ADS

AVL TREES, SPLAY TREES 自平衡的二叉搜索树

AVL Trees（左高和右高不超过1）

AVL树满足height-balanced，对于任意一个节点，左子树和右子树高度相差不超过1（$|h_L - h_R| \leq 1$，空树的高度定义为-1）。搜索、插入、删除均满足$O(\log N)$的复杂度

> **Proof**：height-balanced的BST始终满足$O(\log N)$的复杂度
>
> 最坏情况是每个节点的左子树和右子树高度都差1。因此，对于高度为h的节点，以之为root的子树的节点数满足如下关系：
>
> $$n_h = n_{h-1} + n_{h-2} + 1 \tag{1}$$
>
> 这是一个斐波那契数列，求和公式为：
>
> $$F_i \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^i \tag{2}$$
>
> 这是一个指数关系。由此，$h = O(\ln N)$

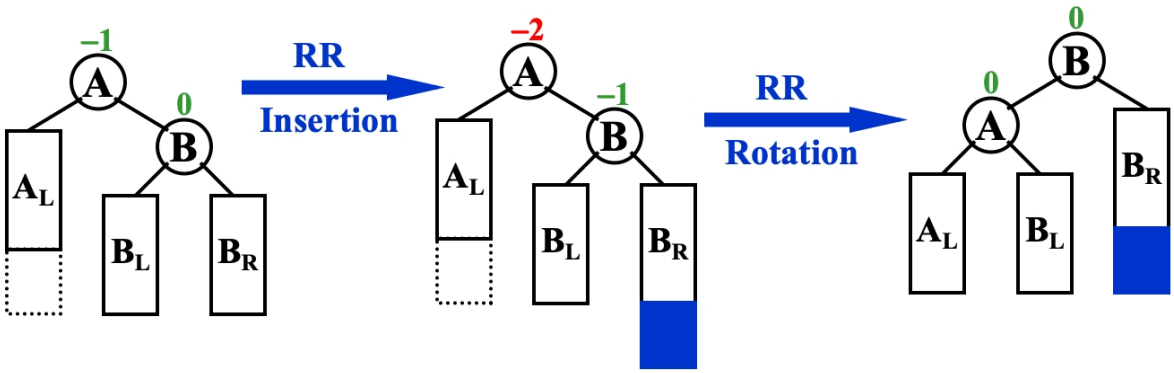
旋转

> 可能会考程序填空题，一般是旋转时的指针操作。做题时建议画图解决，这样更直观一些。
>
> 旋转本身比较简单，代码实现的主要难点在于如何实现树高的记录

当元素的插入导致AVL树不平衡时，需要进行旋转。首先任务是找到最高的不平衡节点，即下面图中的A节点

1. **RR Rotation**

   在右子树的右边插入。转一次，故称为Single Rotation

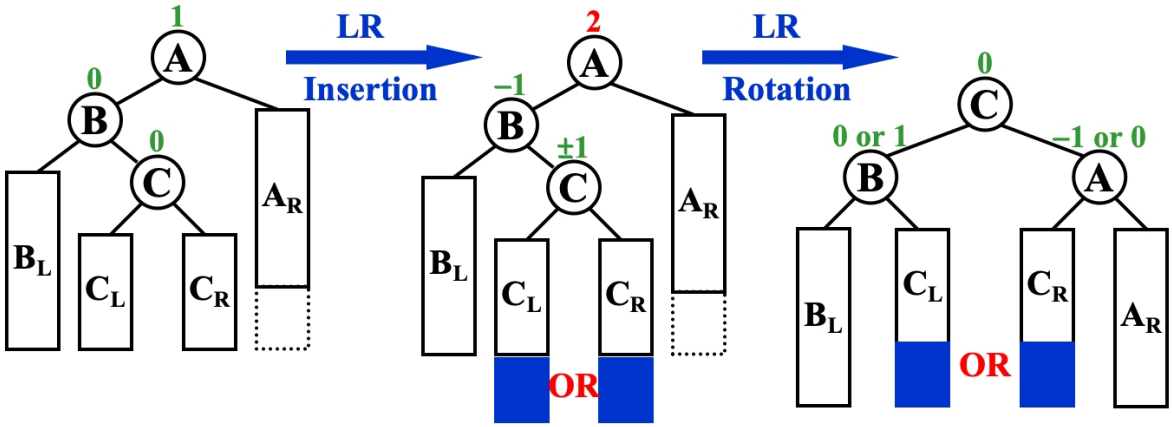


2. **LL Rotation**

   在左子树的左边插入，和RR旋转对称

3. **LR Rotation**

   在左子树的右边插入。需要转两次，先转B和C，再转A和C，故为Double Rotation



4. **RL Rotation**

   和LR旋转对称，也是转两次

考点

AVL树一共是2个考点，一是利用最坏情况的递归式，寻找给定节点数下的最大高度或给定高度下的最少节点；二是考察插入中的旋转

1. **最大高度/最少节点**：If there are 21 nodes in AVL tree, then the maximum depth of the tree is __? The depth of an empty tree is defined to be 0.

   千万不要试图自己画图，这样很容易错的！采用递归的思想解决。最特别的情况就是每棵子树的左子树和右子树高度都差1，即$n_h = n_{h-1} + n_{h-2} + 1$，自底下上递归计算即可。根据$n_0 = 0$和$n_1 = 1$，容易给出0, 1, 2, 4, 7, 12, 20, 33的序列，即高度为6的AVL树至少需要20个节点（显然21能够做到）

2. **插入**：Insert 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree. Which one of the following statements is FALSE?

   一步步插入，观察到不平衡时旋转，注意和Splay树旋转的区别

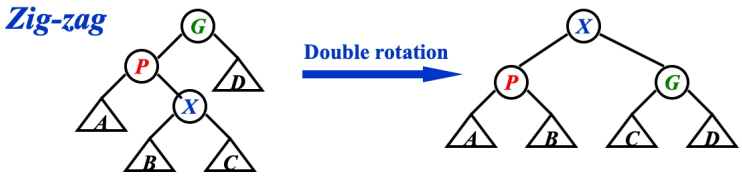
Splay Tree（转到根节点）

放弃$O(\log N)$的追求，放松要求，不再height-balanced。最坏复杂度为$O(M \log N)$（M是连续操作的次数）。每次查询到一个节点时，多次旋转将其变为根节点，方便下次查找

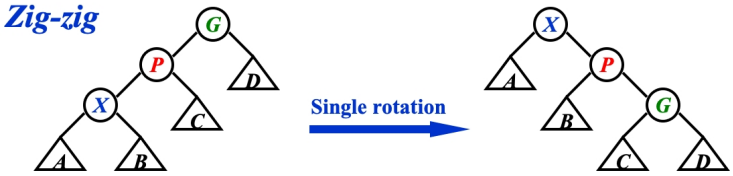
查找

对于非根节点X，其parent为P，grandparent为G

1. **P为根**：旋转X和P
2. **P非根**
   - **Zig-zag**：G、P和X呈「之」字形：X为根，P和G变为两孩子

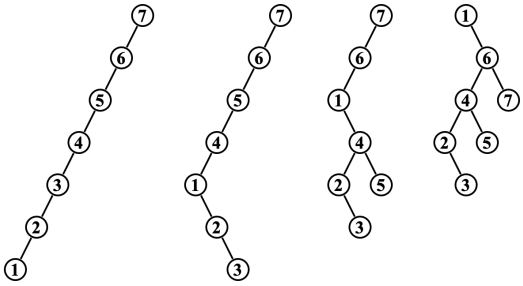


   - **Zig-zig**：G、P和X呈「一」字形：祖孙互换，左右翻转



> Splay树的旋转是一个递归向上的过程，直到将查找到的节点转至根节点

Insert: 1, 2, 3, 4, 5, 6, 7　　**Find: 1**



无论是何种旋转方式，目的都是将X节点转到最高处（方便下次查找）；其子树的位置不需要记忆，可以在旋转后比较大小，填入应有的位置

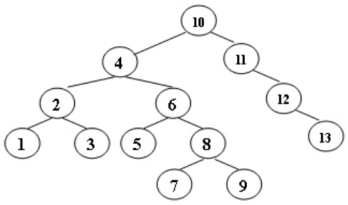
删除

1. **查询要删除的节点**：注意，查找是连带旋转的
2. **直接删掉**：得到其左子树和右子树
3. **在左子树中找最大值**：注意找到最大值以后也要将其旋转到根节点（最大值没有右孩子，不然就有更大的了）
4. **将右子树接到最大值的右孩子上**（现在的最大值也就是左子树的根了）

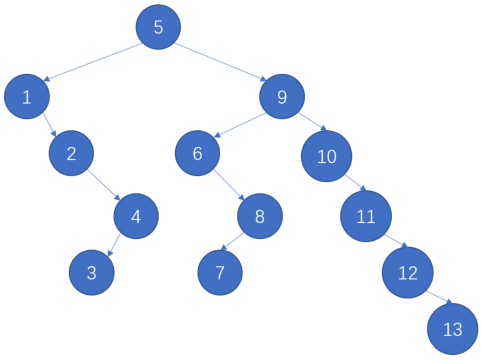
考点

Splay树的考点也是2个，一是旋转操作，二是摊还分析

1. **旋转**：For the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in the following figure, which one of the following statements is FALSE?



记住旋转的目的是将查找的节点转至更高处，直到根节点。本题最终的旋转结果如下：



尽可能不要跳步，宁可多花点时间也不要倒回去重做，否则沉没成本过大

2. **摊还分析**：作为一棵搜索树，Splay树正常来说应采用高度作为势函数，但因为高度比较难以表示，因此采用一种近似的方式：

$$\Phi(T)=\sum_{i\in T}\log S(i)=\sum_{i\in T}\mathrm{Rank}(i)\approx \text{height of the tree} \tag{3}$$

Where $S(i)$ is the number of descendants of i (i inclueded).

AMORTIZED ANALYSIS 摊还分析

基本概念

思路：如果在一组连续的操作中，大部分情况下时间复杂度都很低，个别情况下时间复杂度比较高。此时，就可以将这一组操作看作一个整体进行分析，能将较高时间复杂度那次操作的耗时平摊到其他那些时间复杂度比较低的操作上。

$$\text{worse case bound} \leq \text{amortized bound} \leq \text{average case bound} \tag{4}$$

> 注意，你不能简单地说摊还复杂度好于最坏复杂度、坏于平均复杂度，这几个分析之间不能简单比较

具体实现：Any M consecutive operations starting from an empty data structure take $O[M \cdot f(N)]$ time, then the amortized time bound of this data structure is $O\left[\frac{M \cdot f(N)}{M}\right] = O[f(N)]$.

