CSC A48 - 第4单元 - 高效解决问题 1.-

链接列表是否高效？

到此为止，我们已经学会了如何表示、存储、组织和搜索可能是复杂的数据项的集合。我们知道什么是*抽象数据类型*，以及为什么它们有助于提出解决问题的方法，从而使解决方案独立于具体的实现。我们研究了列表，以及它们作为链接列表的实现，我们用这些来建立一个简单的数据库，能够回答对我们集合中的项目的一些查询。

毫无疑问，你将来会在各种应用中使用列表。因此，在我们结束对列表的讨论之前，值得花点时间思考一下，对于需要频繁查找集合中的项目（无论是查看其内容、更新存储的数据，还是进行汇总计算）的问题，列表作为一种解决方案，其*效率*如何。

假设我们想用链表来实现一个全功能的数据库引擎。该数据库应该能够存储大量数据项的信息（想得大一点--
链接列表中的数千万、数亿、甚至数十亿的节点）。问题是，在*我们对数据库进行搜索所需的时间方面，我们可以期待什么？*

> 在一个集合中拥有如此多的数据项是非常常见的。
>
> IBM的Watson使用数百万份文件来回答Jeopardy的游戏问题
> 谷歌图像搜索在2010年时索引了远超过100亿张图片
> Facebook在2018年9月拥有约15亿日活跃用户
> 亚马逊在全球范围内销售了超过30亿种不同的产品!
>
> 因此，思考*什么是维护一个非常大的收藏品的最有效方式*是非常重要的。

为了充分理解这个问题，我们必须思考一下数据是*如何*存储在列表中的。一个常见的假设，也是对许多数据库应用来说比较合理的假设是，数据的添加*没有特定的顺序*，或者说，*就*我们可能想用来回答数据库查询的任何数据字段而言，*列表中的条目是随机排列的。*

我们通常还假设*查询是以随机顺序进行的*。这是一个合理的假设--
想想在1分钟内来自世界各地的谷歌搜索--
有可能，来自墨西哥、澳大利亚、新加坡和芬兰的人所进行的搜索没有任何顺序或模式，而这些人恰好都在那个特定时间上网。

最后，我们必须提供一些*效率*的定义，以便我们能够评估我们的列表做得如何。对于一个数据库引擎来说，一个合理的衡量标准是，*在我们发现我们要找的那个数据项被找到之前，需要检查多少个数据项。*

现在，回顾一下上一单元的内容，要在一个链表中进行搜索，我们必须做一个***列表遍历***。这意味着从列表的头部开始，沿着列表的节点移动，直到我们找到包含我们要找的信息的节点。

*问题。*我们需要看多少个节点才能找到我们想要的那个？

- 如果我们***超级幸运***，我们想要的数据会在***头部节点***，因此***我们只需查看1个节点***就能找到我们想要的东西，但对于随机排序的数据，这种情况发生的几率是***1/N***，其中***N是我们列表中的节点数***。对于一个有1,000,000,000个条目的列表来说，这种情况发生的几率是非常大的。
- 如果我们***超级不走运***，我们想要的数据在列表的***尾部***（或者它不在列表中），我们必须在找到它之前查看***所有N个节点***，或者可以确定它不在那里。
- 大多数时候，我们既不是超级幸运，也不是超级不幸运。数据在列表的某个地方，有时更接近头部，有时更接近尾部。因为数据是随机排序的，所以在我们运行很多很多的查询之后，***我们需要检查的节点数量平均为N/2***。

这是有道理的。有时我们找到的数据更靠近头部，有时更靠近尾部，但这两者相互平衡，所以平均而言，我们必须查看***链接列表的一半***来回答任何特定的查询。

如果我们的列表只有几千个条目，这确实不算太糟糕。但是，一旦我们的链接列表增长到数百万条，甚至更多，***回答一个查询的成本***就变得太大。简单地说，我们的链接列表***必须查看太多的数据项，以便找到一个特定的数据项***。这就意味着从数据库中请求信息的程序需要更长的等待时间。在某些时候，列表太长了，等待时间变得不切实际。

2.- 为什么我们需要翻阅所有的数据来找到一些东西

我们可能倾向于认为问题的根本原因是我们的链表结构。事实上，我们的链表有一个主要的限制，即我们不能在不进行列表遍历的情况下访问列表中的元素。然而，我们问题的根源并不限于链表的结构。

例如，考虑一个包含2017年和2018年所有热门歌曲名称的***数组***（这是一个小列表，所以我们可以在这里展示一个例子，但你应该想到这适用于任何规模的数组）。

I'm the One
Despacito
Look What You Made Me Do
Bodak Yellow
Rockstar
Perfect
Havana
God's Plan
Nice for What
This Is America
Psycho
Sad!
I Like It
In My Feelings
Girls Like You
Thank U, Next

我们想要回答的问题是。

在一个*随机排序的*数组中搜索一个特定的项目，是否比在一个具有相同（无序）条目的链接列表中搜索更*有效率*？或者，相当于同样的事情，我们问的是，使用数组是否能让我们在平均少于*N/2个*项目的检查中找到我们想要的东西。

看一下上面的数组，应该很清楚，数据的排序没有什么规律。如果我们需要找到一个特定的歌曲名称，我们必须从顶部开始，在数组中寻找，直到我们找到我们想要的东西。*这就是所谓的线性搜索*。就像一个链接列表一样，我们有可能非常幸运，我们的查询就在数组的第一个条目上，我们也可能非常不幸运，所以查询的项目在最后，或者不在数组中；平均来说，我们必须在找到我们的信息之前看遍数组的*一半*。如果数组有*N个*条目，我们又要平均翻阅*N/2个*条目。

这意味着，从*我们需要查看的项目数量的*角度*来回答一个查询*，链接列表和数组的*效率是一样的*。虽然在实际运行时间上可能会有轻微的性能差异（因为链接列表比数组有更多的开销），但随着集合大小的增长，这种微小的差异并不重要：这两种存储信息的方式对于快速回答查询都变得不够了。

3.- 组织我们的数据以进行有效搜索

如果我们要找到一种更快的方式来回答查询，我们需要处理我们的数据的随机顺序。我们需要对它进行排序。一旦我们的数据被排序，我们就可以更有效地搜索特定项目。

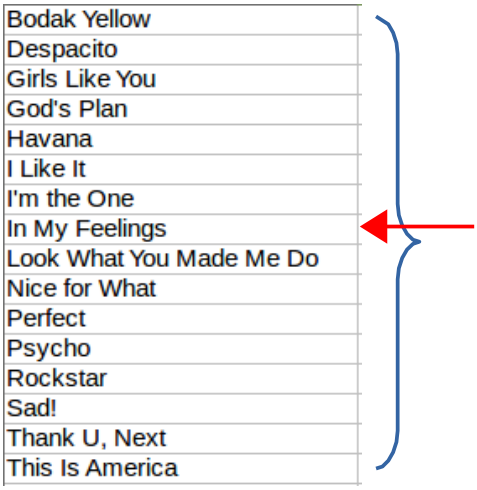
Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America

对于一个人来说，在上面的排序数组中查找一首特定的歌曲要容易得多。我们明白这些条目是按顺序排列的，所以我们可以很容易地***找到某首特定歌曲应该在的位置***。这就是为什么在书的背面、图书馆的书架或电话簿（在那个实际上是一本印刷书的时代！）中的所有内容索引都是排序的。

在编程方面，对数据进行排序的直接好处是使数据更容易搜索，其结果是在一个排序的数组上，比如上面的数组，我们只需要检查几个项目就能找到我们要找的东西。

二进制搜索

在一个***排序的数组***中寻找一个项目的过程被称为二进制搜索。假设我们想在上面的数组中找到 "I'm the One "这首歌。二进制搜索过程如下图所示。



1) 考虑整个数组，找出数组中间的物品（用数组的长度除以2，然后向下取整）。***检查中间的物品。***

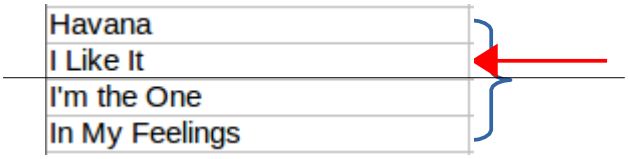
* 如果**它是我们要找的物品**，我们就完成了。
* 否则
  - 如果我们要找的项目***小于***我们刚刚检查的项目，那么它***一定在数组的前半部分***。
  - 如果我们要找的项目***大于***我们刚刚检查的项目，它***一定在数组的后半部分***。

在我们的数组中，"I'm the One "排在 "In My Feelings
"之前（记住，所有条目都是按字母排序的！），所以我们取***数组的前一半。***

2) 我们对原始数组的这块内容执行完全相同的过程。找到中间的项目，并将其与我们的查询进行比较。再来一次

* 如果**它是我们要找的物品**，我们就完成了。
* 否则
  - 如果我们要找的项目***小于***我们刚刚检查的项目，那么它***一定在数组的前半部分***。
  - 如果我们要找的项目***大于***我们刚刚检查的项目，它***一定在数组的后半部分***。

我们想要的歌曲 "I'm the one "在中间项 "上帝的计划
"之后，所以我们需要采取这个阵列的后半部分。



3) 重复同样的过程，直到我们找到我们想要的项目！"。 "I'm the One "是在 "I Like It
"之后，所以我们取上面数组的后半部分。

我们找到了我们要找的那首歌!要做到这一点，我们必须检查***4个条目***。在未排序的数组中

我们本来预计要看***N/2=16/2=8个***条目才能找到一首歌（当然，允许有或多或少的运气）。

这样看来，对数组进行排序是在为我们做一些有用的事情。让我们看看我们是否能弄清楚发生了什么。

* 二进制搜索过程利用数组被排序的事实来*预测*我们想要的数据*必须*在*数组的哪一半*。
* 这意味着，我们*每检查一个项目*，就可以*丢弃一半的剩余条目*。当我们选择数组的哪一半进行下一步搜索时，我们可以确定我们永远不必看另一半中的任何项目
* 因此，我们每完成一步二进制搜索，剩下要检查的项目数量就减少一半。

比如说。

如果我们的初始数组有***1024个***条目的长度。

* 经过第一步的二进制搜索，我们剩下***512个***条目需要检查
* 经过第二步的二进制搜索，我们还剩下***256个***条目
* 第3步之后，我们还剩下***128个***条目
* 第4步之后，我们还剩下***64个***条目
* 第5步之后，我们还剩下***32个***条目
* 第6步之后，我们还剩下***16个***条目
* 第7步之后，我们还剩下***8个***条目
* 第8步之后，我们还剩下***4个***条目
* 第9步之后，我们还剩下***2个***条目
* 第10步之后，我们还剩下***1个***条目

剩下的要检查的项目数量*很快就*会减少*!*在上面的例子中，通过检查***10个项目***，我们能够在***1024***大小的数组中找到任何项目。而这是*在我们非常不走运*，我们想要的项目是我们检查的最后一个项目的情况*下*!

