathcal{O}(n)$

❖ 方案3　　$\mathcal{O}(n\log n)$　　//稍后第10章...保持兴趣

初始化时，将所有树组织为一个**优先队列**　　O(n)

取出频率最低的两个节点，合并得到的新树插入队列　　O(logn) + O(logn)

改进

❖ 方案4 $\mathcal{O}(n)$

所有字符按频率排序 //O(nlogn)

使用$\mathcal{O}(n)$空间，维护两个有序队列...