> **例子**：Dynamic Arrays，数组每次存满都加倍
>
> 从空数组开始，进行n次操作，所需要的时间最多为：
>
> $$n + 2^0 + 2^1 + \cdots + 2^{\log_2(n-1)} < 3n \tag{5}$$
>
> 因此，amortized time bound of dynamic arrays is:
>
> $$O(\frac{3n}{n}) = O(1) \tag{6}$$

摊还分析的三种方法

Aggregate Analysis 总分析法

也就是上面提到的具体实现。n个操作序列最坏情况为$n \cdot O(N)$，则其摊还复杂度为：

$$\frac{n \cdot O(N)}{n} = O(N) \tag{7}$$

Accounting Method 记账法

这是一种更直观的理解方法。每次执行操作时，系统会为某些操作"收费"，将超出当前操作成本的部分积累起来，作为储备以备后续高代价操作的支出。When an operation's amortized cost exceeds its actual cost, we save the difference as credit to pay for later operations whose amortized cost is less than their actual cost

The difference between aggregate analysis and accounting method is that the later one assumes that the amortized costs of the operations may differ from each other. 也就是说，记账法认为不同操作的成本不同，对其进行具体摊销；总分析法更类似于平均法，假设不同操作的摊销成本相同

Potential Method 势函数

思路：Potential Function是一个人为定义的、表示数据结构messy程度的函数。单次操作的成本加上势函数的变化就是该次操作的摊还成本。势函数可以用于解释为何某些操作非常昂贵，但总体上的算法性能依然可以得到很好的保证。

$$\text{amortized cost} = \text{real cost} + k \cdot \Delta\Phi \tag{8}$$

势函数满足$\Phi \geq 0$且$\Phi_{\text{start}} = 0$。因此，我们有：

$$\begin{aligned} \sum \text{amortized cost} &= \sum(\text{real cost} + k \cdot \Delta\Phi) \\ &= \sum \text{real cost} + k \cdot \sum \Delta\Phi \\ &= \sum \text{real cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}}) \\ &\geq \sum \text{real cost} \end{aligned} \tag{9}$$

> **例子**：Two-Stack Queue，一个In栈一个Out栈，入栈至In，出栈至Out。如果出栈时Out为空，则把In栈中的所有元素都入栈到Out中
>
> 定义势函数Φ为In栈的高

当Out栈空且需要出栈时，需要将In栈中所有元素全都出栈，故：

$$\text{real cost} = O(h) \tag{10}$$

势函数变化为：

$$\begin{aligned} \Delta\Phi &= 0 - h \\ &= -h \end{aligned} \tag{11}$$

由此，我们有：

$$\begin{aligned} \text{amortized cost} &= \text{real cost} + k \cdot \Delta\Phi \\ &= O(h) + k \cdot (-h) \\ &= O(1) \end{aligned} \tag{12}$$

The Potential Function of Splay Tree

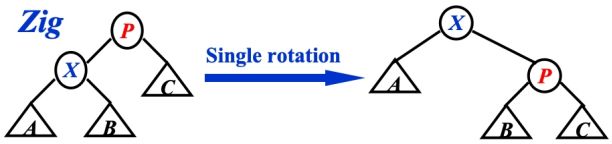
因为在旋转过程中的高度很难确定，所以采用一种近似的方式：

$$\Phi(T) = \sum_{i \in T} \log S(i) = \sum_{i \in T} \text{Rank}(i) \approx \text{height of the tree} \tag{13}$$

Where $S(i)$ is the number of descendants of i (i inclueded).

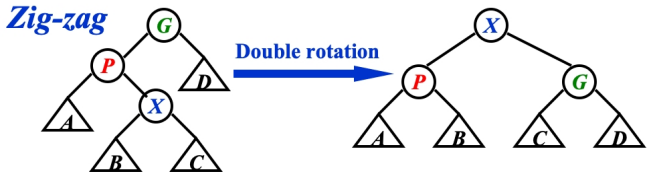
下面分别分析三种旋转方式，其中第i次amortized cost计作$\hat{c}_i$，第i次real cost计作c_i，$\text{Rank}(i)$计作R_i：

- **Zig**



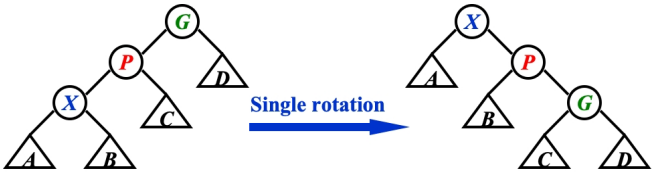
$$\begin{aligned} \hat{c}_i &= 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \\ &\le 1 + R_2(X) - R_1(X) \end{aligned} \tag{14}$$

- **Zig-zag**



$$\begin{aligned} \hat{c}_i &= 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \\ &\le 2[R_2(X) - R_1(X)] \end{aligned} \tag{15}$$

- **Zig-zig**



$$\begin{aligned} \hat{c}_i &= 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \\ &\le 3[R_2(X) - R_1(X)] \end{aligned} \tag{16}$$

假设每次都发生上面三种情况中的最坏情况，则Splay树的摊还分析复杂度为（根为T，查找的节点为X）：

$$3[R(T) - R(X)] + 1 = O(\log N) \tag{17}$$

考点

摊还分析可能会考概念/证明/势函数，其中比较常见的是概念和势函数的选择

1. **概念题**：When doing amortized analysis, which one of the following statements is FALSE?

A. Aggregate analysis shows that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. Then the amortized cost per operation is therefore $T(n)/n$

B. For potential method, a good potential function should always assume its maximum at the start of the sequence

C. For accounting method, when an operation's amortized cost exceeds its actual cost, we save the difference as credit to pay for later operations whose amortized cost is less than their actual cost

D. The difference between aggregate analysis and accounting method is that the later one assumes that the amortized costs of the operations may differ from each other

2. **概念题**：Recall the amortized analysis for Splay Tree and Leftist Heap, from which we can conclude that the amortized cost (time) is never less than the average cost (time). (F)

   摊还和平均不能直接比较

3. **势函数选择**：Consider the following buffer management problem. Initially the buffer size (the number of blocks) is one. Each block can accommodate exactly one item. As soon as a new item arrives, check if there is an available block. If yes, put the item into the block, induced a cost of one. Otherwise, the buffer size is doubled, and then the item is able to put into. Moreover, the old items have to be moved into the new buffer so it costs *k*+1 to make this insertion, where *k* is the number of old items. Clearly, if there are *N* items, the worst-case cost for one insertion can be $\Omega(N)$. To show that the average cost is $O(1)$, let us turn to the amortized analysis. To simplify the problem, assume that the buffer is full after all the *N* items are placed. Which of the following potential functions works?

   A. The number of items currently in the buffer

   B. The opposite number of items currently in the buffer

   C. The number of available blocks currently in the buffer

   D. The opposite number of available blocks in the buffer

   这道题乍一看上去A和D是等效的，但实际上不是。每次item都+1，但buffer满时会直接翻倍，只有后者可以真正达到摊销复杂度的目的，否则每次$\Delta\Phi$都是-1

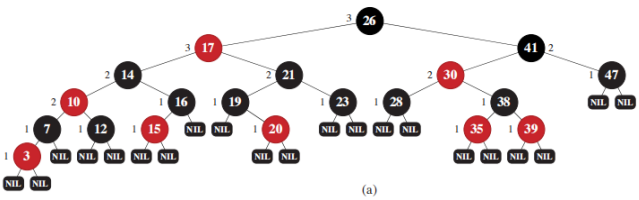
B+树和红黑树

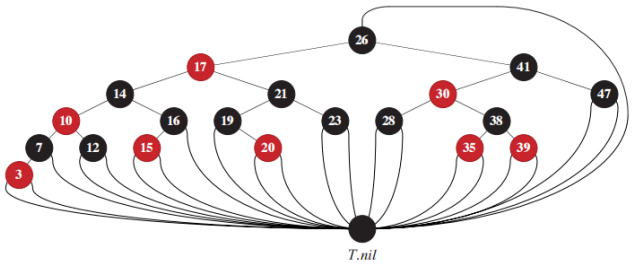
因为AVL和Splay树旋转次数太多，所以考虑进一步放宽条件，减少旋转次数，得到红黑树和b+树

红黑树

定义

1. **根节点为黑**
2. **叶子节点为黑**（定义一个空节点NIL，将其所有叶子节点之下，保证该性质）

> 正常的实现如下（做题时默认树结构如下，题目中给出的树常常不含有NIL节点，需要在脑中补全）
>
> 
>
> (a)
>
> 在代码实现中，可以将所有叶子节点连到同一个NIL上，这样可以大大节省空间



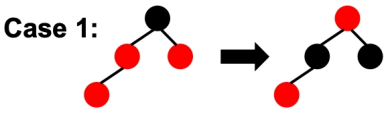
3. **红节点的孩子为黑**（因此红点的数量不可能多于黑点）
4. **任一节点到每一个NIL路径上的黑点数相同**

插入

新插入的节点为红色，这样就不会影响黑点数相同的要求（以下所有操作左右对称时同理），且在旋转变色的操作过程中始终保持路径上黑节点数相同

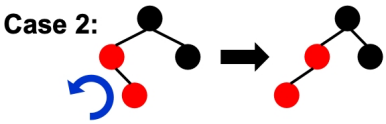
1. **上一层为2红色节点**

   仅改颜色，保证红节点孩子一定为黑。将指针指向祖父节点，递归向上（因为祖父节点的颜色改变了，也需要调整。如果祖父节点为根则不需要调整）



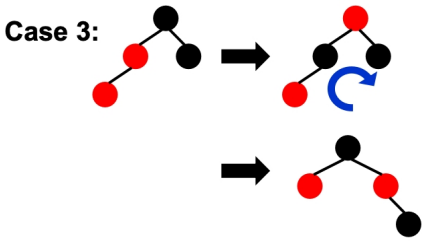
2. **上一层一红一黑，且插入位置在内侧**

   旋转2个红色节点，变为Case 3



3. **上一层一红一黑，且插入位置在外侧**

   先进行Case 1中的变色，然后旋转



删除

根据删除节点位置的不同，先进行替换，保证BST结构

删除叶节点：直接变为NIL

1. **删除节点为红**

   无需额外操作（因为删除红色节点不影响任何性质）

2. **删除节点为黑**

   兄弟为红

   兄弟节点和父节点互换颜色（红色节点的父节点一定为黑），将兄弟节点转到父节点处。旋转后兄弟为黑，父节点为红，按照下面方法继续调整。

   兄弟为黑

   - **兄弟节点存在红色的孩子**

先当作AVL树旋转保证平衡性（如果兄弟有2个孩子或孩子在外侧，就把兄弟转到父节点；如果兄弟只有1个孩子且在内侧，就把孩子转到父节点），再上色（让新的Parent与原先的保持一致，然后逐层向下填色）

> 其实整个操作无非就是涉及4个节点：兄弟节点，兄弟节点的孩子，父节点。而我们的目标就是调整这4个节点的结构，使其满足红黑树的性质
>
> **关于旋转**：其实不需要记情况，想要保证平衡，就一定是把中间大小的节点转到父节点
>
> **关于上色**：这个过程是不需要向上递归的，所以新的父节点应与原先的父节点颜色一致

- **兄弟节点孩子均为黑**

  先兄弟变红（这样就保证所有经过父节点的路径都少了一个黑节点，就相当于是删掉了父节点），然后逐层向上调整（以父节点为新的基准节点，观察其兄弟节点情况，旋转变色；如果父节点为红，则直接变黑；如果父节点为根，不用动，因为父节点左右的黑数是相同的）