与我们*在*未排序的数组或链接列表*上进行线性搜索所*要检查的预期***N/2=1024/2=512***项相比，应该很清楚，二进制搜索是一种*更有效的*查找信息的方法。

一般来说，二进制搜索要检查多少个项目？

从上面我们可以看到，在每一步中，剩下的要检查的项目数量都会减少一半。问题是，给定一个*初始值N，即数组中的条目数，需要多少步才能达到只剩下一个元素的程度？-*
这相当于找出*我们需要检查的最大条目数，以找到我们想要的东西。*

答案是$k=\log_2(N)$

对于有***1024个***条目的数组来说，***$k=\log_2(N)=10$***，这正是我们上面为了得到一个条目所要做的步骤的数量。让我们看看在三种情况下，随着数组中条目数的增加，我们需要检查多少个项目，这意味着什么。

a) 我们在二进制搜索中需要查看的***最大***（最坏情况）项目数
b) 我们需要查看的线性搜索的***平均***项目数量
c) 我们在线性搜索中需要查看的***最大***（最坏情况）项目数

NB	***二进制搜索：$\log_2(N)$***	***线性搜索（平均）。N /2***	***线性搜索（最差）。N***
2	***1***	***1***	***2***
4	***2***	***2***	***4***
8	***3***	***4***	***8***
16	***4***	***8***	***16***
.			
.			
1,024	***10***	***512***	***1,024***
2,048	***11***	***1,024***	***2,048***
4,096	***12***	***2,048***	***4,096***
.			
.			
1,048,576	***20***	***524,288***	***1,048,576***
.			
.			
33,554,432	***25***	***16,777,216***	***33,554,432***

从上表中，你应该很清楚地看到，***二进制搜索的效率高得惊人***。即使在最坏的情况下，只检查了25个项目，我们也能***在一个有3300多万个条目的数组中***找到***任何一个项目！相反，在一个未排序的数组或链表上进行线性搜索，预计要经过1600多万个项目。***相反，在一个未经排序的数组或链表上进行线性搜索，预计平均要经过1600多万个项目，如果运气不好的话，还要经过3300万个项目才能找到我们想要的东西。

3.- 计算的复杂性

上述想法可以被做成一个具体的、通用的框架，用于比较不同算法在执行任务时的***计算成本。***

在理解我们如何比较不同的算法时，一个关键的想法是，***我们要找到一个衡量一个特定算法所要做的工作量与N的函数，即该算法所处理的数据项的数量。***我们把这种衡量标准称为算法的***计算复杂性***。

在上面的例子中，我们看到***线性搜索***在最坏的情况下要检查***N个***项目，而***二进制搜索***只需要看$\log_2(N)$。之所以要看***N***的函数，是因为

我们可以***预测***一个算法对于越来越大的数据集将如何执行。在我们的搜索例子中，我们有两个不同的功能。

$$f_{\text{binary search}} = Log_2(N)$$
$$f_{\text{linear search}} = N$$

所以我们说二进制搜索的复杂度为***log_2 (N)***，而线性搜索的复杂度为***N***。由于***log_2 (N)***的值比***N***增长得慢得多，我们可以得出结论，***无需在每个可能的数组上测试，二进制搜索比线性搜索***对大集合的***效率要高得多。***我们可以通过绘制两个函数在***N***值增加时的曲线，并比较它们所要做的工作量来形象地说明这一点。

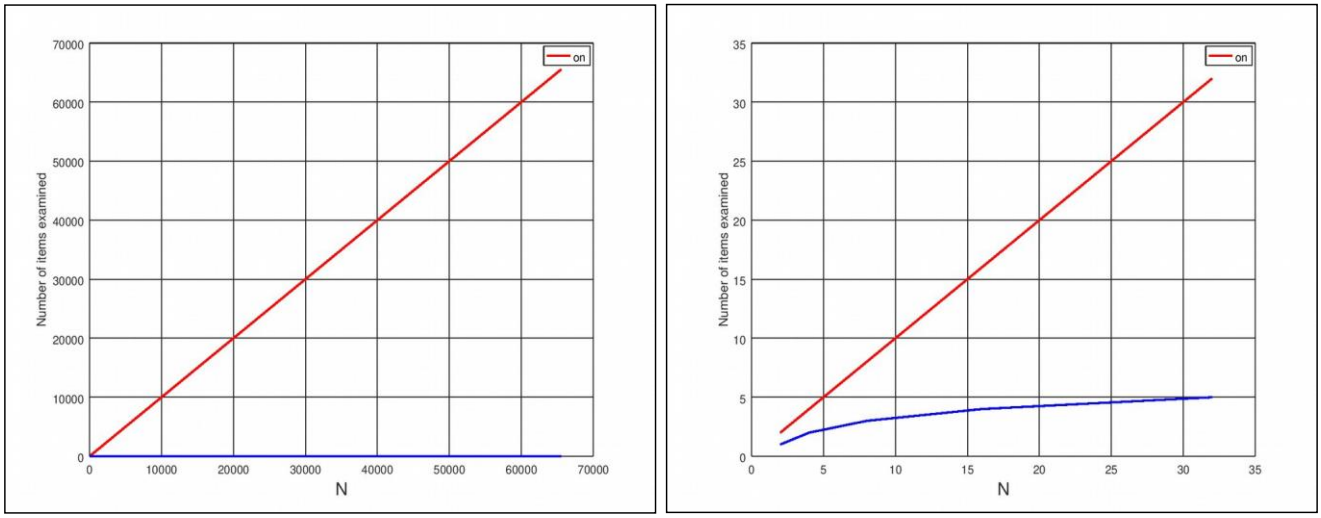


图1：两种不同搜索算法的复杂性的可视化。线性搜索（红色）和二进制搜索（蓝色）。二进制搜索的效率高得惊人，与线性搜索相比，它在大数组上看起来根本不需要做任何工作!对于较小的N值，每个函数的曲线形状更容易在图上进行比较（右边）。

从上面的图中可以看出，两种算法中哪一种***更有效***。我们可以从中得出的结论是。

给定两种算法和描述其计算复杂度的***N***的函数，***随着数据项数量的增加，***增长更慢的函数***总是会胜出***。因此，具有较慢增长的函数的算法被认为具有***较低的复杂性***，因此被证明对大数据集合来说是***更有效的***。

问题。关于两个在相同数据上做搜索的程序的***运行时间***，但其中一个使用二进制搜索，而另一个使用线性搜索，我们现在能说什么？

从上面的讨论中可以看出什么。

能够比较执行某些任务的不同可能方式，对于实施任何问题的良好解决方案至关重要。在计算机科学中，比较算法的方法是通过了解***其计算复杂性***。计算复杂性表示为***N的函数***，即程序要处理的数据项的数量。

大O记号

为了能够更好地理解和比较不同算法的复杂性，我们使用了一种特殊的符号，叫做***大O***符号。***大O符号***的重要性在于，它允许我们从函数的一般类别来考虑算法的效率。下面是我们的意思。

假设我们有不同的数组***线性搜索***的实现（也许是由不同的开发者用不同的编程语言编写的），***二进制搜索***也是如此。然后我们着手仔细测量它们在不同大小的数组上的***运行时间***，我们发现它们的表现如下。

a) .75*N (例如，对于N=10，该实施方案运行时间为7.5秒)
b) 1.25*N
c) 2.43*N
d) 1.15 * Log_2 (N)
e) 1.75 * Log_2 (N)
c) 15.245 * Log_2 (N)

上面的结果需要一些思考--
我们知道*线性搜索*要做多少工作才能在数组中找到一个项目，而且在最坏的情况下，这将是在数组中寻找所有***N个***条目。但是我们还没有考虑到当引入不同的实现方式时会发生什么。

也许用C语言实现的会比用Java实现的略快，也许用Java实现的会比用Matlab写的快一些。但需要注意的是，***所有这些都是N的线性函数***，这是***线性搜索算法***的一个属性。唯一可以在（正确的）实现中改变的是常数因素。我们可以用不同的方式说完全相同的话。任何正确的***线性***搜索算法的实现都会有一个复杂度，其特征是***c*N***，其中***c***是某个常数，对于大的***N***值。

对于***二进制搜索***的不同实现方式也是如此。它们都是***N的对数函数***乘以某个常数，这个常数与实现有关。

我们希望有一种方法可以以一种***独立于实施的***方式来比较算法。我们需要知道***哪种算法更好，哪种算法需要更少的工作***来解决一个特定的任务。如果我们能以这种方式分析算法，我们就能始终选择最有效的算法。大O符号明确了***N的最快增长函数***，该***函数***表征了某种算法的复杂性。实际定义是这样的

$$f(x) = O(g(x))$$

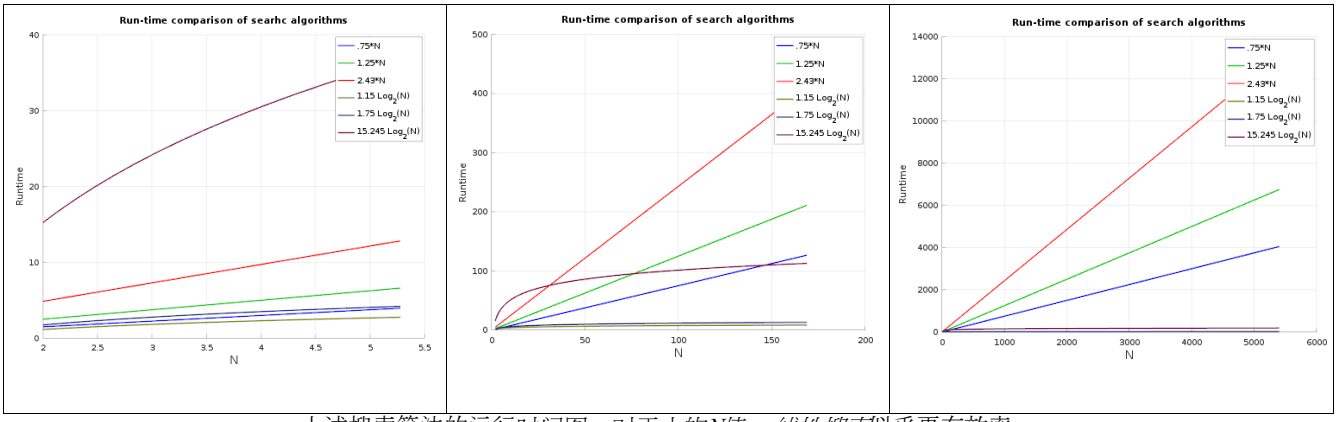
当且仅当，存在一些正的常数***c***，使得对于***x***的足够大的值来说。

$$|f(x)| \leq c \quad g(x), \quad x > x_0$$

换个说法，说***某个算法的运行时间f(x)是O(g(x))级的***，意味着在假设***x***足够大的情况下，运行时间小于或等于某个常数***c*g(x)***。就我们上面的搜索算法而言，我们可以这样说。

- 线性搜索的复杂度为***O(N)--***理解为 "线性搜索的复杂度为N阶"
- 二进制搜索的复杂度为***O(Log$_2$ (N))*** - 读作 "二进制搜索的复杂度为Log$_2$ (n)"

事实上，我们可以更进一步说，二进制搜索的复杂性是***O(Log(N))***，而不需要担心对数是以2为底的事实。***为什么说这是合理的呢？***考虑一下下面的图。



上述搜索算法的运行时间图。对于小的*N*值，*线性搜索*似乎更有效率。
然而，当*N=150的*时候，所有的*线性搜索*实现都已经超过了*最慢的二进制搜索*实现，而当*N=1000*的时候，*线性搜索*就全部结束了。结论是：无论如何实现，只要*N*足够大，*二进制搜索*最终会胜出。

在上面的图表中不能忽略的一点是，***对于足够大的N来说，大O复杂度增长较慢的算法将最终获胜***。*Log(N)*函数（有任何基数）比*任何线性函数增长慢得多*，所以我们*总是期望二进制搜索在足够大的数组上更快*。