删除度为1的节点：将孩子连上来

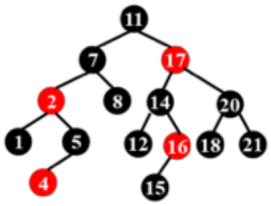
- **删除节点为红**：无需额外操作
- **删除节点为黑**：将连上来的孩子变黑（作为度为1的黑节点，其孩子一定为红，不然会违反黑色节点数相同的性质）

删除度为2的节点：将左子树最大的或右子树最小的替换上来（保留被删除节点的颜色），然后就转化为上面两种情况了

考点

> 经过课程组讨论，之后对于红黑树和b+树的具体操作的考察可能会减少，因为背诵这个确实比较痛苦。但还是有概率会出现在考试中

1. **合法性判断**：下面红黑树是否合法

   不合法。注意到16节点的左右子树中黑节点数不同。警惕只有一个儿子的节点

2. **AVL树的比较**：Is it true that DELETE operation in a RED-BLACK tree of n nodes requires $\Omega(\log n)$ rotations in the worst case? (F)

   红黑树插入和删除的旋转次数都是有限的，AVl树删除可能会导致一路旋转到root

Number of Rotations	AVL	Red-Black Tree
Insertion	≤ 2	≤ 2
Deletion	$O(\log N)$	≤ 3

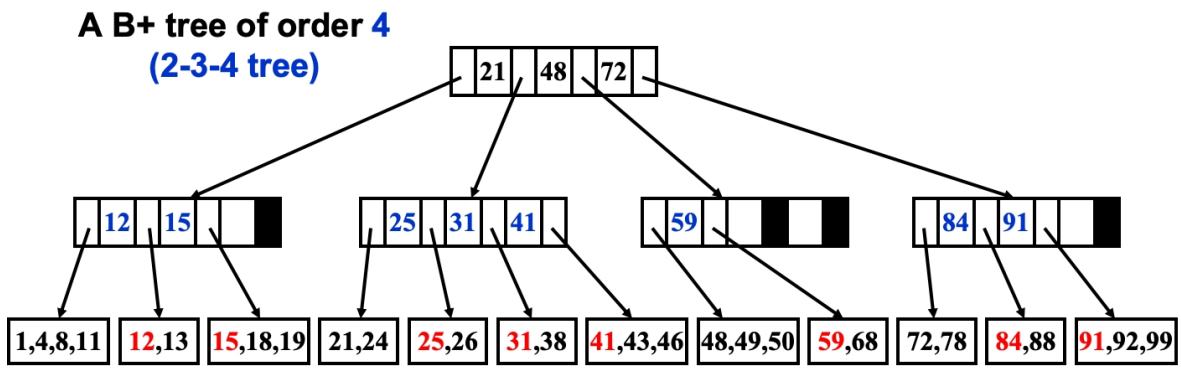
b+树

定义

一个b+ tree of order M具有如下性质：

1. 根含有2至M个孩子，或根为叶子节点（可能会在此设立错误点）
2. 内部节点和叶子节点（非根）的孩子数介于$\lceil M/2 \rceil$和M之间
3. 所有叶节点深度相同（数据储存在叶子节点中）

> 这里给出一个2-3-4[1]树的例子，不难发现其具有如下**性质**：



1. 每个节点都含有M个指针（不要求完全，有一些可以为NULL）
2. 节点中的数字分别为其每个子树（除第一个子树）中最小的那个数字
3. 所有数据均存储在叶子节点

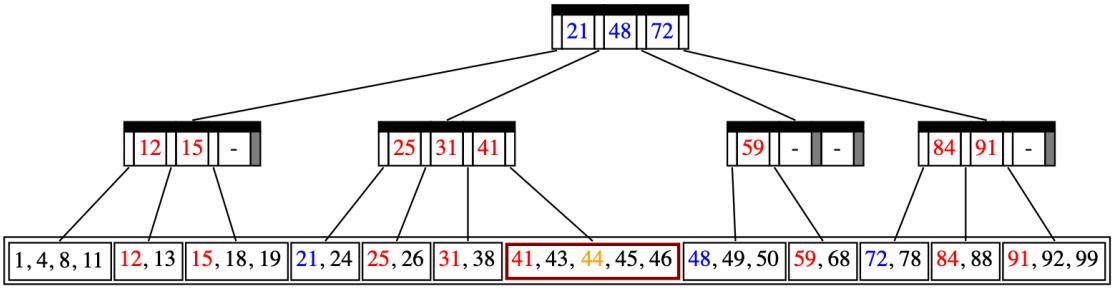
基本操作

> b+树的操作比较自然，只需要感性理解就可以做出题目。但是代码实现较为复杂

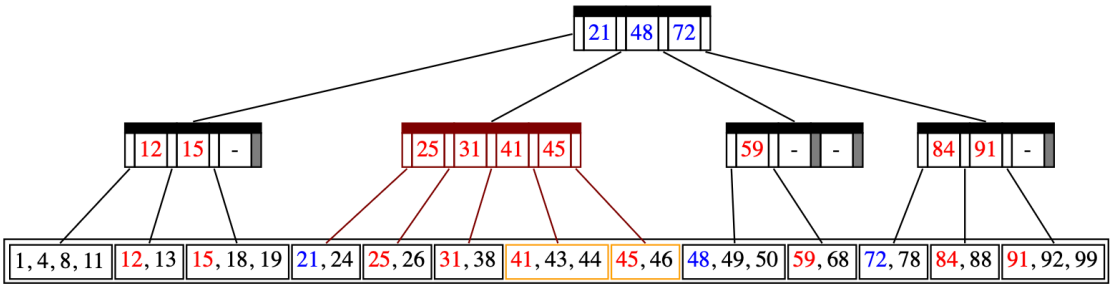
- **插入**

  如果插入的地方未满，直接插入即可；如果已满，向上递归拆分（将节点从中间拆开，父节点增加该中间大小的数；如果父节点也满，就按照同样方法继续向上拆分）

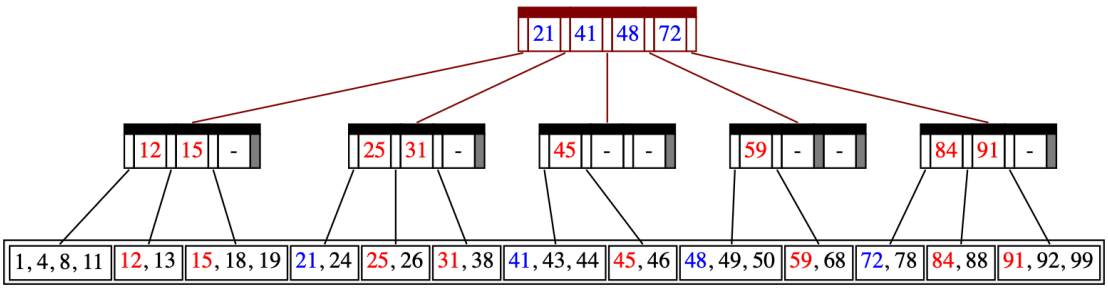
  1. 插入，发现已满

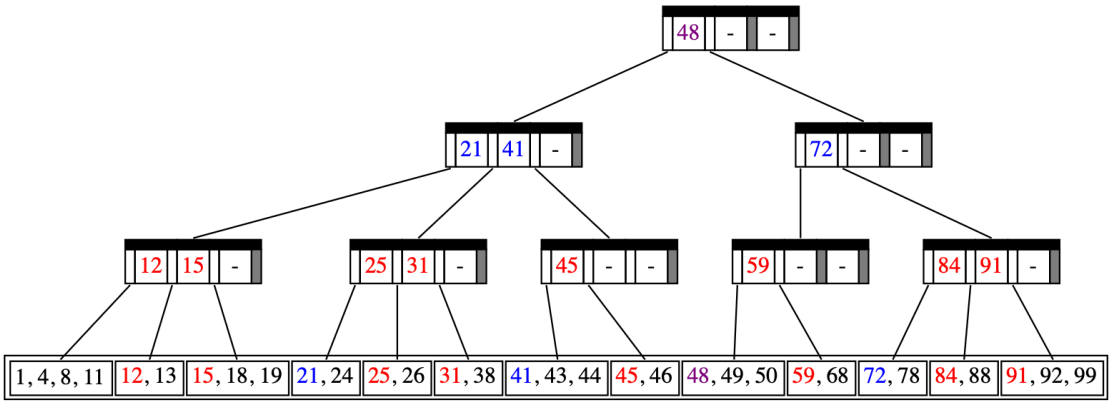


  2. 拆分节点，中间的数（45）加入父节点（因为b+树所有数据均存储在叶子节点，在上移时注意不要将叶子节点中的数误删）



  3. 发现拆分上移后的父节点也满，继续向上拆分（拆分非叶节点时要将中间的数上移并删除，注意叶子节点和非叶节点结构的不同）





- **删除**

  和插入接近。能从兄弟借的就拆一个过来，不够借的话就删掉，将上面的合并下来

考点

> b+树的插入删除操作相比红黑树要好很多，不过期中考也不考，所以先不写了

除了插入和删除，b+树还可能会涉及最多节点问题

1. **最多keys**：A 2-3 tree with 3 nonleaf nodes must have 18 keys at most. (T)

   因为2-3树相当于order为3，一个叶节点最多存3个，无非就是要看最多能有多少叶节点。因为一共只有3个内部节点，所有叶节点都要在同一层，所以只能是1个根，2个孩子，剩下的是叶子节点。因为孩子节点为内部节点，可以有2~3个孩子，所以一共是$2 \times 3 \times 3 = 18$

2. **概念题**：Which of the following statements concerning a B+ tree of order M is TRUE?

   A. the root always has between 2 and M children

   B. not all leaves are at the same depth

   C. leaves and nonleaf nodes have some key values in common

   D. all nonleaf nodes have between $\lceil M/2 \rceil$ and M children

   注意A选项，只有根时也认为其为一棵b+树；D选项还应排除根

INVERTED FILE INDEX 倒排索引

> 倒排索引是一种常用的文本检索技术。正排是以文本为中心的，由文本指向词；而倒排由词指向文本

基本表示方法

给出左下所示的文本，可以写出右边的倒排索引。

- 第一个数字是单词出现的总次数（因为有一些虚词不具有实际意义但出现频率很高，所以记录出现次数来排除这些单词）
- 小括号中前一个数字代表单词所出现的句子，后一个数字代表出现的位置

Doc	Text
1	Gold silver truck
2	Shipment of gold damaged in a fire
3	Delivery of silver arrived in a silver truck
4	Shipment of gold arrived in a truck

No.	Term	Times; Documents Words
1	a	<3; (2;6),(3;6),(4;6)>
2	arrived	<2; (3;4),(4;4)>
3	damaged	<1; (2;4)>
4	delivery	<1; (3;1)>
5	fire	<1; (2;7)>
6	gold	<3; (1;1),(2;3),(4;3)>
7	of	<3; (2;2),(3;2),(4;2)>
8	in	<3; (2;5),(3;5),(4;5)>
9	shipment	<2; (2;1),(4;1)>
10	silver	<2; (1;2),(3;3,7)>
11	truck	<3; (1;3),(3;8),(4;7)>