从算法复杂性到问题复杂性

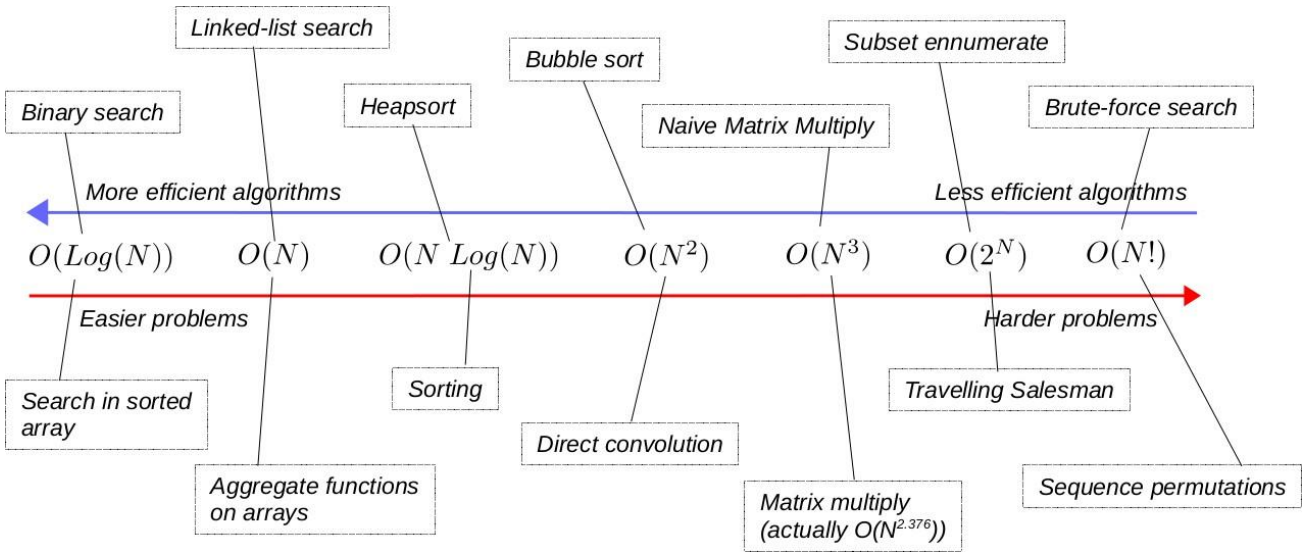
我们开始的目标是了解在数组中寻找特定项目的两种不同算法的效率如何。复杂性分析是实现这一目标的绝佳工具。然而，在计算机科学中使用复杂性分析的背后还有一个更强大的想法。

我们可以研究、量化和证明有关***某个问题的复杂性的***结果。这有什么区别呢？让我们举个例子。对一个数组进行排序（我们很快会更详细地回到这个例子中来！）。

- ***算法复杂度***意味着我们可以研究不同的算法来对一个数组进行排序，并计算出随着数组大小的增长，哪一种算法会更有效率。
- ***问题复杂性***是指我们研究排序的实际问题，并试图找出***理论上的下限，即最好的算法要对***一个数组进行排序***需要做多少工作***。

例如，证明***线性搜索***的复杂度为***O(N)***，就相当于证明不可能找到一个复杂度小于***O(N)的***算法来做***线性搜索***。

这是一个强大而重要的想法，因为它允许我们根据问题的计算***难度对***其进行***分类***。较难的问题的特点是由快速增长或非常快速增长的函数给出的***大O***复杂性。作为一个开发者，熟悉你有一天可能要处理的*不同类别的问题*是很重要的，并且要仔细考虑，考虑到一个算法正在处理的问题的复杂性，对它的期望是合理的。



上面的图显示了具有不同***大O***复杂性的算法（在上面）和问题（在下面）的例子。请注意，对于给定的问题（如排序），你可以很容易地找到其复杂度大于（冒泡）该问题的最佳已知复杂度界限的算法。知道一个给定问题的最佳已知复杂度估计值是什么，你就可以评估解决*该*问题的算法有多好。

明年，在B63课程中，你将学会分析算法以弄清其复杂性，你将学习不同的方法来衡量、量化和表达我们对不同算法和不同ADT上操作的复杂性的认识。

现在，不要忘记，我们可以用不同的量来衡量复杂性：*一个集合中的项目被访问的次数，一个算法的运行时间，某个函数要执行的数学运算的数量*。使用哪种测量方法取决于你要解决什么问题，但分析，以及它告诉你一个算法或问题在解决它方面做了多少工作，是以同样的方式来理解的。

另外，别忘了，***计算复杂性分析***适用于每一种类型的问题，而不仅仅是在一个集合中搜索信息。它在计算机科学的所有涉及数值计算的领域中都无比重要，包括机器学习和人工智能。确保你对上面的观点建立了良好的理解，明年在B63中扩大和加强它。

所以这意味着执行情况并不重要，对吗？

不完全是--
有效地解决一个问题需要你***首先考虑你解决问题的算法的复杂性***。但是，一旦你确信你有一个具有最佳复杂度的算法，你就需要为它写一个好的实现方案一旦我们为一个问题选择了一个具有*正确复杂度的*算法，我们就到了*关心那些讨厌的常数因素的时候了。*

回到我们最初的搜索算法的例子，一旦***N***足够大，例如***N=1000000，***实现之间的差异就变得很重要。

1.15 * Log_2 (N) = 22.9
15.245 * Log_2 (N) = 303.6

因此，在你需要等待20秒左右的结果，还是需要等待5分钟以上的结果之间，执行情况是完全不同的。

练习。考虑一个需要2个输入数组的函数。

```
void multiply_accumulate(float input[N], float output[N])
```

该函数计算 "*输出*"数组中的每个值为

$$output[i] = \sum_j input[i] * input[j]$$

也就是说，*输出*数组中的i^{th} 条目是*输入*数组中的i^{th}条目与*输入*数组中的其他所有条目（包括它自己）相乘的结果，并将所有这些相加。

实现这一功能的一种方法如下所示。

```
void multiply_accumulate(float input[N], float output[N])
{
      int i,j;

      for (i=0; i<N; i++)
      {
            output[i]=0。
            for (j=0; j<N; j++)
            {
                  output[i]=output[i]+（input[i]*input[j]）。
            }
      }
}
```

问题。上图所示函数的***大O***复杂性是多少？(如果你很难从代码中找出答案，可以试着列出***N的***一个小值的循环所做的操作，看看这是否能给你一些建议。

问题。如果我们不*访问输入数组中的条目*，而是用函数进行的*乘法数量*来定义复杂度，那么***大O的***复杂度是否会发生变化？(这个定义更可能与这个函数的运行时间有关，因为它不仅需要访问信息，还需要对数据进行计算)

问题。这就是我们对这个问题所能做的最好结果吗？是否有更好的算法？如果有，该算法的***大O***复杂性是多少？

4.- 回到我们的问题上--如何使搜索信息更有效率

在本单元开始时，我们试图了解在让我们搜索集合中的项目方面，链表的效率如何。我们现在已经看到，在链接列表中搜索的复杂度为***O(N)***，正因为如此，对于大型集合来说，它们不是存储需要经常搜索的信息的最佳方式。

我们还看到，***二进制搜索***的复杂度为***O(Log(N))***，这使得它在寻找信息方面的效率令人难以置信，即使是非常大的集合。因此，基于这一点，我们可以得出结论，我们应该放弃链接列表，我们应该简单地坚持使用排序项目的数组，这样我们就可以有高效的搜索。

然而，当我们意识到阵列的局限性（固定的容量，要么不足以容纳数据，要么浪费空间）使得它们很难成为管理和存储大量物品集合的良好解决方案，因为这些物品的数量事先并不清楚，而且集合的大小会随着时间的推移发生很大的变化。

所以我们现在有一个严重的问题。

在这一点上，看起来我们可以有以下两种选择

- 一种动态的数据结构，使我们能够在不浪费空间的情况下组织和维护大量的项目集合。

或

- 一个数组，有固定的容量和相关的问题，但在这里我们可以进行二进制搜索来有效地搜索信息。

我们错过了什么？

假设我们决定高效的搜索对我们来说比不浪费空间更重要，所以我们愿意使用一个数组，对我们的集合来说足够大，这样我们可以进行二进制搜索并快速找到项目。听起来是个不错的计划，*除了我们没有考虑到这样一个事实，即我们希望项目以任意的顺序被添加到我们的集合中*。二进制搜索要求我们的数组被排序。

因此，在我们决定最好的办法是使用一个排序的数组和二进制搜索之前，我们必须考虑首先对数组进行排序的***成本***（你现在知道我们指的是***计算复杂性***）。

分拣的成本

考虑一个有***N个***条目的（未排序）数组。让我们考虑一下我们能想到的最简单的排序算法：***冒泡排序***。如果你以前没有见过冒泡排序，这里有一个简单的实现。

```
void BubbleSort(int array[], int N).
{
   // 遍历一个数组，交换任何条目，使得
   // array[j] > array[j+1]。继续这样做，直到
   // 数组被排序（最多，N次迭代） int t;


   for (int i=0; i<N; i++)
   {
     for (int j=0; j<N-1; j++)
     {
       如果(array[j]>array[j+1])
       {
         t=array[j];
         array[j]=array[j+1];
         array[j+1]=t;
       }
     }
   }
}
```

该函数在数组中进行，反复交换元素，使最大的一个元素

向数组的末端移动。在每次迭代结束时，肯定又有一个条目被放在了正确的位置上（事实上，如果你仔细想想，上面的*j*的循环可以缩短为`for (int j=0; j<N-i-1; j++)`，因为在i^{th}迭代之后，数组中的最后*i个*条目被排序）。

在一个未排序的数组上试试。该函数将产生一个按升序排序的数组，这是二进制搜索的要求，但是，计算成本如何？

如果我们要计算数组访问量，请考虑函数的嵌套循环结构。

外循环有`N`次迭代

内循环有`N-1`次迭代

内循环最多更新2个数组条目

所以该函数的复杂度是$N(N-1)2 = 2(N^2 - N) \rightarrow O(N^2)$。泡沫排序的复杂度是***N***阶^2（注意我们忽略了***N***项--为什么要这样做？）

即使我们使用*j*上的较短形式的循环，即只有*N-i-1*次迭代，这一复杂性结果也是成立的。结论是。

如果我们想用一个数组来寻找复杂度为***O(Log(N))的***项目
首先我们必须对数组进行排序，其复杂度为$O(N)^2$

突然间，使用排序数组看起来就不那么吸引人了--
排序数组的二次成本将完全抵消使用二进制搜索获得的优势。

注意事项。

- 如果我们的数据是一次性添加到集合中的（并且预计以后不会有插入或删除），那么我们可以认为，最初对数组进行排序的成本被*划分到*以后进行的*所有搜索操作中*。例如，如果*N=1000*，经过一次
百万次的查询，排序的额外成本归结为每次搜索***的***额外数据访问量

- 相反，如果数据不断被插入/删除，那么排序的成本就会超过二进制搜索所带来的节省。在这种情况下，我们倾向于使用链接列表。

等一下!奥巴马总统说过不要使用泡沫分类!