改进方法

- **Stop Words**：停用词。因为有些词出现次数很多但没有实际意义（比如“a”、“the”等）；因此我们可以给定一个阈值，忽略一些出现次数过高的单词
- **Word Stemming**：词干分析。单词有单复数、时态之分，在建立索引时可以将单词转化为词干，避免将同一个单词的变形误判为不同单词
- **Distributed Indexing**：分布式。因为搜索的数据量非常庞大，所以需要分布式存储。
  1. **Term-partitioned Index**：字典序。按照关键词进行存储。abc开头的单词存在一个地方，def开头的存在另一个地方
  2. **Document-partitioned Index**：文档。按照文档进行存储。文档123存在一个地方，456存在另一个地方
- **Dynamic Indexing**：动态索引。文档内容是不断变化的，所以索引也应该能够根据文档的调整及时调整
- **Compression**：压缩。优化索引结构以减小存储压力
- **Thresholding**：优先查找并显示最相关的文档

性能评估

- 一些非常直观的准则
  - Indexing Speed
  - Searching Speed
  - Expressiveness of Query Language (Ability to express complex information needs / Speed on complex queries)
- **精确度与召回率**

  Precision：衡量检索到信息的质量。检索出的信息中有多少是有效的。检索出的相关信息占所有检索到信息的比例

$$\text{Precision } P = \frac{R_R}{R_R + I_R} \tag{18}$$

  Recall：衡量检索的效果。能检索出多少有效信息。检索出的相关信息占所有相关信息的比例

$$\text{Recall } R = \frac{R_R}{R_R + R_N} \tag{19}$$

	Relevant 相关	Irrelevant 不相关
Retrieved 检索到	R_R	I_R
Not Retrieved 未检索到	R_N	I_N

考点

倒排索引的考察都是概念性质的，其中最为常见的是precision和recall的计算，以及两种distributed index的区别

1. **Distributed Index**： In distributed indexing, document-partitioned strategy is to store on each node all the documents that contain the terms in a certain range. (F)

   这道题的难点主要是英语语法，它的逻辑应该是这样的：store on each node ((all the documents) that contain the terms) in a certain range)，将certain range内每个包含该term的document存在node上。也就是说，节点是按照term分配的，显然是字典序

2. **Retrieval & Relevancy**： When evaluating the performance of data retrieval, it is important to measure the relevancy of the answer set. (F)

   注意retrieved和relevant是两个维度，retrieval其实就是召回率

3. **Range Search**：While accessing a term by hashing in an inverted file index, range searches are expensive. (T)

   这实际上是一个和倒排索引没什么关系的题目，所谓的range search指的是搜索一个范围内的关键词。因为哈希表不是连续分布的，所以对于这种情况只能一个个找，代价很高

4. **精确度与召回率的计算**：没什么好说的，公式背熟

HEAP

Leftist Heap 左斜堆（Npl左大于Npl右）

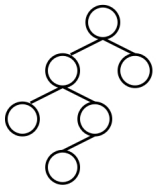
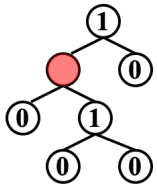
左斜堆相比普通堆的改进在于支持快速的堆合并

定义与性质

Npl：Null path length of Node X，节点X到没有2个孩子节点的最短路径长度。$\mathrm{Npl(NULL)}=-1$（叶节点的Npl是0，因此空节点的NPL就只能是-1了）

$$\mathrm{Npl(X)}=\min\{\mathrm{Npl(C)}+1\}\text{ for all C as children of X} \tag{20}$$

Leftist Heap：对于堆中的每个节点，它左子节点的Npl大于等于右子节点的Npl（注意这个概念啊，不要搞混谢谢）

> **例**：下面这两个哪个是左斜堆?
>
> 
> 
>
>
> 左边是，右边不是。注意到红色节点右子树的Npl大于左子树

- **Declaration**：存储的数据、左子树、右子树、Npl

```
struct TreeNode
{
  ElementType Element;
  PriorityQueue Left;
  PriorityQueue Right;
  int Npl;
};
```

- **一些性质**

  1. 递推：一个节点的Npl等于其右孩子的Npl加上1
  2. 如果一个节点的Npl为r，则以其为根的树至少是一个$r+1$层的完美二叉树（注意Npl是从0开始的）

基本操作

1. **merge**

> 插入是合并的一种特殊形式（将插入的节点看作一棵只有一个节点的左偏树），删除也是基于merge实现的。所以最基本的操作其实只有merge

- **递归**

  先比较两个待合并的子树根节点的Element，选择较小的作为根节点（以小根堆为例，大根堆相反）。其左子树依旧为左子树，右子树和另一个待合并子树合并，成为新的右子树

  合并完成后，检查是否满足`Root->Left->Npl >= Root->Right->Npl`，如果不满足则交换左右子树位置（因为是递归算法，所以这个交换相当于是自底向上的）

```
LeftistHeapNode * merge(LeftistHeapNode * x, LeftistHeapNode * y) {

    if (x == NULL) return y;
    if (y == NULL) return x;

    if (x->Element > y->Element) {
        swap(x, y);
    } // 选择小的为新的根节点，因为最后要return x，所以这里采用交换

    x->Right = merge(x->Right, y);

    if (x->Left->Npl == NULL || x->Left->Npl < x->Right->Npl) {
        swap(x->Left, x->Right);
    }

    x->Npl = x->Right->Npl + 1; // 更新Npl

    return x;
}
```

- **迭代**

> 显然没有人会在考试的时候用脑子跑递归，所以迭代的方法更需要记忆（但其实和递归的方法思路很像）

选择根比较小的作为结果的根。然后将右子树拿掉，将右子树的根与另一个树比较，选择较小的作为右子树的根，然后把右子树的右子树拿掉……（是不是和递归很像？）

代码实现的时候别忘了迭代算法最后还需要自底向上调整Npl，并交换左右子树，以保证左子树Npl大于右子树

```
LeftistHeapNode * merge(LeftistHeapNode * x, LeftistHeapNode * y) {
    LeftistHeapNode * tx = x, * ty = y; // 两棵树还没有合并进去的子树
    LeftistHeapNode * res = NULL, * cur = NULL; // `res`是最终返回的根节点，`cur`是当前merge的节点

    while (tx != NULL && ty != NULL) {

        if (tx->Element > ty->Element) {
            swap(tx, ty);
        } // 选择小的作根，保证小根堆性质

        if (res == NULL) { // 仅在第一次循环有效，初始化res和cur
            res = tx;
            cur = tx;
        } else { // 第一次循环之后有效
            cur->Right = tx; // tx是两棵待合并子树合并后的根
            cur = cur->Right; // 现在相当于是需要合并cur->Right和ty
        }
        tx = tx->Right; // 所以新的tx就是cur->Right（tx->Right）
    }

    if (ty != NULL) {
      cur->Right = ty;
    } // 上面的循环结束后，tx已经整个连上去了，还剩下ty


    adjust(res); // 此处不仅要调整Npl，还要自底向上保证每个节点的左Npl均大于右Npl
```

```
        return res;
    }
```

2. **单点删除**

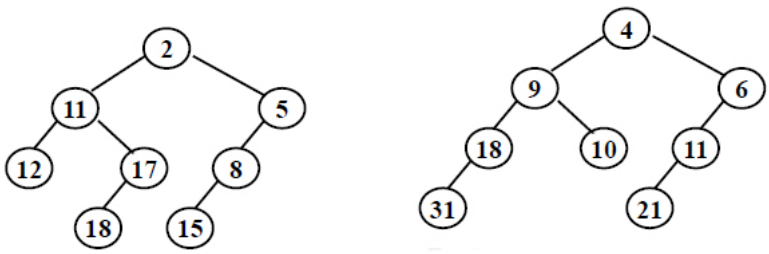
   直接merge被删除的两个节点的子节点。注意完成后要向上更新Npl

```
LeftistHeapNode * del(LeftistHeapNode * cur, ElementType x) {

    if (cur->Element == x) { // 如果找到，直接merge然后return
        return merge(cur->Left, cur->Right);
    } else {
        if (cur->Element > x) return cur; // 如果删除的比根还要小，说明没找到

        if (cur->Left != NULL) del(cur->Left, x); // 在左子树里面寻找
        if (cur->Right != NULL) del(cur->Right, x); // 在右子树里面寻找

        adjust(cur); // 调整Npl
    }
}
```

考点

因为左斜堆的知识难度不大，所以每个方面都有可能会考，在作业中，概念、merge、删除和代码都有涉及

1. **Merge**：Merge the two leftist heaps in the following figure. Which one of the following statements is FALSE?



   A. 2 is the root with 11 being its right child

   B. the depths of 9 and 12 are the same

   C. 21 is the deepest node with 11 being its parent

   D. the null path length of 4 is less than that of 2

   用迭代的思路做，别忘了自下而上调整结构即可

2. **单点插入**：We can perform BuildHeap for leftist heaps by considering each element as a one-node leftist heap, placing all these heaps on a queue, and performing the following step: Until only one heap is on the queue, dequeue two heaps, merge them, and enqueue the result. Which one of the following statements is FALSE?

   A. in the k-th run, $\lceil N/2k \rceil$ leftist heaps are formed, each contains $2k$ nodes

   B. the worst case is when $N = 2K$ for some integer K

   C. the time complexity $T(N) = O(\frac{N}{2}\log 2^0 + \frac{N}{2^2}\log 2^1 + \frac{N}{2^3}\log 2^2 + \cdots + \frac{N}{2^K}\log 2^{K-1})$ for some integer K so that $N = 2^K$

   D. the worst case time complexity of this algorithm is $\Theta(N \log N)$

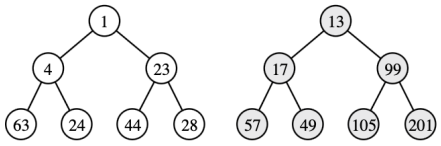
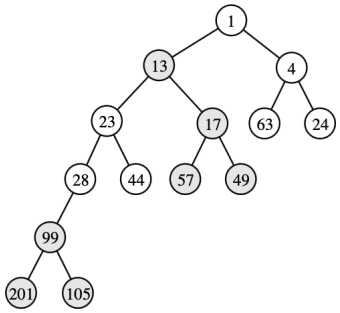
   C是对的，把C选项加起来，发现D不对

3. **代码**：递归的代码考察的可能性更大，需要注意

Skew Heap（无条件交换）

> 斜堆也是一个大根堆（小根堆）

斜堆在左偏堆的基础上，省去左斜的判断，无条件交换左右子树。这样就不需要回溯交换，从而实现了完全的自上而下。同时，skew heap还不需要存储npl，因此其空间消耗一定小于leftist heap（注意skew heap和leftist heap的插入都是一种特殊的merge，不是到底的）

> **例子**：其实只要理解了左斜堆的merge操作，skew heap的很好理解
>
> 
>
>
> 上面两个Skew Heap合并后为：
>
> 
>
>
> 在手搓的时候，注意先插入再互换，不要两步合一，一不小心就糊涂了
>
> 此外，最下面的105和201也互换，一定要递归到底！merge空树的特殊情况不作额外处理，也需要交换