插图1："我认为泡沫排序将是错误的方式。"(图片：美国联邦政府--公共领域)

泡沫排序绝对***不是***最好的排序算法--
我们有时会对小数组使用它，因为它非常简单，实现起来也很快，但我们可以做得更好。如果你还记得我们上面关于***问题的计算复杂度的***讨论，最好的已知排序算法的复杂度为***O(N Log(N))***。有几种算法可以达到这个性能水平，包括明年你将在B63中详细研究的几种算法。现在让我们看看，如果我们用更有效的（平均）***快速排序来***取代我们昂贵的冒号排序，我们会得到什么。

快速排序的工作原理是在数组中选择一个条目（称为*支点*），然后用这个支点将数组分成*小于支点的*元素和*大于或等于*支点的元素。然后在每个子数组上重复这个过程。当所有的子数组都有零或单一元素时，这个过程就结束了（这时数据就被排序了）。排序后的数组由排序后的子数组构成。

有很多优化方法，而且这个过程可以在不使用子数组的额外空间的情况下进行（可以*在原地*进行）。也有很多方法来选择*支点*。对快速排序的分析超出了本课程的范围（你可能会在B63中看到它的细节），但重要的结果是

- 快速排序的平均复杂性为***O(N Log(N))***。
- 快速排序的最坏情况下的复杂度是$O(N^2)$。

上面两个语句的区别非常重要--
给定一个随机排序的数组，我们可以预期做快速排序的成本接近平均情况下的复杂度***O(N Log(N))***，这很好。然而，如果我们非常不走运（当我们挑选子数组中*最小*或*最大的*条目作为*枢轴*时），除枢轴外的所有条目最终都会出现在两个子数组中的一个中--
如果这种情况发生在过程的每一步，我们就会看到最坏的情况下的复杂度$O(N^2)$，与冒泡排序相同！

我们必须很不走运才能碰到最坏的情况，所以你会发现快速排序在实践中是非常常用的

。让我们假设，我们使用快速排序来保持我们的数组排序，然后依靠二进制

搜索来寻找信息。我们的问题解决了吗？

我们的搜索引擎现在必须做以下工作。

- 对集合进行排序，如果运气好的话，其复杂度为***O(N Log(N))***
- 使用二进制搜索，在集合中搜索项目，复杂度为***O(Log(N))***

像以前一样，如果我们假设数据是一次性添加到集合中的，并且以后没有插入或删除发生，我们可以在我们在集合上运行的所有搜索中分摊排序的成本。对于*N=1000，*现在只需要运行大约5000次搜索，就可以达到每次搜索的额外成本下降到*一个*额外的项目访问（相比之下，如果使用冒泡排序，我们需要100万次搜索）。

显然，这比使用冒泡排序要好得多，而且开始看起来像一个现实的解决方案。然而，在实践中，我们希望我们的数据会发生变化，会有插入、删除和修改现有项目的情况。如果我们在每次发生这种情况时都对数组进行排序，那么排序的成本就会再次主导搜索的总计算成本，而我们又回到了将常规的链表作为更好的解决方案。

那么，我们是否找到了一种有效的方法来使大量的收藏品可以被搜索到？

不完全是...

- 阵列方案有空间限制（固定尺寸可能不够，或者浪费空间）。
- 它需要进行排序。用最好的排序算法，其复杂度为***O(N Log(N))***，已经比*线性搜索*差了。
- 我们预计我们的集合会随着时间的推移而改变--
  插入、删除和修改将需要工作，以确保阵列保持排序。
- 结论是，*排序数组+二进制搜索*不一定是组织、存储和搜索大量且不断变化的项目集合的最佳解决方案。

我们需要什么？

- 很明显，为了高效搜索，我们需要分类的数据
- 我们不能以一种有效的方式保持数组的排序
- 而且我们希望链接列表能够按需请求和使用空间，所以我们想要的是

*一个*允许我们的*数据结构*。

- 按需申请空间，对藏品的数量没有固定限制
- 保持我们的数据排序，其*计算成本*比每次重新运行排序算法的*成本低*。
- 允许我们有效地搜索数据（理想情况下，与*二进制搜索*一样有效）。

我们现在将看到一个这样的数据结构，研究它的属性，并讨论它的一些应用。

5.- 树、二进制树和二进制搜索树

回顾一下，链接列表是一种数据结构，其中节点包含一个集合中的一个项目，以及一个指向后继节点的链接，该节点是列表中的下一个节点。

树是这个想法的概括，它由节点组成，每个节点包含一个或多个来自集合的数据项，以及*一个或多个与子节点的链接*（相当于继任节点，但现在我们可以有很多！）。

树是计算机科学中极为常见的结构，用于广泛的用途。

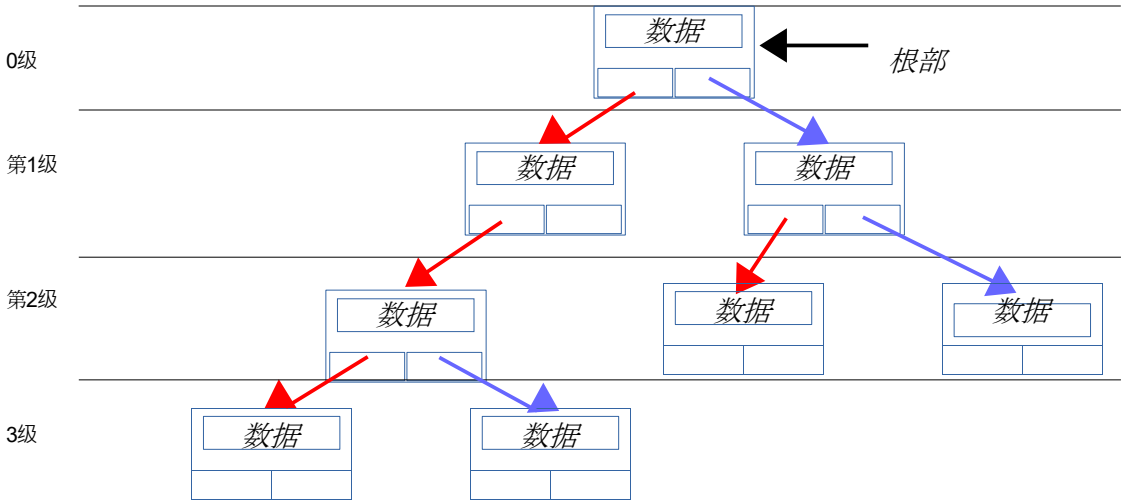
插图2：计算机图形学使用树形结构来创建、操纵和渲染虚拟的树木、植物和铰接模型，如动物。照片。Solkoll, Wikimedia Commons, 公共领域

一个特别常见的树的类型是*二叉树*，它的特性是每个节点都有*两个子节点*，*左边的子节点*和*右边的子节点*。

下图说明了一棵二叉树。请注意以下几点。

- 每个节点都有两个空间用于链接--
  一个用于左边的孩子（以红色箭头显示），一个用于右边的孩子（以蓝色箭头显示）。
- 节点可以有零个、一个或两个孩子

- 位于树顶的节点被称为***根节点***。与***链接列表的头部***类似，我们通过保留一个指向根节点的指针来管理一棵树。
- 树中的每***一级***都由与根节点有相同***距离***或***深度的***节点组成。每一层包含的节点数量最多是上一层的2倍。
- ***叶子结点***是***没有子节点的***结点。



二进制搜索树

二进制搜索树（BST）是二进制树，对于树上的每个节点，BST的特性是持有。

- 一个节点的***左边子树上***的***数据***的值***小于或等于***该节点的数据的值。
- 一个节点的***右侧子树上***的***数据***的价值***大于***该节点的数据的价值。

这意味着，在每个节点上，我们可以快速确定是否。

- 数据在当前节点中
- 数据不在当前节点中，但如果它在树中，它一定在左边的子树中
  - 或 -
- 数据不在当前节点中，但如果它在树中，它一定在正确的子树中

在这一点上，你应该考虑二进制搜索!***BST***的目的是让我们以这样一种方式来组织一个大的集合，使我们能够快速地搜索它。事实上，在假设我们的数据是以随机顺序插入树中的情况下，***BST***可以提供***O(Log(N))的***搜索复杂性。让我们来考虑***BST***中的搜索过程。

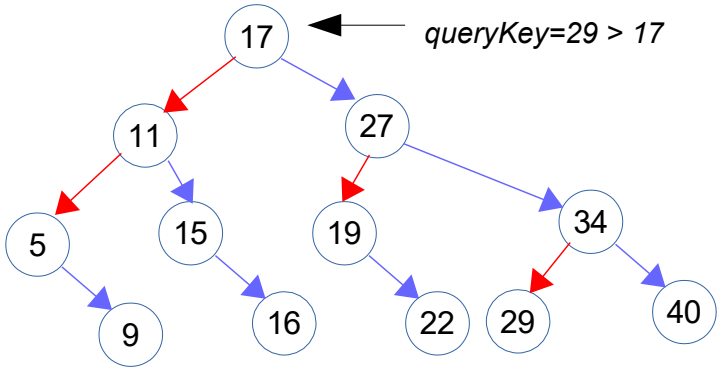
searchForKey ← subtree, queryKey

如果queryKey等于子树根节点上的key
找到钥匙了!返回一个对子树根节点的引用
否则
*如果queryKey**小于或等于**子树根节点中的key searchForKey ← left subtree, queryKey*
否则
searchForKey ←右侧子树，queryKey

搜索过程从树的顶端开始，将查询值与存储在某个节点的值进行核对。如果找到该值，则搜索成功，并返回对该节点的引用，否则，它将检查查询值应该在左边还是右边的子树中，并继续沿着该子树搜索。

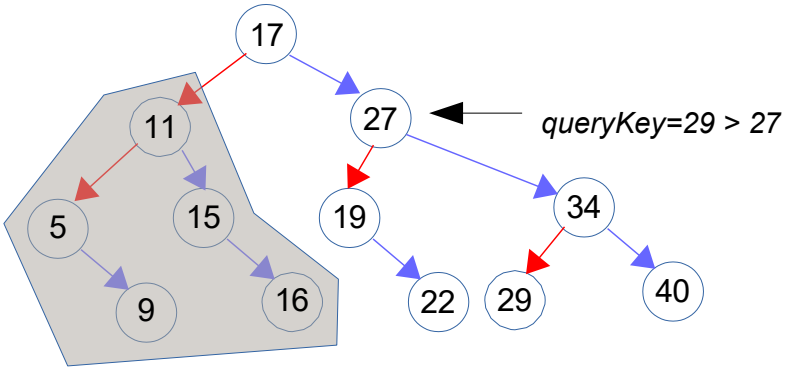
对于BST，我们通常把用于搜索树中信息的值称为***键***。如果树包含的数据项是简单的数据类型，那么***键***和***数据值***是一样的。但请记住，我们打算用这些数据结构来组织和维护***复合数据类型***的大集合。所以一般来说，***键***将是***一个合适的字段，或者是复合数据类型中的一个字段子集***。键必须是可比较的（也就是说，除了检查相等之外，我们必须能够根据一些对我们的数据有意义的排序来判断这两个键中哪个大，哪个小）。这通常涉及到编写一个与我们的数据类型一起工作的比较函数。

搜索过程如下图所示

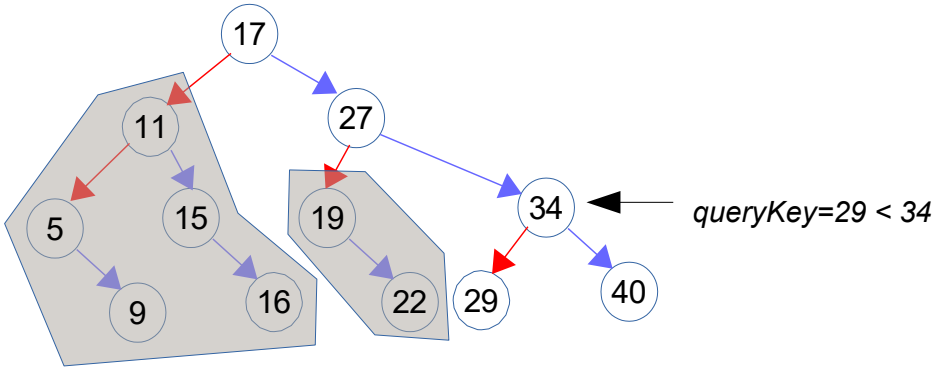


从根开始，*查询键*（本例中为29）与节点中的键进行比较。在这种情况下，*查询键大于*节点中的键，所以我们知道*查询键*，如果它在树中，*一定是在右边的子树中*。