合理性

可以这样理解，在递归中，每次都是保留左子树，右子树进行merge操作，这样会导致右子树生长。因此在合并完成后将右子树转为左子树可以保证左斜性质

> **例题**：The result of inserting keys from 1 to $2^k - 1$ for any $k > 4$ in order into an initially empty skew heap is always a full binary tree.
>
> True. 采用递归的思想。对于一个满二叉树，插入一个数，它比树中任何一个数都要大。想象它在插入过程中一路右走，但是每次都会被甩到左边（因为无条件交换），最终它会出现在该有的位置。考试遇到这种题别手搓了，反而容易出岔子，直接选T就可以

摊还分析

> 采用potential method对斜堆的merge操作进行摊还分析

1. 为给出势能函数，我们还需要引入两个概念
   - **Heavy Node**：A node p is heavy if the number of descendants of p's right subtree is at least half of the number of descendants of p。也就是说，p的右子树比较重
   - **Light Node**：非heavy tree

   > 由此，我们能给出如下性质：
   >
   > 1. 如果一个节点是heavy node，其右子树发生合并（并翻转），则其一定变为light node
   > 2. 如果一个节点是light node，其右子树发生合并（并翻转），则其可能变为heavy node
   > 3. 合并过程中，如果一个节点发生heavy和light间的变化，那它原来一定在堆的最右侧路径上（因为merge就是自上而下merge右子树）

2. **势能函数**：堆中heavy node的个数

$$\Phi = \text{number of heavy node in } Heap \tag{21}$$

   根据这个势能函数，我们可以得到合并操作的复杂度为：

$$\hat{c} = c + \Phi(H_{\text{merged}}) - \Phi(H_x) - \Phi(H_y) \tag{22}$$

其中c是merge的复杂度，$\Phi(H_{\text{merged}})$为合并后的堆的势能，$\Phi(H_x)$和$\Phi(H_y)$为合并前两个堆的势能。因为合并过程中只有最右侧路径上的节点会发生heavy和light的变化，所以我们只需要关注最右侧路径即可

假设l_x为H_x最右侧路径上light node的数量，h_x为H_x最右侧路径heavy node的数量，h_x^0为H_x所有不在最右侧路径上heavy node的数量（即H_x上heavy node的总和为$h_x + h_x^0$），我们有：

$$\begin{cases} c = l_x + h_x + l_y + h_y & \text{（最右路径上每个点都相当于一棵需要merge的子树的根）} \\ \Phi(H_{\text{merged}}) \le l_x + h_x^0 + l_y + h_y^0 & \text{（因为heavy node合并后一定会变为light node，lignt node合并后可能变为} \\ \Phi(H_x) = h_x + h_x^0 & \\ \Phi(X_y) = h_y + h_y^0 & \end{cases}$$

$$\begin{aligned} \Rightarrow \hat{c} &\le 2(l_x + l_y) \\ &= O(\log N) \end{aligned} \tag{24}$$

考点

除了skew heap本身的性质，有时还会和leftist heap比较考察

1. **顺序插入**：The result of inserting keys 1 to $2^k - 1$ for any $k > 4$ in order into an initially empty skew heap is always a full binary tree. (T)

   从小到大插入，skew heap会成为一棵full binary tree，想象一下插入过程即可

2. **merge**：考察skew heap的常规操作，基本功

3. **和左斜堆比较**：Which one of the following statements is FALSE about skew heaps?

   A. Comparing to leftist heaps, skew heaps are always more efficient in space

   B. Skew heaps have $O(\log N)$ worst-case cost for merging

   C. Skew heaps have $O(\log N)$ amortized cost per operation

   D. Skew heaps do not need to maintain the null path length of any node

   A选项是对的是因为leftist heap需要对Npl进行存储，需要消耗更多的空间

4. **和左斜堆比较**：If a leftist heap can be implemented recursively, so can its counterpart skew heap. (F)

   Leftist heap递归代价小于skewed heap，后者的right path的长度可以是$O(n)$量级，会导致递归深度太大，造成栈溢出，但这种情况leftist heap依旧可以运行

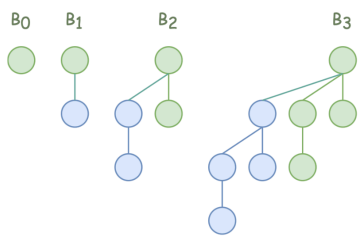
BINOMIAL QUEUE 二项队列

定义

> 二项队列本质上是二项树的集合，所以我们需要先介绍二项队列

Binomial Tree 二项树

定义：二项树满足小（大）根堆的性质，即parent节点小于（大于）children。所有k阶二项树都是同构的，由2个$k-1$阶二项树合并得到



合并：直接将一棵树接在另一棵树的根下，注意堆性质的满足（所以显然二项树不是二叉树）

性质

1. k阶二项树的结构唯一确定，其根有k个孩子（废话）
2. B_k的第d层一共有C_k^d个节点（数归证明）

Binomial Queue 二项队列

因为二项树的结构限制较为严格，只能有2^k个节点，直接用其来存储数据较为浪费。但是，如果我们参考二进制数的表达形式，将多个不同大小的二项树进行组合（$x = k_0 \cdot 2^0 + k_1 \cdot 2^1 + k_2 \cdot 2^2 + \ldots, k_i = 0 \text{ or } 1$），就可以实现任意量的数据存储了。因此二项队列中$k$阶的二项树要么有1个，要么没有

所以说，二项队列其实就是二项树的集合，其队首即为所有二项树中最小（大）的那个

操作

队列合并

合并2个二项队列，实际上就是合并2个二项树的集合，就像是二进制加法一样自下而上合并

> 单点插入也看作队列合并处理
>
> 注意insertion操作的摊还复杂度是常数级别的，可以想象一下二进制数数的方式。而其他操作（merge、find min、delete min）都是$\log$级别的

查询队首

队首为所有二项树的根中最小的那个，查找的复杂度为$O(\log N)$。有些时候会有一个额外的指针指向最小根，此时复杂度为$O(1)$

队首出队

观察二项树，我们发现一棵二项树的根下所连的每棵子树都是一棵二项树。所以删除队首只需要找到队首将其删除，然后merge剩下的所有二项树即可

考点

二项队列的操作比较简单，值得注意的也就是复杂度相关的问题

1. **复杂度**：For a binomial queue, ____ takes a constant time on average.

   A. merging

   B. find-max

   C. delete-min

   D. insertion

   在默认情况下，只有插入是常数复杂度的（摊还、平均，换句话说，二进制加法的摊还复杂度是常数级别的），其余均为$O(\log N)$
2. **代码**：二项队列可能会考察代码填空，但总体难度不大，就不占篇幅了

BACKTRACING 回溯

Eight Queens 八皇后

基础问题

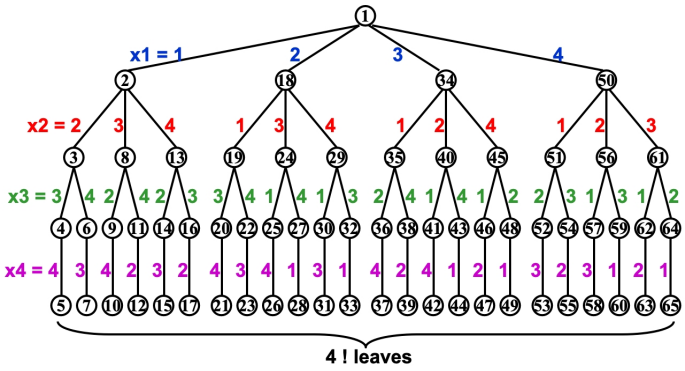
1. **问题描述**

Find a placement of 8 queens on an 8×8 chessboard such that no two queens attack. (Two queens are said to attack iff they are in the same row, column, diagonal, or antidiagonal of the chessboard)

2. **初步分析**

> 该问题主要有2个约束条件：不在同一行/列、不在对角线。如果采用枚举的方法，初步判断是阶乘级别的复杂度（只考虑第一个约束条件）

我们将问题简化为四皇后，可以给出如下枚举树（仅考虑第一个约束条件）。其中每个叶子节点都对应一种可能解：

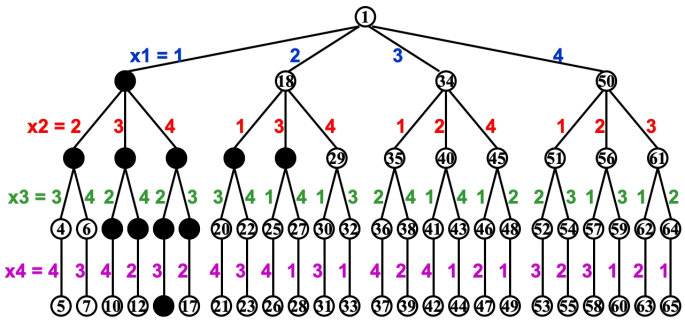


> 注意这棵树只是为了方便理解才建立的，实际代码实现中不需要建立额外的数据结构，DFS很简单的，没有这么麻烦🤣

3. **回溯**

> 显然，上面并非每一条路径都是可行的，因为还未考虑第二条约束条件。因此，我们可以用DFS进行回溯（当不满足对角线条件时，退回到上一个节点）

从根开始向下探索。如果某个节点的所有孩子都无法走通，说明该节点不存在可能解；将该节点标黑，退回到上一节点，探索其还未标黑的兄弟节点（注意标黑只是一个形象的理解，不需要DFS上的特殊实现，因为其递归过程天然存在这种效果）



应用：The Turnpike Reconstruction Problem 收费站重建问题

1. **问题描述**

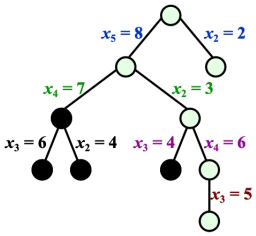
Given N points on the x-axis with coordinates $x_1 < x_2 < \cdots < x_N$. Assume that $x_1 = 0$. There are $\frac{N(N-1)}{2}$ distances between every pair of points.