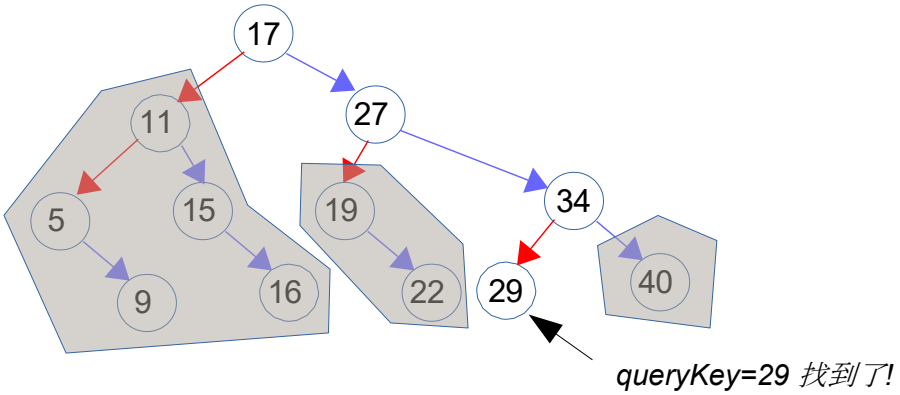
现在，搜索过程沿着右边的子树继续进行，这意味着***大约一半的剩余键被丢弃，不必再进行搜索了!***



queryKey大于节点(27)的键，所以我们知道我们想要的值*一定在右边的子树上*。沿着右子树继续搜索，*从搜索中放弃了大约一半的剩余键*。



在这一点上，*queryKey小于节点上的键*，所以我们知道如果它在树上，它*一定是在左边的子树上*，并继续我们的搜索。再一次，从搜索中抛弃了大约一半的剩余键。



我们已经找到了我们的*queryKey!*请注意，在最后的图表中，*树中的许多键被丢弃了*，这意味着搜索过程*在搜索过程中不必检查它们*。

你可以看到，*与二进制搜索*类似，***BST***中的搜索过程很快就会从

搜索存储在树上的所有值中的很大一部分。问题是***需要检查多少个项目，我们才能找到查询键（或确定它不在树上）***？

在***BST***中，*查询键*和树中的值之间的每一次比较，搜索都会向下移动一级。这意味着，搜索*最多*要检查*的*节点数量等于***BST****的深度*--树中的层数。

那么，***一个完整的BST有多少个级别？***

一个***完整的BST***是一个***BST***，其中每一层除了最后一层都是完整的（所有可用的节点都有数据，没有未使用的节点）。最后一层可能是满的，也可能不是满的，但是如果它不是满的，那么键就会向树的左边挤。

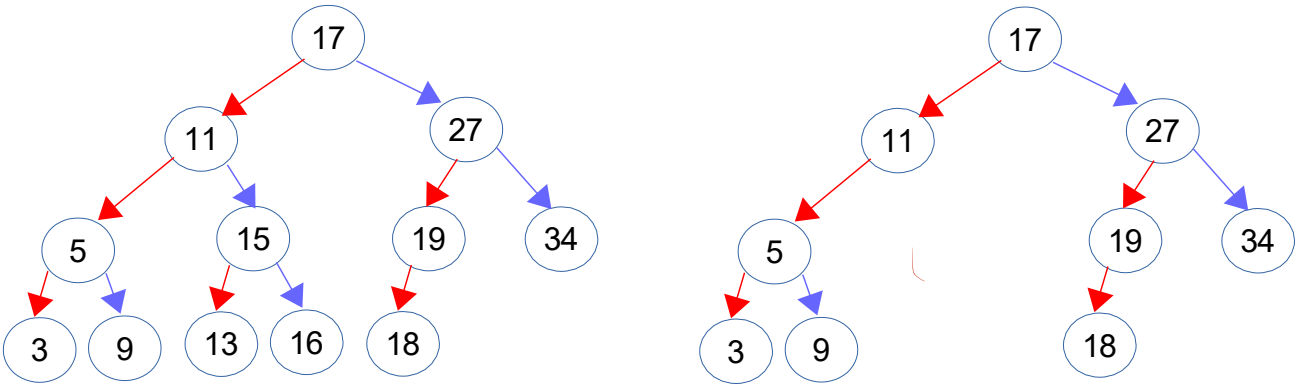


图2：左图：一个完整的BST，0-2层已经完成，最后一层的所有钥匙都挤在左边。右图。不是一个完整的BST，第2层缺少钥匙，最后一层的节点没有向左排列。

在一个***完整的BST***中，每一级包含的***节点与上面所有级别+1一样多***。例如，第2层包含4个节点，而树中第2层以上有3个节点。第4层包含16个节点，0-3层共包含15个节点。这意味着我们在***BST***中每增加一级，*其存储密钥的能力就增加一倍。*

因此，一个具有***k级的BST***可以存储多达***N=2个^k***，因此，对于一个具有***N***键，我们需要一个至少有***对数$_2$（N）***的***BST***。

一个存储***N***个键的***完整BST***的高度是 $\lceil Log_2(N+1) \rceil$ (时髦的括号代表最高函数）。因此，在这样的树中搜索一个键，其复杂度为***O(Log(N))***。与二进制搜索相同!

不幸的是，BST一般***不会是满的***，所以我们上面得到的***O(Log(N))***搜索复杂度值并不是全部的情况。然而，为了理解为什么会这样，我们必须实现***BST***所支持的基本操作。

对BST的操作

*二进制搜索树*必须至少支持以下操作。

- 初始化一个空的二叉树
- 将一个节点插入***BST***中
- 移除（删除）***BST***中的一个节点
- 在馆藏中搜索一个特定的项目

有时会定义额外的操作，但它们往往是上述操作的特例或组合。

6.- 实施BST

让我们改进一下我们的*Kelp*应用程序的设计，用***BST***代替我们用来跟踪餐厅评论的链接列表（我们希望通过这样做，评论的查找会更快）。作为提醒，存储实际评论的 *"评论"*复合数据类型的定义并没有改变。

```
Typedef struct Restaurant_Score
{
      char restaurant_name[MAX_STRING_LENGTH];
      char restaurant_address[MAX_STRING_LENGTH];
      int score;
}评论。
```

现在不同的是，将在我们的***BST***中存储评论的 ***节点的***定义。事实证明，***BST***节点很像一个链接列表节点，但有两个指针而不是一个。这些指针分别对应于节点的***左右两个***子节点。

```
typedef struct BST_Node_Struct
{
      审查；                                  //存储一个审查
      结构 BST_Node_Struct *left;        // 一个指向其左边孩子的指针 结构
      BST_Node_Struct *right; // 和一个指向其右边孩子的指针
```

如果你想为不同的数据类型创建一个***BST***，你只需要改变上述节点定义中的数据组件。

与链接列表一样，我们需要写一个函数来*分配和初始化一个新的'BST_Node'*。

```
BST_Node *new_BST_Node(void)
{
      BST_Node *new_review=NULL;        //新节点的指针


      new_review=(BST_Node *)calloc(1, sizeof(BST_Node)) 。
```

```
        //初始化新节点的内容（与链接列表相同） new_review->rev.score=-1;
        strcpy(new_review->rev.restaurant_name,");
        strcpy（new_review->rev.restaurant_address,")
        new_review->left=NULL;
        new_review->right=NULL

        返回new_review。
}
```

除了名字之外，上面的函数和我们在链接列表节点中使用的函数之间的唯一区别被红框标出。我们现在有两个指针需要被设置为*NULL*。

将节点插入BST中

对于链式列表，我们不得不保留一个指向*列表头部的*指针。在***BST***的情况下，我们将需要保留一个指向树根（顶部的节点）的指针。插入过程***必须确保在树中插入一个新的键时，BST属性被强制执行***。

这意味着树中的键被插入，以便对于树中的任何节点，左子树中的键小于或等于该节点中的键，右子树中的键大于该节点中的键。由于我们的***BST***中的数据是一个复合类型*"Review"*，我们需要定义哪个字段作为***键***。让我们使我们的***BST***保持我们的评论按餐厅的名字排序。因此，在下面显示的***BST***的实现中，每当我们谈论一个给定节点的***键时***，我们实际上是在谈论***BST***节点的"*评论*"组件的"*餐厅名称*"字段。

让我们看看插入函数是如何定义的，并看看它如何工作的例子。它接受两个参数作为输入。一个指向我们要插入新节点的***子树根***的指针，以及一个指向树中要插入的*新评论的*指针

```
BST_Node *BST_insert(BST_Node *root, BST_Node *new_review)
{
        if (root==NULL)                 //树是空的，新的节点成为 return
                new_review;             // 根。

        // 确定新的键应该在哪个子树中 如果(strcmp(new_review-
        >rev.restaurant_name,\\)
                        root->rev.restaurant_name)<=0)
        {
                root->left=BST_insert（root->left,new_review）。
        }
        否则
        {
                root->right=BST_insert（root->right,new_review）。
        }
```

上面的函数中发生了什么？

它接收一个指向BS*T_Node的*指针，该节点是***BST***的***根***。请记住，***BST***的任何子树也是一个***BST***!因此，许多不同的节点都可以被认为是***根节点***，这取决于你所看到的***BST***的哪一节。下图说明了这一点。

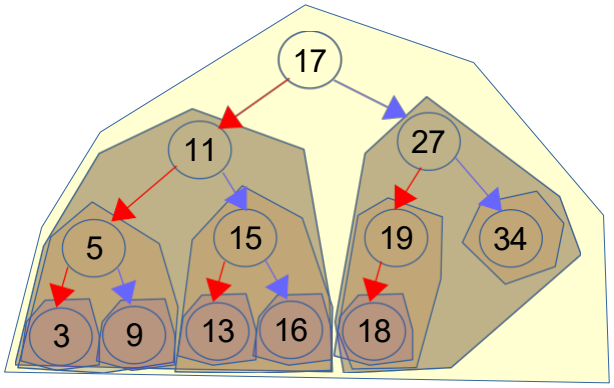
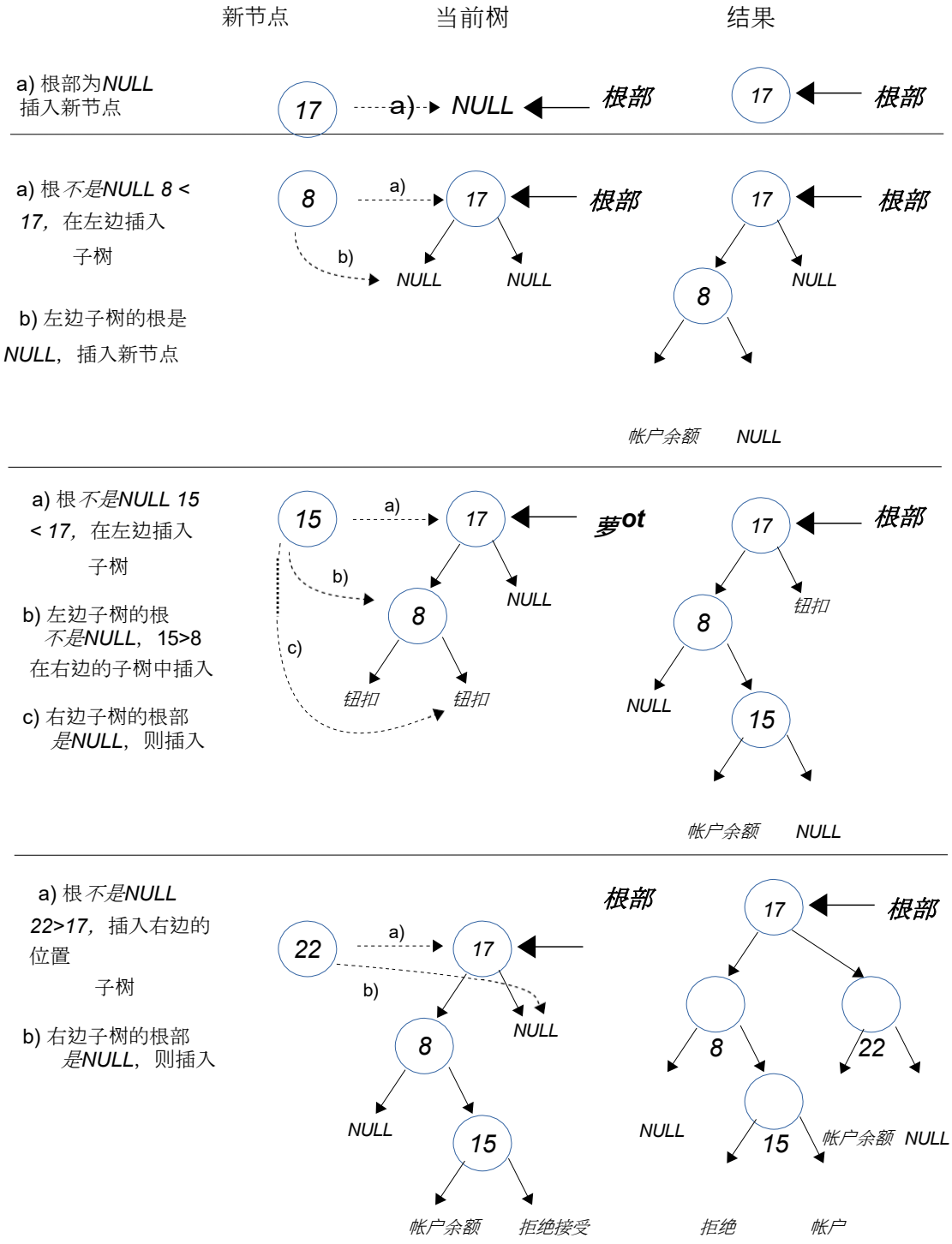


*图3：**BST**中的每个子树也是**BST**。上图显示了一个样本**BST**中的不同子树（扎根于不同的节点）。请注意，每个叶子节点都有**两个空的子树**作为子女。*

给出某个BST的***根***（可以是整棵树或其中的一个子树），插入函数检查***根***是否***为空***。在这种情况下，树是空的，我们要插入的节点成为根。

如果***根***不是***NULL***，（子）树就不是空的，所以插入函数决定新的评论应该在***左边的子树***还是***右边的子树中***，并继续在相应的子树中插入评论（使用相应的节点作为***根***）。最终，插入函数会找到一个空的分支来插入新的节点，因为在每一步，它都会根据新评论的'*餐厅名称*'与树上其他评论的比较来测试什么是正确的子树，新的评论会被插入到正确的位置以保持***BST***中键的严格排序。

下图显示了一个只包含整数值的***BST的***插入过程的几个例子。这个过程与我们包含餐厅评论的***BST是***一样的，只是对于评论，我们比较的是餐厅名称而不是整数。



*练习。*从上图中最后一个BST开始（插入22后），列出所进行的步骤，并画出插入后的结果树。

19、4、1、6和21（按顺序）。

重新审视BST中搜索的复杂性

我们知道在BST中搜索钥匙的复杂度取决于树的高度，我们也知道在一个***完整的BST***上，树的高度是 $\lceil Log_2(N) \rceil$
然而，给定随机排列的钥匙依次插入***BST***中，我们最终会得到一个完整的***BST***的情况很少。***BST的形状，以及它的高度，都取决于键的插入顺序***。

如果键的顺序或多或少是随机的，我们可以预期插入将或多或少地在树的任何一级的左和右子树之间平均分配。然而，如果键的顺序不是随机的，我们很快就会遇到麻烦。

练习。画出依次插入下列键的BST。1, 2, 3, 4, 5, 6, 7, 8, 9, 10

问题。结果BST的高度是多少？概括到一组*N个最初排序的键*依次插入树中，BST的高度将是多少？

问题。关于***BST的最坏情况下的搜索复杂性***，你能从上面推断出什么？

问题。按排序顺序插入键所产生的BST的形状让你想起了什么？

对***BST***的搜索复杂度的全面分析超出了我们这里的范围，你可能会在明年的B63中仔细研究。但是现在，你应该记住，我们讨论的***BST的O(Log(N))***搜索复杂性只是***平均情况下的复杂性***。在假设键在插入树中时是随机排序的。***最坏情况下的复杂度***完全是另一回事，当我们决定是否使用***BST***来存储和有效地访问大量的项目集合时，应该牢记这一点。

在继续讨论***BST***上的其他操作之前，值得指出的是，有几种类型的树，如***AVL树***、***B树***和***2-3-4树***，其特性是无论键的插入顺序如何，它们都***保持平衡***。这样的***平衡树保证了***在所有条件下，***搜索***、***插入***和***删除***操作的复杂度仍然是***O(Log(N))***。你将在B63中了解这些树，它们如何工作以及它们如何保持平衡的形状。

在BST中搜索

BST中的搜索操作是寻找一个特定的***查询键***。一旦找到指定的***查询键***，就会返回一个指向包含该键的节点的指针。对于存储复合数据类型的BST，搜索操作必须知道复合数据类型的哪个字段作为键。

让我们看看这是如何运作的。

```
BST_Node *BST_search(BST_Node *root, char name[1024] )
{
      // 按餐厅名称查询餐厅评论

      如果（root==NULL） 返回NULL。          // 树或子树是空的

      // 检查该节点是否包含我们想要的评论，如果是，则返回
      //一个指向它的指针
      如果（strcmp(root->rev.restaurant_name, name)==0）返回
            root。

      // 不在这个节点中，搜索相应的子树 if (strcmp(name, root-
      >rev.restaurant_name)<=0)
      {
            返回BST_search(root->left,name)。
      }
      否则
      {
}
```

其实现几乎与插入操作相同。它在当前节点中寻找与查询相匹配的键，如果没有找到，就根据查询与当前节点中的键的比较情况搜索相应的子树。当我们找到我们想要的键，或者达到一个空的分支时，搜索就结束了。

在这一点上，谈论如何处理***重复的键***变得很重要。***BSTs***的定义并不禁止树中存在重复的值。然而，如果存在重复值，搜索的行为就变得不好定义了--
这是因为没有原则性的方法来决定用户到底想要哪一个重复的节点（这也是我们在上一单元中的链接列表的问题！）。

问题。假设我们对同一家餐馆有多个评论。其中*哪些*会被上面定义的搜索功能所返回？

由于重复键的存在所带来的模糊性，***数据库依赖于***对数据项使用***独特的标识符***。你经常使用的例子包括***学生号***、***社会保险号***、***产品条形码***等。这些都是为集合中的每个项目生成的，并与之相关联，以确保我们总是有一种方法来识别任何特定的条目，而不会产生歧义。

如果没有生成唯一标识，通常可以从项目信息的***数据字段组合***中创建一个。例如，对于一个音乐记录的集合，我们可以通过检查***以下所有字段是否符合查询要求来***唯一地识别任何一个给定的记录。标题、艺术家、标签、年份。

数据库主键具有唯一性的约束，在我们的BST（或链接列表，或任何其他旨在支持集合搜索的数据结构）中，我们必须强制使用唯一键来识别项目。你将在数据库课程C43中了解所有关于如何设计和使用主键的知识。

非常重要的是。 一旦我们确定了我们将使用什么作为唯一的键来定位我们的集合中的项目，插入操作必须强制保证***没有重复的记录被添加到BST中***。这意味着，如果我们试图插入一个其唯一键与已存在的项目相匹配的项目，插入操作将不会被执行。

更新集合中的一个项目

更新操作是简单的搜索，然后对项目中所需要的数据字段进行更新。***然而，*** 应该注意的是，***不允许对作为***项目***关键的数据值进行更新***。更新键可能意味着键在树内排序的BST属性将不再成立，因此会破坏搜索、插入和删除操作。

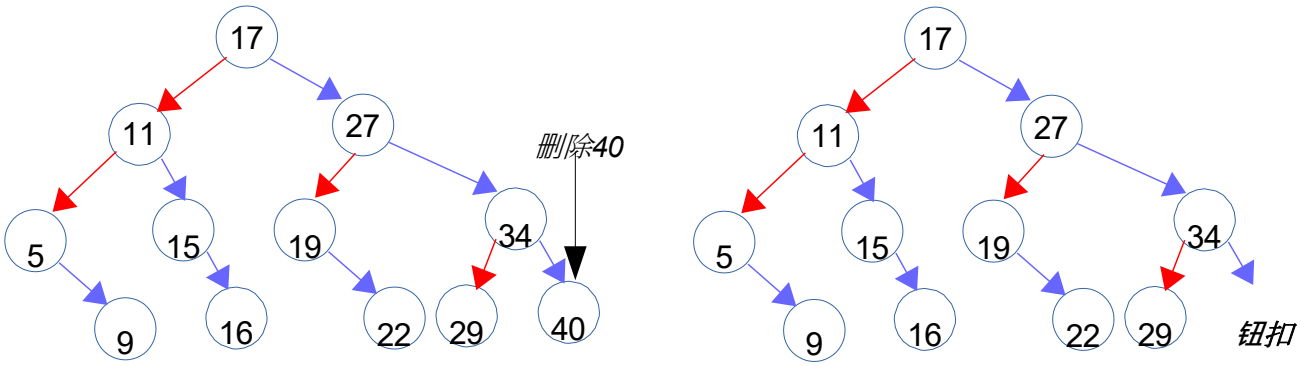
从BST中删除项目

*删除*操作比*搜索*和*插入*都要有趣一些。在一个链接列表中，插入涉及到寻找要删除的项目和它的前任，然后链接前任和后任节点以保持列表的链接结构--然后删除所需的项目节点。