在一条直公路上一共有N个收费站，任意2个收费站间的距离的集合（可重集合）共有$\frac{N(N-1)}{2}$个元素；现在给出这个可重集合D，需要设计算法求出点的坐标

2. **思路**

先确定$x_1 = 0$，显然$x_N = \max\{D\}$，这样就把头和尾先确定了。然后我们从D中从大到小取出元素，则其一定追加在x_1的右侧（x_{N-1}）或x_N的左侧（x_2）。画出搜索树，发现和八皇后一样，DFS求解

> **例子**：$D = (1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10)$
>
> 1. 计算得$N = 6$
> 2. $x_1 = 0$，$x_6 = 10$
> 3. DFS回溯



$\alpha - \beta$ Pruning 剪枝

Tic-tac-toe 井字棋

1. **问题描述**

   3×3的棋盘，连成横、竖、对角线即为胜利

2. **问题分析**

   > 注意区分layout和game，前者指的是给一个棋盘，在上面摆棋子，考虑可能的摆放形式，不考虑摆放顺序；后者指的是一直下棋，直到将棋盘填满，一共有多少种可能的过程

   - 每个格子都可以是X、O或空格，所以共有3^9 possible board layouts
   - 每一步棋都可以选择棋盘上任意空位处，所以共有$9!$ possible games

3. **Minimax Strategy**

   我们采用一个方程衡量每个位置的goodness：

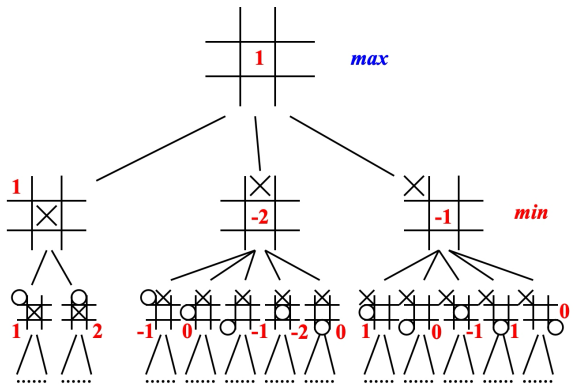
   $$f(P) = W_{\text{computer}} - W_{\text{human}} \tag{25}$$

   其中W为将棋子放在位置P后可能的获胜情况数（注意这只是一种衡量方式，现实中往往不会采用如此简单的函数）

   > 比如对于下面情况，想要红圈获胜，只能是四条边界线；想要蓝叉获胜，可以是两条对角线、下左右三条边界线以及中间一条横线。因此$f(P) = 6 - 4 = 2$
   >
   > 

   其中human的目标就是最小化$f(P)$，为min方；而computer的目标是最大化$f(P)$，为max方

   我们从max方的视角出发，考虑到对方一定会按照$f(P)$最小化的原则选择自己的行为，而我们的任务就是在假设对方作出最小化选择的基础上最大化自己的选择。即，我们每次决策时，其实是在有限个选择中选择最大的那个；此处所谓的最大不是简单的将棋子摆放在此处后得到的$f(P)$最大，而是选择此处后，对方基于最小化原则让此处最小后，从所有最小化后的选择中选出最大的那个；如此便实现了递归。下面给出一个例子

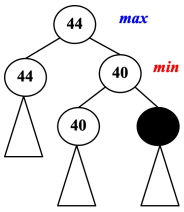
   > 在第一层（max层）决策处，计算机的目标是选择下一层（min层）中最大的；然后进入到min层，此时min方作决策，其目标为选择下一层（max层中最小的）；依此类推
   >
   > 
   >
   >
   > 按照这个过程，我们发现不将所有节点都算出来就无法得到正确答案，于是复杂度来到了糟糕的阶乘级别。因此我们需要引入一定优化方法，剪枝就是其中一种

   这个过程可能看上去很复杂，但用递归的视角来看，每一层的决策都只需要考虑低一层的结果即可

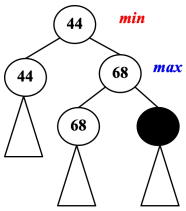
4. $\alpha - \beta$ **Pruning**

   剪枝就是将不可能选择的节点裁剪掉，减少无效的进一步递归带来的负担。体现在决策树上就是某一个节点标黑，以其为根的子树均无需搜索。具体有如下两种情况：

   - max节点的兄弟节点比其父节点的兄弟节点小（因为其兄弟节点决定了其父节点不及40，而父节点的兄弟节点为44，因此max层一定不会选择此路径）

   - min节点的兄弟节点比其父节点的兄弟节点大（其兄弟节点决定了父节点不低于68，而其父节点的兄弟节点为44，因此min层一定不会选择此路径）

   > 注意，遇到which node is the first to be pruned问题，被剪掉的一定是右孩子。因为遍历时先左后右，如果左孩子导致父节点不会被选择，那么右孩子的探索就会被取消，真正被剪掉的是右孩子

考点

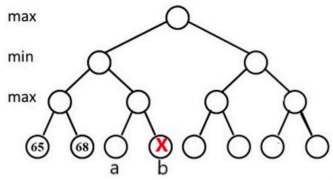
回溯的考点主要是以井字棋为例的剪枝，判断题中也可能会出现概念的考察

1. **概念**：What makes the time complexity analysis of a backtracking algorithm very difficult is that the sizes of solution spaces may vary. (F)

   导致回溯算法复杂度分析难度大的原因有很多，比如剪枝的出现情况等

2. **剪枝**：Given the following game tree, if node b is pruned with $\alpha - \beta$ pruning algorithm, which

   of the following statements about the value of node a is correct?

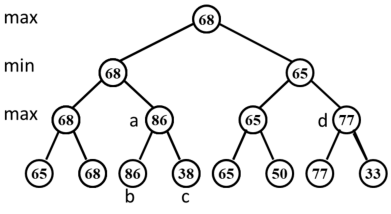
   A. greater than 68

   B. less than 65

   C. less than 68

   D. greater than 65

3. **剪枝**：Given the following game tree, which node is the first one to be pruned with α-β pruning algorithm? (C)

DIVIDE & CONQUER 分治

Introduction

过程

1. **Divid**：将问题拆分为几个简单的子问题
2. **Conquer**：将子问题递归解决
3. **Combine**：Combine the solutions to the sub-problems into the solution for original problem. 将子问题的解合并到原问题中

Example: Maximum Sequence of Sum

问题：给出一个数字序列（有正有负），从中找出一个连续子序列，使得该子序列的总和最大

分治算法：将1个序列对半分为2个，比较左边最大的子序列、右边最大的子序列、横跨两边最大的子序列

复杂度：采用数归。对于每次操作，所需要的时间$T(N)$为减半后两边需要的时间$T(N/2)$和横跨中间需要的时间cN的加和。因此我们有如下递推式：

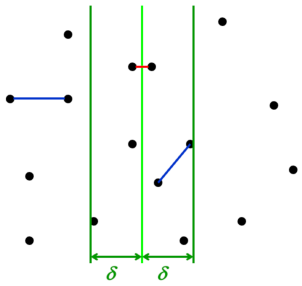
$$\begin{aligned} T(N) &= 2T(N/2) + cN \\ &= O(N \log N) \end{aligned} \tag{26}$$

Closest Pair of Points

在平面上有一些点，找到其中距离最近的2个

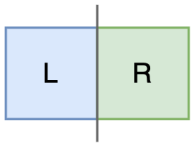
分治算法

1. 在平面中画一条线，将平面分为2部分。只需找到左右部分中距离最近的（递归），和横跨中间的进行比较
2. 但是横跨中间点的组合还是很多，所以需要限制范围。我们先找到两边距离最近的，并取其中更小的，记为δ；由此我们知道跨越中间情况想要更小，一定在分界线$\pm\delta$的范围内

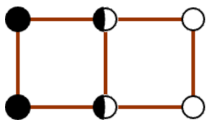


3. 这样虽然缩小了范围，但还是需要选择范围内的所有组合进行计算，最坏复杂度依旧是平方级别的，仍有优化空间
4. 找到可能的更小距离本质上就是从分界线左边取出一个点，并找出分界线右边距其最近的点。假设我们从下往上找（这样就无需考虑位于当前点下方的可能性，因为已经探索过了）

   想要找到距离某个点最近的点，其纵坐标一定在上方δ范围内；由此，可选点被限制在右侧的一个$\delta \times \delta$（或者说$2\delta \times \delta$的矩形内，因为是自下而上寻找的，搜索时不区分左右）的范围内



5. 同时，因为单侧的点距离不少于δ，其实在该范围内最多只能放下6个点（PPT上认为是8个，更加宽松，但反正是常数个），所以对于给定的某个点，跨边界查找的复杂度是常数级的，因此遍历所有点复杂度最坏也是$O(N)$。这样，across的复杂度降为了$O(N)$；当然，总体的复杂度还是需要采用递推式进行计算



复杂度分析

在上面的算法分析中，我们有：

$$\begin{aligned} T(N) &= \underbrace{2T(N/2)}_{\text{divide}} + \underbrace{O(N)}_{\text{across}} \\ &= O(N \log N) \end{aligned} \tag{27}$$

递推式求复杂度

> 分治的考点基本上就是递推式的处理

递推式的一般情况是

$$T(N) = aT(N/b) + f(N) \tag{28}$$

我们有如下一些方法对其进行求解：

Substitution 代换法

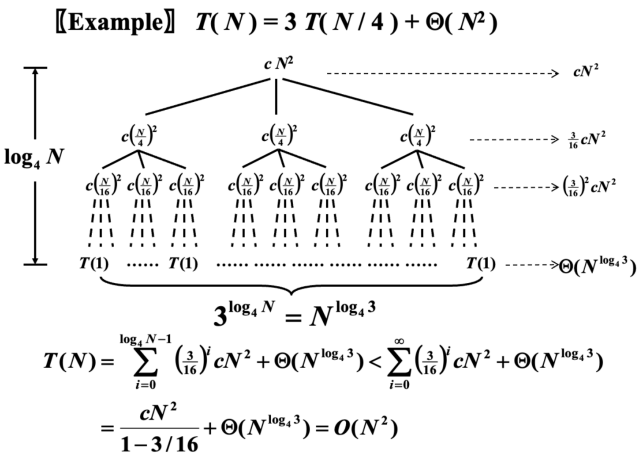
先猜出答案，然后代入（考试的时候反正没有填空题，肯定直接代入啊。别想着做出来，做出来也是错的）

> 对于$T(N) = 2T(N/2) + N$，大胆猜测答案为$O(N \log N)$，代入：
>
> $$\begin{aligned} T(m) &= 2T(m/2) + m \\ &= 2 \cdot O\left(\frac{m}{2} \log \frac{m}{2}\right) + m \\ &= O(m \log m) \end{aligned} \tag{29}$$

Recursion Tree 递归树

相当于是强行展开的可视化表现

> 逐层展开，然后求和；其实就是数学上递推的求和



Master 主方法

> 感觉被这个公式还是比较痛苦的，考试中将会以何种形式出现还有待验证。就作业来看，似乎Form 3考察的可能性更高。不过，总体来说，更推荐的方法还是将可能的答案带进去验证（一般有2个选项一看就不对，带入验证2个选项的速度绝对比计算要快）

本质上是观察combine的代价$f(N)$和conquer的代价$T(N/b)$哪个更大

Form 1：比较$f(N)$与$N^{\log_b a}$。本质上是a相较于b是否够得上$f(N)$的增长

1. $f(N) = O\left(N^{(\log_b a) - \varepsilon}\right)$，则$T(N) = \Theta\left(N^{\log_b a}\right)$
2. $f(N) = \Theta(N^{\log_b a})$，则$T(N) = \Theta(N^{\log_b a} \log N)$

3. $f(N)=\Omega\left(N^{(\log_b a)+\varepsilon}\right)$，且$af(N/b)<cf(N)$ for some constant $c<1$ and all sufficiently large N（这个条件称作regularity condition），则$T(N)=\Theta(f(N))$

Form 2：比较$af(N/b)$与$f(N)$。这种方法看上去形式上要更为简单一些，当然实际上和form 1是一样的

1. 如果$af(N/b)=\kappa f(N)$ for fixed $k<1$，则$T(N)=\Theta(f(N))$
2. 如果$af(N/b)=Kf(N)$ for fixed $K>1$，则$T(N)=\Theta\left(N^{\log_b a}\right)$
3. 如果$af(N/b)=f(N)$，则$T(N)=\Theta\left(f(N)\log_b N\right)$

Form 3：对于$T(N)=aT(N/b)+\Theta\left(N^k\log^p N\right)$的特殊形式，约束条件比较精准，基本上遇到这种形式只能用此方法

$$T(N)=\begin{cases}O\left(N^{\log_b a}\right) & \text{if } a>b^k \quad \text{相当于}\log_b a>k\\ O\left(N^k\log^{p+1}N\right) & \text{if } a=b^k\\ O\left(N^k\log^p N\right) & \text{if } a<b^k\end{cases} \tag{30}$$

考点

1. **解递推式**：When solving a problem with input size N by divide and conquer, if at each stage the problem is divided into 8 sub-problems of equal size $N/3$, and the conquer step takes $O(N^2\log N)$ to form the solution from the sub-solutions, then the overall time complexity is __.

   A. $O(N^2\log N)$

   B. $O(N^2\log^2 N)$

   C. $O(N^3\log N)$

   D. $O(N^{\log 8/\log 3})$

   先写出递推式：

   $$T(N)=T(N/3)+O(N^2\log N) \tag{31}$$

   把A代入，得：

   $$N^2\log N=\frac{8}{9}N^2\log N-\frac{8\log 3}{9}N^2+O(N^2\log N) \tag{32}$$

   令：

   $$O(N^2\log N)=\frac{1}{9}N^2\log N+\frac{8\log 3}{9}N^2 \tag{33}$$

   即可。这样，B和C就不用算了，因为它们都比A宽松。只需要检查D，但D都小于N^2了，显然不对

2. **比较复杂度**：

3. To solve a problem with input size N by divide and conquer algorithm, among the following methods, __ is the worst.

   A. divide into 2 sub-problems of equal complexity $N/3$ and conquer in $O(N)$

   B. divide into 2 sub-problems of equal complexity $N/3$ and conquer in $O(N\log N)$

   C. divide into 3 sub-problems of equal complexity $N/2$ and conquer in $O(N)$

   D. divide into 3 sub-problems of equal complexity $N/3$ and conquer in $O(N\log N)$

   A和C比，B和D比，发现答案一定在CD中产生。D比较好猜，将$N\log N$代入D，发现成立。C想要小于D，只能是N，显然不行（除非$O(N)$是负的，别这么蠢），那就只能假定C比D大了。比$N\log N$大的话，猜想是N^a的形式，代入，消高次项，发现$a>1$，得证

DYNAMIC PROGRAMMING 动态规划

> 期末考试会有一道很难的动态规划题，大多数同学都做不出来，所以还是尽可能减小试错成本，保证前面的题少错才是关键

> 动态规划采用的是类似迭代的思路，因为问题的求解基于之前记住的子问题的答案，所以迭代的顺序非常重要。客观题中可能会出现循环顺序的考察

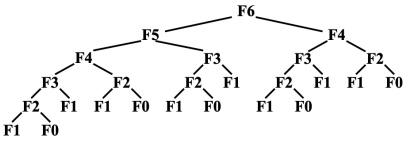
Introduction

引入与介绍

假设我们要计算斐波那契函数，一般的递归方法如下：

```
int Fib(int N) {
  if (N <= 1)
    return 1;
  else
    return Fib(N - 1) + Fib(N - 2);
}
```

画出上面算法的递归树：



我们发现F0～F4都被重复计算了好多次。如果算完一次后就可以记住那时的值，就可以避免递归中的重复计算。这也就是动态规划的核心思想「Those who cannt remember the past are condemned to repeat it.」

动态规划的本质就是将总问题拆分为子问题，同时记住子问题的答案以避免重复求解。因此动态规划适用于子问题重复较多的场景

使用条件

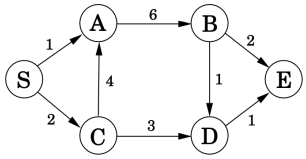
上面提到DP适用于需要反复用到子问题答案的问题求解。但是，很多问题看似满足这样的条件，但不能使用DP进行求解。下面给出几种常见的无法使用DP求解的情况：

1. **Optimal Substructure**：如果总问题的最优解不要求子问题最优解，那记录子问题最优解就显得很没有意义了（如果之前子问题的次优解反而可以让总问题最优，那动态规划的记录就失去意义了，这个和贪心有点像）
2. **Overlapping Sub-problems**：如果子问题没有重复，那也不需要记录，否则只会导致空间的浪费

Shortest Path in DAGs 最短路径问题

Introduction

给出一个有向图，每条边都有对应的长度，寻找2个节点间的最小距离



Solution

我们可以将这个问题拆分为更小的子问题，以节点D为例，我们有：

$$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\} \tag{34}$$

由此我们可以得到递推式。如果采用简单的递推，我们其实会画出一棵含有很多重复项的递归树；但如果我们在找完每个节点的最短路径后将其记录下来呢？这就是动态规划

Maximum Subarray Problem

Introduction

在上一章的分治法中我们提到了这个问题，当时我们将其拆分为子问题（divide）分别求解并计算横跨中间的可能（conquer）。按照分治法的分割方式，子问题似乎并不存在简单的重复，但实际上，中间merge的部分是存在重复的；而动态规划因为分割子问题的方式不同，所以速度更快

Solution

动态规划的在线算法在FDS中已经学过，不再赘述。其递推方程为：

$$\mathrm{OPT}(i)=\begin{cases}x_1 & \text{if } i=1\\ \max\{x_i, x_i+\mathrm{OPT}(i-1)\} & \text{if } i>1\end{cases} \tag{35}$$

其中$\mathrm{OPT}(i)=$ max sum of any subarray of x whose rightmost index is i. 以第i个元素结尾的最大子序列

Optimal Binary Search Tree

> 注意这里的"最优"指的是the best fot static searching，也就是仅搜索，不含插入和删除

Introduction

给定N个单词，$w_1<w_2<\cdots<w_N$，其中w_i被搜索到的概率为p_i。我们需要建一棵树，让期望的总搜索时间最小。首先，我们有：

$$T(N)=\sum_{i=1}^{N}p_i\cdot(1+d_i) \tag{36}$$

Subquestions

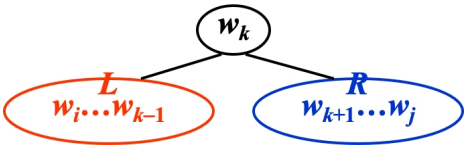
1. **Notation**

   为了更方便描述，我们定义一下一些概念

   - T_{ij}：OBST for $w_i,\dots,w_j$ $(i<j)$，由w_i到w_j构成的最优二叉搜索树
   - c_{ij}：Cost of T_{ij}，搜索时间
   - r_{ij}：Root of T_{ij}
   - w_{ij}：Weight of T_{ij}，节点概率的加和，$w_{ij}=\sum_{k=i}^{j}p_k$

2. **Analysis**

   关注T_{ij}，假设根节点为w_k，则其结构如下：

   我们有如下递推公式：

   $$\begin{aligned}c_{ij}&=p_k+\mathrm{cost}(L)+\mathrm{cost}(R)+\mathrm{weight}(L)+\mathrm{weight}(R)\\&=p_k+c_{i,k-1}+c_{k+1,j}+w_{i,k-1}+w_{k+1,j}\\&=w_{ij}+c_{i,k-1}+c_{k+1,j}\end{aligned} \tag{37}$$

   - 加上p_k是因为访问w_k需要一个单位的时间消耗，而其概率为p_k，所以平均消耗为p_k
   - 而$\mathrm{cost}(L)$和$\mathrm{cost}(R)$是在以子树的根节点为根的情况下得到的消耗，但是现在增加了一个w_k，相当于查找子树中任意一个节点的消耗都多了1，因此平均消耗增加为$\sum p=\mathrm{weight}$

   于是，我们就找到了最优方案的条件：

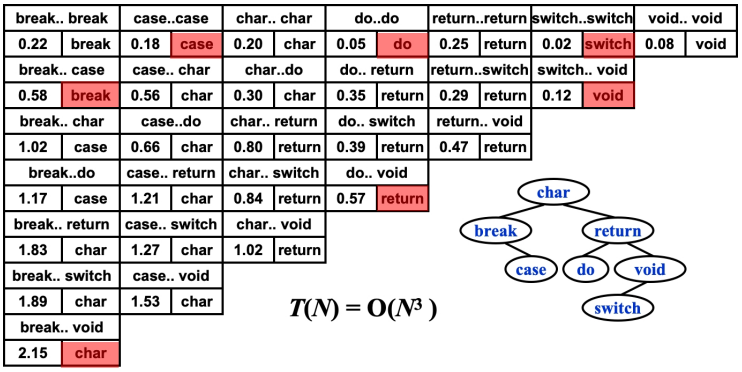
   $$T_{ij}\text{ is optimal}\Rightarrow r_{ij}=k\text{ is such that }c_{ij}=\min_{i<l\le j}\{w_{ij}+c_{i,l-1}+c_{l+1,j}\} \tag{38}$$

Solution

如果采用简单的递归操作，会发现大量的c_{ij}被重复计算了（比如计算c_{ij}时需要用到$c_{i,k-1}$，而计算$c_{i,j-1}$时还需要用到$c_{i,k-1}$），因此需要记住每个子问题的答案

此处给出一种迭代解决方案：

1. 第一行的代表1个单词的比较，显然只能以自己为根；
2. 第二行比较2个单词（注意每2个相邻的都需要比较），利用第一行的结论可以得出第二行的结论；
3. 第三行也同理，比较3个单词，利用第一行和第二行的结论可以给出第三行的结论
4. 以此类推，最后会找到整棵树的根。根据char这个根，可以划分出2棵子树（break～case和do～void），然后返回之前的表格中找这2棵子树的根，以此类推



Knapsack Problem

Introduction

给出一系列物品的价值v_i和重量w_i，需要在限定总重的情况下在包中装入总价值尽可能高的物品组合

Subquestions

1. **子问题**

   $\mathrm{OPT}(i,w) =$ optimal value of knapsack problem with items $1,\dots,i$ subject to weight limit w. 重量限制为w，可选物品为1～i

2. **递推方程**

   我们可以将子问题拆分为2种可能解：

   - 选中物品i，则问题转化为$\mathrm{OPT}(i-1,w-w_i)$
   - 不选物品i，则问题转化为$\mathrm{OPT}(i-1,w)$

   由此，我们给出递推方程：

$$\mathrm{OPT}(i,w)=\begin{cases}0 & \text{if } i=0\\ \mathrm{OPT}(i-1,w) & \text{if } w_i>w\\ \max\{\mathrm{OPT}(i-1,w),v_i+\mathrm{OPT}(i-1,w-w_i)\} & \text{othrerwise}\end{cases}\tag{39}$$

Solution

从0个物品、重量限制为0开始，找出每个$\mathrm{OPT}(i,w)$，列出一张$(n+1)\times(w+1)$的表格，每一格都可以由前面求得的结果得到。因此复杂度为$n\times w$（注意这不是一个多项式级别的解）

Ordering Matrix Multiplication

Introduction

给出几个矩阵，已知它们的规模（行×列）。因为矩阵具有结合律（但没有交换律），所以我们可以通过添加括号来减小计算量

> 对于矩阵乘法$M_{1[10\times20]}\cdot M_{2[20\times50]}\cdot M_{3[50\times1]}\cdot M_{4[1\times100]}$
>
> - 如果按照$M_{1[10\times20]}\cdot(M_{2[20\times50]}\cdot(M_{3[50\times1]}\cdot M_{4[1\times100]}))$的顺序相乘，
>
>   则需要的时间为$50\times1\times100+20\times50\times100+10\times20\times100=125,000$