对于BST来说，情况更为复杂，因为我们必须确保在删除之后，BST属性在树的各个地方仍然成立。

有三种情况我们必须考虑删除。

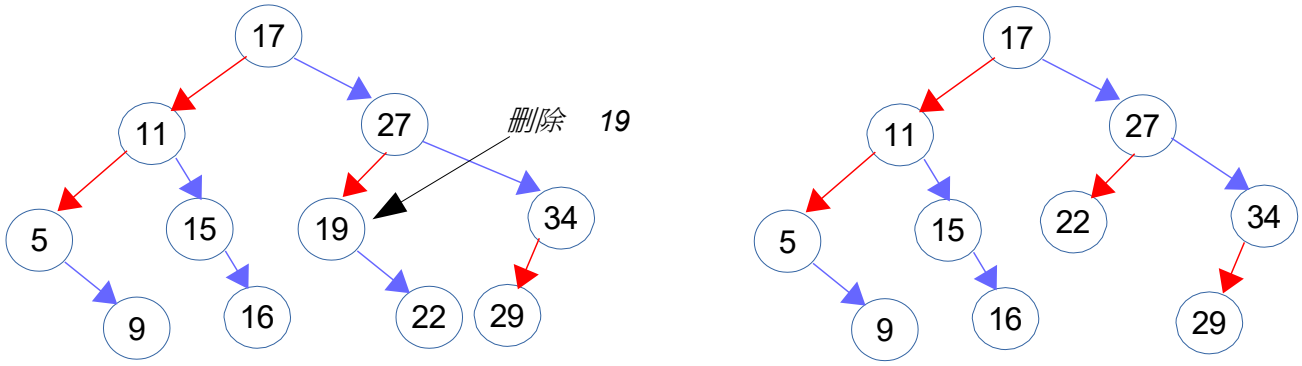
a) 要删除的项目是一个叶节点（没有子节点）。在下面的例子中，假设我们想删除包含'*40'的*节点。



我们需要做的就是把对应的paretent节点的链接设置为***NULL***。在上面的例子中，*'40'是'34'的右侧子节点*，所以我们将相应的链接设置为***NULL***，并删除（释放内存）。

含有*'40'的*节点。

b) 要删除的项目***只有一个孩子***。在这种情况下，我们只需将节点的子节点链接到节点的父节点，然后删除该节点。在下面的例子中，删除*'19'*意味着我们将*'22'*链接到*'27'*，然后释放*'19'*这个节点所使用的内存。



前两种情况可以通过以下代码来处理。

```
BST_Node *BST_delete(BST_Node *root, char name[1024] )
{
	// 删除一个名字与查询相符的餐厅的评论。
	// 假设有独特的餐厅名称!


	如果（root==NULL） 返回NULL。		// 树或子树是空的

	// 检查这个节点是否包含我们要删除的评论，如果（strcmp(name,root-
	>rev.restaurant_name)==0）。
	{
		如果(root->left==NULL &&&root->right==NULL)
		{
			//情况a），没有孩子。父方将
			//被更新为有NULL而不是这个
			//节点的地址，我们删除这个节点free(root)。
			返回NULL。
		}
		否则，如果(root->right==NULL)
		{
			//情况b)，只有一个孩子，左子树
			// 父级必须被链接到左边
			// 这个节点的子节点，我们释放这个节点，tmp=root-
			>left。
			free(root);
			return tmp;
```

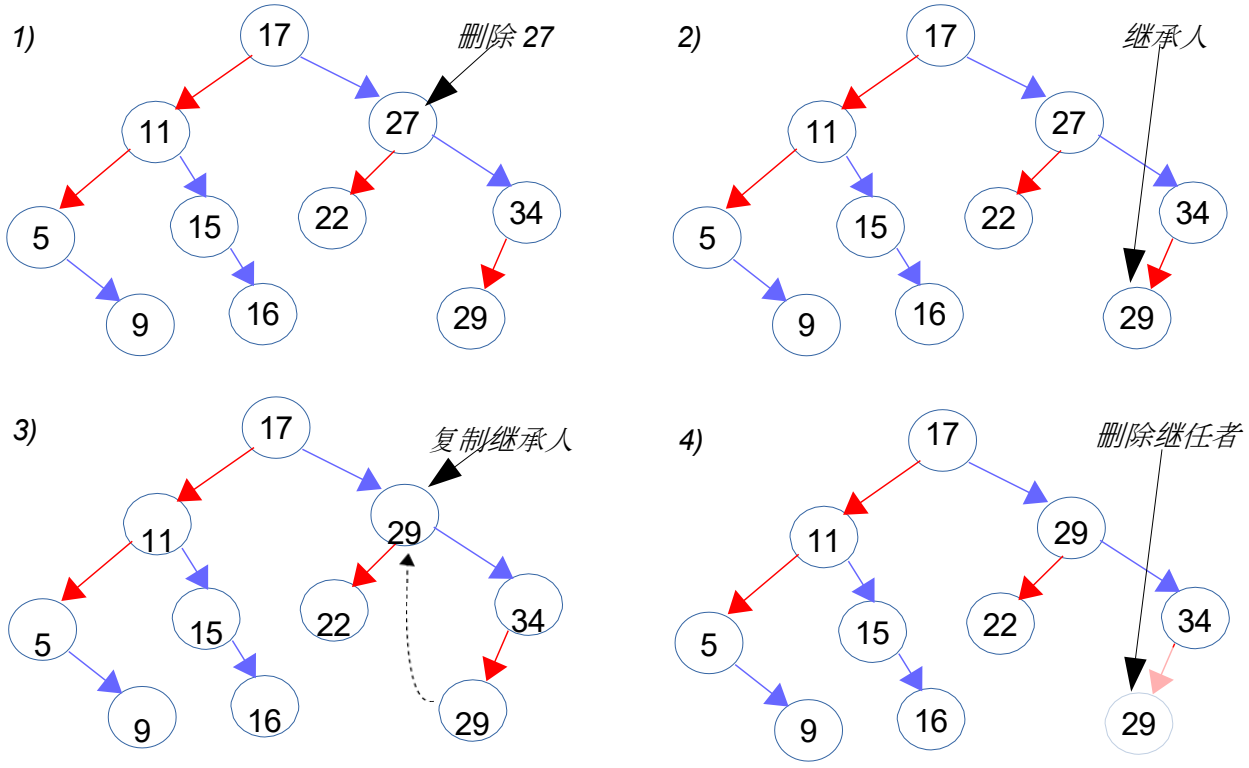
```
                //情况b)，只有一个孩子，右子树
                // 父体必须被链接到右边
                // 这个节点的子节点，我们释放这个节点，tmp=root-
                >right。
                free(root);
                return tmp;
        }
        否则
        {
                //情况c），两个孩子。
                // 你将为你的练习实现这一点！返回根。
        }
}

//不在这个节点上，在相应的子树上删除，并
//更新相应的链接
如果(strcmp(name, root->rev.restaurant_name)<=0)
{
        root->left=BST_delete（root->left,name）。
}
否则
{
```

现在，对于最后一种情况。

c) 要删除的节点有*两个子节点*。这就有点难度了。当我们要删除的节点有两个子节点时，删除的工作方式如下。我们找到我们要删除的节点的*继承者*--
这是一个*键值在我们要删除的节点中依次排列的节点*。另一种思考方式是：*继任者是右边子树中最小的键*。

一旦确定了继任者。
- 将*继承节点*的*数据复制到我们要删除的节点*上。
    - 如果节点包含一个复合数据类型，复制整个复合数据类型
- 然后*从右边的子树中删除继任节点。*
    - 这又可以是上述a)、b)或c)中的一种情况。



正如你所看到的，实际存储'*27*'的***节点***没有去任何地方，我们不需要重新链接任何东西！我们所做的是用继承者的数据替换节点中的数据。我们所做的是用*继承者的数据*替换节点中的数据，然后删除继承者的节点。

练习。想一想，依次删除下面的键后，树会是什么样子（在每次删除后画出树的样子）。

9, 22, 29, 11

练习。完成上述*BST_delete()*函数的实现，实现案例
c). 你需要写代码来找到继承者，这样你就可以把它的数据复制过来，这样你就知道该从子树上删除什么。

树形遍历

我们还需要讨论一组对***BST***的操作--
这些操作被称为*树形遍历*。树状遍历包括*按照预先指定的顺序访问**BST**中的每个节点*。我们使用树形遍历的目的有很多，包括

- 打印***BST***中节点的数据
- 计算我们收集的数据的汇总统计数据

- 当更新适用于所有节点时，更新信息（例如，假设我们的***BST***包含一个商店的商品价格，并且有一个全店的销售，折扣适用于所有东西）。

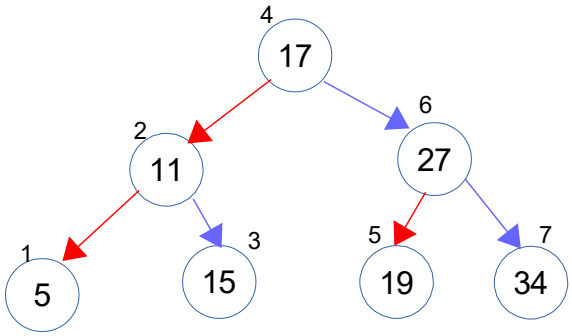
- 一旦我们的程序完成，删除***BST***

主要有三种类型的遍历，以访问节点的顺序来区分。

- 顺*序内遍历*，这可能是最常见的，它规定对任何给定的子树来说，树的节点都是按这个顺序访问的。

1) *依次*遍历左边的子树
2) 根部 - 在节点上应用所需的操作
3) *依次*遍历右边的子树

下面的例子显示了树中的节点在依次遍历时的访问顺序。



如果在每个节点上，我们进行的操作只是打印该节点上的键，那么内序遍历将产生。

5, 11, 15, 17, 19, 27, 34

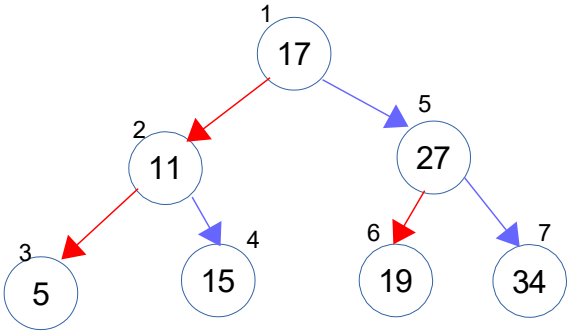
这给了你一个*BST中的键*的*有序列表*。因此，该遍历的名称是。

练习。写一个函数，按餐厅的字母顺序打印餐厅的评论名称。

- ***前序遍历，***对于这种类型的遍历，节点被访问的顺序是由。

1) 根部 - 在节点上应用所需的操作
2) 以*预设顺序*遍历左边的子树
3) 按*预设顺序*遍历右子树

下面的树显示了节点将被*预先*访问的顺序。



这一次，如果在每个节点的操作是打印钥匙，我们将得到以下列表。17, 11, 5, 15, 27, 19,

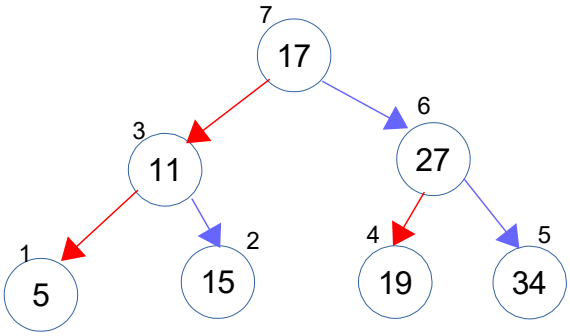
34

这似乎不是一个非常有趣的列表，然而，***前序遍历***在编译器和自然语言理解等应用中非常有用，在这些应用中，一大块代码的结构、一个序列或数学运算、或句子中的说话部分由树状结构表示。如果我们想创建一个***BST***的副本，这也是我们需要使用的遍历顺序。