- 如果改为$((M_{1[10\times20]}\cdot M_{2[20\times50]})\cdot M_{3[50\times1]})\cdot M_{4[1\times100]}$，
  则时间为$20\times50\times1+10\times20\times1+10\times1\times100=2,200$

Solution

1. **子问题分解**

   假设M_i是一个$r_{i-1}\times r_i$的矩阵，m_{ij}为M_{ij}的最优解，不难得到如下递推式（拆分子问题的方式见"复杂度分析"）

$$m_{ij}=\begin{cases}0 & \text{if } i=j\\ \min_{i\le l<j}\{m_{il}+m_{(l+1)j}+r_{i-1}r_lr_j\}\end{cases} \tag{40}$$

2. **代码实现**：代码实现比想象中更加简单粗暴，实际上就是构建一个如下图所示的二维数组，记录每一种情况的结果

$$\begin{array}{lllll} m_{1,N} & m_{2,N} & \dots & m_{N-1,N} & m_{N,N}\\ m_{1,N-1} & m_{2,N-1} & \dots & m_{N-1,N-1} & \\ \vdots & \vdots & \dots & & \\ m_{1,2} & m_{2,2} & & & \\ m_{1,1} & & & & \end{array} \tag{41}$$

```
void OptMatrix( const long r[], int N, TwoDimArray M ) {
  int i, j, k, L;
  long ThisM;
  for( i = 1; i <= N; i++ )
    M[i][i] = 0;
  for( k = 1; k < N; k++ )
    for( i = 1; i <= N - k; i++ ) {
      j = i + k;
      M[i][j] = Infinity;
      for( L = i; L < j; L++ ) {
        ThisM = M[i][L] + M[L + 1][j] + r[i - 1] * r[L] * r[j];
        if( ThisM < M[i][j] )
          M[i][j] = ThisM;
      }
    }
}
```

复杂度分析

$M_{ij}=M_i\cdot\ldots M_j$，$b_n$ is the number of different ways to compute M_{1n}

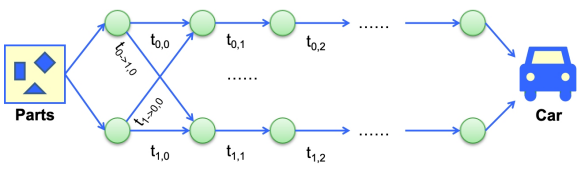
按照最后一步相乘的子序列对情况进行分类（不会有重复，因为最后一步不同一定不是同一种乘法，只需要考虑是否有缺漏情况即可），$M_{1n}=M_{1i}\cdot M_{i+1,n}$

$$\begin{aligned}\Rightarrow b_n&=\sum_{i=1}^{n-1}b_ib_{n-1}\\ &=O\left(\frac{4^n}{n\sqrt{n}}\right)\end{aligned} \tag{42}$$

Product Assembly

Introduction

有2条流水线装配一台车，每个阶段所需要的时间都不一样，且流水线间可以切换。求出最小的总装配时间



Solution

其实这本质上是一个最短路径问题，只需要记录做完每个步骤所需要的最短时间即可。The optimal path to $stage$ is based on an optimal path to $stage-1$

```
f[0][0] = 0;
f[1][0] = 0; // f[0][i]代表从第一条流水线到第i步的所需要的最短时间，f[1][i]则来自第二条流水线
for( int stage = 1; stage <= n; stage++ )
  for( int line = 0; line <= 1; line++ ) {
    f[line][stage] = min(
      f[line][stage - 1] + t_process[line][stage - 1],
      f[1 - line][stage - 1] + t_transit[1 - line][stage - 1] );
  }
solution = min( f[0][n], f[1][n] );
```

> 要我说，这个题用贪心会快得多，不知是否正确?

All-Pairs Shortest Path

Introduction

For all pairs of v_i and v_j $(i \neq j)$, find the shortest path between. 和最短路径算法的区别就是，这个问题需要找到每两个点间的最短路径，而非给定两个点

如果我们用single-source algorithm做V遍（也就是对每个节点都跑一遍最短路径算法），则所需要的复杂度为$O(V^3)$

Solution

1. **Sub-problem**：这里的拆分方式比较有意思，我们仅考虑前k个节点（相当于是将图的规模减小了）

   定义$D^k[i][j]$：i经过前k个节点到达j的最短距离。则$D^{-1}[i][j]=\text{Cost}[i][j]$，$i$到$j$的最短路径为$D^{N-1}[i][j]$

   $$D^k[i][j]=\min\{\text{lenth of path } i\to\{l\leq k\}\to j\} \tag{43}$$

2. **递推**：子问题拆分完毕后，递推就显得比较简单了。i到j的最小距离要么不经过k，要么经过k，分别对应$\min$函数中的两种情况

   $$D^k[i][j]=\min\{D^{k-1}[i][j],D^{k-1}[i][k]+D^{k-1}[k][j]\} \tag{44}$$

3. **代码**

   ```
   void All_Pairs( TwoDimArray A, TwoDimArray D, int N ) {
     int i, j, k;
     for( i = 0; i < N; i++ )
       for( j = 0; j < N; j++ )
         D[i][j] = A[i][j]; // 处理D^{-1}[i][j]
     for( k = 0; k < N; k++ )
       for( i = 0; i < N; i++ )
         for( j = 0; j < N; j++ )
           if( D[i][k] + D[k][i] < D[i][j] )
             D[i][j] = D[i][k] + D[k][j];
   }
   ```

   从代码中的三重循环不难发现，本算法的复杂度为$O(N^3)$

考点

动态规划在期末考中会有一道代码题，难度较大。客观题的考点一般是写递归式，或者考察对DP的理解

1. **概念类**：In dynamic programming, we derive a recurrence relation for the solution to one subproblem in terms of solutions to other subproblems. To turn this relation into a bottom up dynamic programming algorithm, we need an order to fill in the solution cells in a table, such that all needed subproblems are solved before solving a subproblem. Among the following relations, which one is impossible to be computed?

   A. $A(i,j)=\min\{A(i-1,j),A(i,j-1),A(i-1,j-1)\}$

   B. $A(i,j)=F(A(\min\{i,j\}-1,\min\{i,j\}-1),A(\max\{i,j\}-1,\max\{i,j\}-1))$

C. $A(i,j)=F(A(i,j-1),A(i-1,j-1),A(i-1,j+1))$

D. $A(i,j)=F(A(i-2,j-2),A(i+2,j+2))$

关注C选项，虽然其出现$j+1$，看上去是使用了还未经过的内容。但实际上，当我们开始计算(i,j)时，$(i-1,1\le j\le \max_j)$都已经计算过了，换句话说，$(i-1,j+1)$已经在遍历$i-1$的过程中计算得到了

而D选项并没有，因为i和j无法同时超出范围。只有当$i<i_{\text{now}}$时，才可以有$j>j_{\text{now}}$；j同理

换句话说，DP要么一行一行做，要么一列一列做。因此，要么i没有超过的，要么j没有超过的

2. **代码题**：建议时间充足且检查完毕再作答。如果时间有但不算太多，又刚好想到比较蠢的暴力算法，可以先写，说不定就整个过去了。DP需要注意的除了状态方程之外，还有初始化

GREEDY 贪心

Introduction

1. **定义**：每一步都选择当前条件下最优解。Make the best decision at each stage, under some greedy criterion. A decision made in one stage is not changed in a later stage, so each decision should assure feasibility.
2. **优缺点**：

   贪心算法找到的是一个局部最优解，未必每次都能找到全局最优解。Greedy algorithm works only if the local optimum is equal to the global optimum.
   虽然无法保证找到最优解，但贪心解总是可以找到一个不坏的解。Greedy algorithm does not guarantee optimal solutions. However, it generally produces solutions that are very close in value (heuristics) to the optimal, and hence is intuitively appealing when finding the optimal solution takes too much time.

Activity Selection Problem

Introduction

Given a set of activities $S=\{a_1,a_2,\dots,a_n\}$ that wish to use a resource. Each a_i takes place during a time interval $[s_i,f_i)$. Activities a_i and a_j are compatible if $s_i\ge f_j$ or $s_j\ge f_i$, in other words, their time intervals do not overlap.

Our target is to select a maximum-size subset of mutually compatible activities.

Solution

1. **DP Solution**：$c_{ij}=c_{ik}+c_{kj}+1$，相当于要将整个这样的二维数组都算出来，因此复杂度为$O(N^2)$
2. **Greedy Solution**：我们有下面几个greedy rule，注意每一步都需要选择no overlapping的
   - Select the interval which starts earliest
   - Select the interval which is the shortest
   - Select the interval with the fewest conflicts with other remaining intervals
   - Select the interval which ends first（本方法可以找到最优解）

   因为在贪心前需要进行排序，所以复杂度为$O(n\log n+n)=O(n\log n)$

   > 注意这个排序的时间，因为贪心本身往往是线性的复杂度，但如果涉及排序，排序就会成为复杂度最大的贡献者

Huffman Code

Introduction

> 假设四个字符为a、u、x、z，如果采用常规的编码方式，令$a=00,u=01,x=10,z=11$，则编码一篇含有1000字符的文本需要2000bit

现在我们定义frequency：number of occurrences of a symbol，统计每个字符出现的次数，并将出现次数较多的字符编码为长度较短的2进制数。比如$a=0, u=110, x=10, z=111$

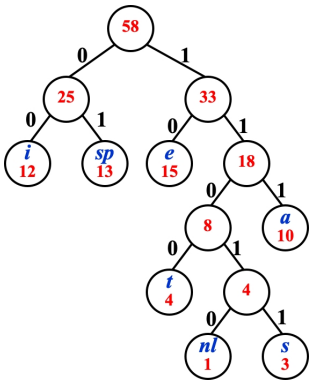
> 为了保证解码是唯一的，要保证no code is a prefix of another，没有一个编码是其他编码的前缀

如果我们给出字符及其对应的出现次数，应当如何对其进行编码?

Haffman's Algorithm

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1

根据上面的表格，我们可以画出这样一棵编码树：



想要画出上面这样的一棵编码树，需要先找到出现频率最低的2个字符nl和s，将其连到一个根节点上，并将根节点的frequency定义为nl与s的加和（因为到达这棵树的次数 = 到达树中所有节点次数的加和）；然后将这棵树放到小根堆中，从小根堆中重新取出frequency最小的2个，建树，入堆……

当小根堆中只剩一个元素时，说明编码树构建完毕。每个字符的哈夫曼编码就是根节点到它的路径（左0右1）

```
void Huffman (PriorityQueue heap[], int C) {
  // 一共有C个字符
  for (int i = 1; i < C; i++) {
    创建一个新节点new_node
    删除小根堆中的root，连到new_node的左孩子上
    删除小根堆中的root，连到new_node的右孩子上
    new_node->Weight = new_node->Left->Weight + new_node->Right->Weight
    将new_node入堆
  }
}
```

$$T = O(C \log C) \tag{45}$$

可以证明，本贪心算法总可以给出最优解

---

1. 注意题目中2-3树的意思是，它的内部节点的孩子数为2或3，即b+ tree of order 3；同样的，2-3-4树内部节点的孩子数为2或3或4，即order为4。 ↩