- ***后序遍历，***指定树中的节点必须按以下顺序访问。

1) ***以后置顺序***遍历左边的子树
2) ***以后序***方式遍历右边的子树
3) 根部 - 在节点上应用所需的操作

下面的树显示了***后序遍历所***要访问的节点的顺序。



通过***后序***遍历，我们会得到以下的钥匙列表。5, 15, 11, 19, 34, 27,
17

再一次，这个列表看起来并不特别有趣，但是***后序遍历***在解析中有所应用，而且也是我们在*删除BST*时需要访问节点的顺序。

*练习。*画出依次插入下列键的树。57, 25, 69, 16, 35, 59, 71, 72, 73, 9, 3, 14, 99

然后按顺序列出条目，这些条目将由*顺序内*、*顺序前和顺序后的遍历*产生。

*练习。*写一个函数来删除餐厅评论的***BST***，确保所有分配给***BST***节点的内存被释放。

*练习。*在main()中加入一个交互式循环，允许你输入新的评论，搜索特定的评论，或者删除现有的评论，从而完成你对一个小餐馆评论数据库的实现。

7.- 我们的问题解决了吗？

在这一点上，我们对***BST***能做什么，以及如何实现它们有了很好的了解，但我们还没有确定所有这些工作是否帮助我们解决了寻找一种更有效的方式来存储、组织和访问大量信息集合的问题。我们可能怀疑***BST***有可能给我们带来更快的搜索，因为它的平均搜索复杂度比链接列表的平均搜索复杂度要好，但我们仍然要把一些松散的东西绑起来。

a) 构建***BST***所需的时间与构建一个链表所需的时间相比

- 对于***N个***项目，可以在***O(N)***时间内建立链表--
  每个项目都是在头部添加的，所以不需要遍历。这是非常好的!
- 对于同样的***N个***项目，BST ***平均***可以在***O(N Log(N))***时间内建立，因为插入总是发生在***树的底部***，而且树的高度是***O(Log(N))***。

优势。链接列表

b) 建立***BST***所需的时间与对数组进行排序所需的时间（因此我们可以使用二进制搜索）。

- 使用*quicksort()*可以在***平均O(N Log(N))***时间内对一个数组进行排序。
- 正如我们在上面看到的，***BST平均***可以在***O(N Log(N))***时间内建立。

优势。无--在复杂程度上打成平手

c) 搜索复杂性

- 排序的数组。***O(Log(N))最坏情况下***使用二进制搜索。
- 链接列表。***O(N)的平均和最坏情况***。
- ***BST***：***O(Log(N))平均值***，***O(N)最坏情况***。

*优势。平均来说，**BST**和排序数组，**最坏的情况***是排序数组。

d) 存储使用

- 排序的数组。固定大小，不适合增长/缩小的集合
- 链接列表。空间使用是***O(N)*** - 每个项目一个节点
- ***BST***: 空间使用量为***O(N)*** - 每个项目一个节点

*优势。**BST**和**linked-list***，它们只对实际添加到集合中的项目使用空间。

那么，我们该如何选择？

通过这些工作的意义在于帮助你看到，在选择如何存储和组织数据集合时，有相当多的因素需要考虑。到目前为止，我们有3种主要的存储方式：数组、链接列表和***BST***，如上所示，每一种都有优点和缺点。那么，我们该如何选择呢？

a) 考虑到你的收藏品要有多大。

- *对于非常大的集合*，你要选择能给你带来最佳*大**O**的*数据结构。
对于像*插入*、*删除*和*搜索*这样的常见操作来说，其复杂程度可想而知。

-
考虑到空间要求。如果项目有较小的内存需求（例如，只是数字数据，如ints或floats，或小的复合数据类型），预先分配一个大的数组并不是不合理的，即使其中的条目可能不被使用。另一方面，如果你的项目有很大的内存占用（例如，有很多数据的复合数据类型，或者像图像、声音片段等多媒体内容），你肯定希望有一个数据结构，*只为你在给定时间的集合中的项目*分配内存。

- *对于较小的集合*，这将取决于你是否倾向于实现的简易性与你的数据结构在普通操作中的性能。

b) 考虑将对集合的项目进行什么样的操作。

- 如果你主要是对整个集合进行操作，可以选择一个链表或数组（它不一定要排序）。这方面的一个例子是，在 "三维点网格 "中使用的
计算机图形学，可能包含数百万个点，但我们不进行单个点的查找，而是渲染整个网格。这些通常被保存在数组中。

- 如果*搜索*、*插入*或*删除*会很频繁，那么你应该倾向于选择一种能给你带来最佳*大**O***复杂性的数据结构来进行这些操作。

你不必满足于*一种数据结构*。数据库*以表的*形式组织大量的信息--
这些基本上是巨大的数组，其中每一行都包含集合中一个项目的所有数据字段。它们被存储*在磁盘上*，并且*没有排序*。与包含实际数据的表分开，数据库以树的形式保存*索引*（通常是***B-树***或其变体），其中包含用户要运行查询的*搜索键*。*索引*中的每个节点都包含*原始表中的*位置，在那里可以找到与查询键匹配的项目信息。

这个方案的另一个优点是，数据库可以为同一个表设置多个索引。例如，对于ACORN中的学生记录，可以有一个按学生编号组织的索引，和一个按姓氏/名字组织的索引。每个索引都允许数据库根据不同的常用查询词有效地搜索记录。

*永远记住。*每当你设计一个存储、组织和访问信息的解决方案时，你必须考虑集合的大小，如何使用它，在它上面运行什么操作，然后思考你所知道的所有ADT和数据结构，它们对于常见操作的复杂度等级，以及你如何使用它们来构建最有效的数据存储和操作解决方案。***期待在B63中学习更多关于存储、组织和搜索信息的方法***，包括平衡树、哈希表、堆、图和这些的组合。

这一切在实践中是否有效？

值得一看的是，我们在这里学到的东西在实践中是否真的有效。下面的图表比较了我们的三种存储方案在组织和搜索一个大的整数集合的任务中的表现。我们的三种方法是。

- 一个排序的数组+二进制搜索
- 一个链接列表
- A ***BST***

我们将在规模越来越大的集合上测试它们，在每个集合上，我们将对随机选择的键运行10,000次查询（以检查这些数据结构的搜索性能如何比较）。

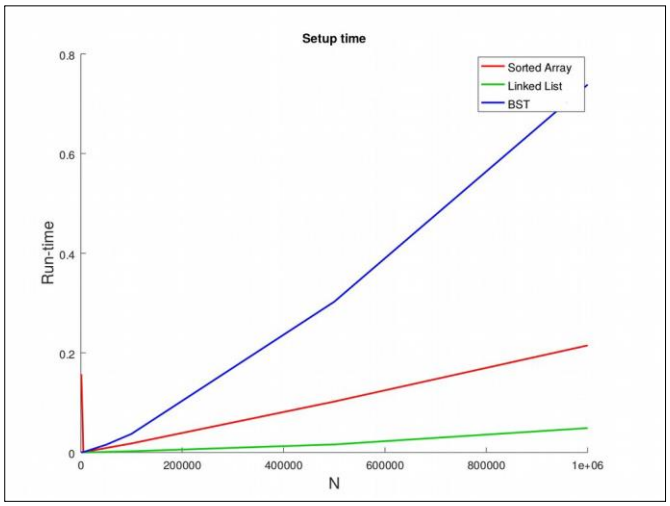
你会记得，对***BST***的一个担心是，*搜索*、*插入*和*删除*的最坏情况下的复杂性是***O(N)***。重要的是要弄清楚，在不同的项目集上，数据结构在多次测试中的性能是否真的会发生很大的变化，以至于我们在决定使用哪种结构时*不应该依赖平均情况下的复杂度*。为了研究这个问题，我们将对每个集合的大小进行50次测试，每次测试都有不同的随机生成的键。这是一个相对较小的样本，但它至少应该给我们一个指示，我们的数据结构在不同的输入数据集上的表现是否符合我们的预期。

最后，我们以三种不同的方式展示运行时间。

- 设置时间。将数据插入集合的时间（对于数组，这包括运行quicksort()的时间）。
- 搜索时间。一旦集合被存储，运行10,000次查询所需的时间
- 总时间。两者的总和，对每个数据结构的执行情况有一个完整的了解。

关于下面的图表的重要说明。请非常注意--
X轴不是按比例绘制的！*X*方向上的每一个刻度都对应着***比前一个刻度多10倍的项目***。如果我们把图表按比例显示出来，它就会溢出页面，跑到隔壁的房间里去!在你思考图表所显示的内容时，请牢记这一点。

设置时间。在数据结构中插入***N个***项目所需的时间--
对于数组，这包括quicksort()所花费的时间。



该图显示了所有三种数据结构的设置时间。正如我们在上面的分析中所预期的那样，链接列表具有优势，其设置时间仅为***O(N)***。排序数组和***BST的时间***都明显更长。

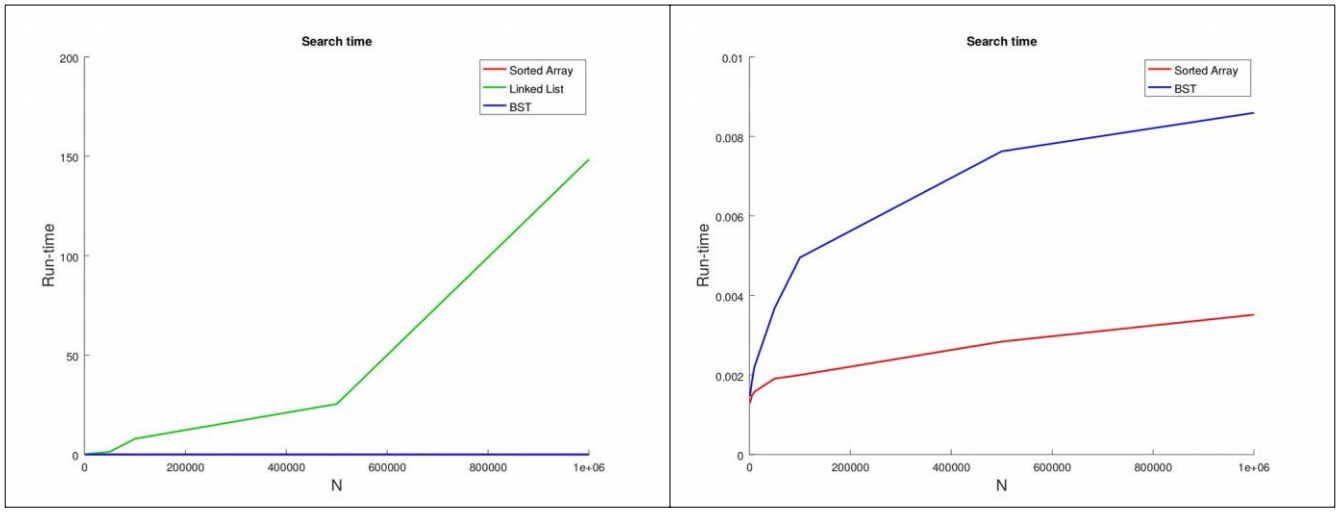
结论。实践中的设置时间与我们在上面几节中对每个数据结构行为的分析一致。

疑问。在最小的项目数上，排序数组的设置时间有一个奇怪的峰值（它不是在零点，那只是X轴的比例的影响）。有什么可以解释这种奇怪的行为？

搜索时间

下面，在左边，你可以看到在每个数据结构上运行10,000次搜索的时间。在50次不同的运行中，以不同的随机键排序，取平均值。正如你所看到的

很容易看出，链表的性能比数组和***BST***差很多。这也是预料之中的，链表的平均***O(N)***搜索复杂度无法与***BST***的平均***O(Log(N))***搜索复杂度和二进制搜索的保证***O(Log(N))***搜索复杂度竞争。右边的图只显示了***BST***和排序数组的搜索时间，以供比较。



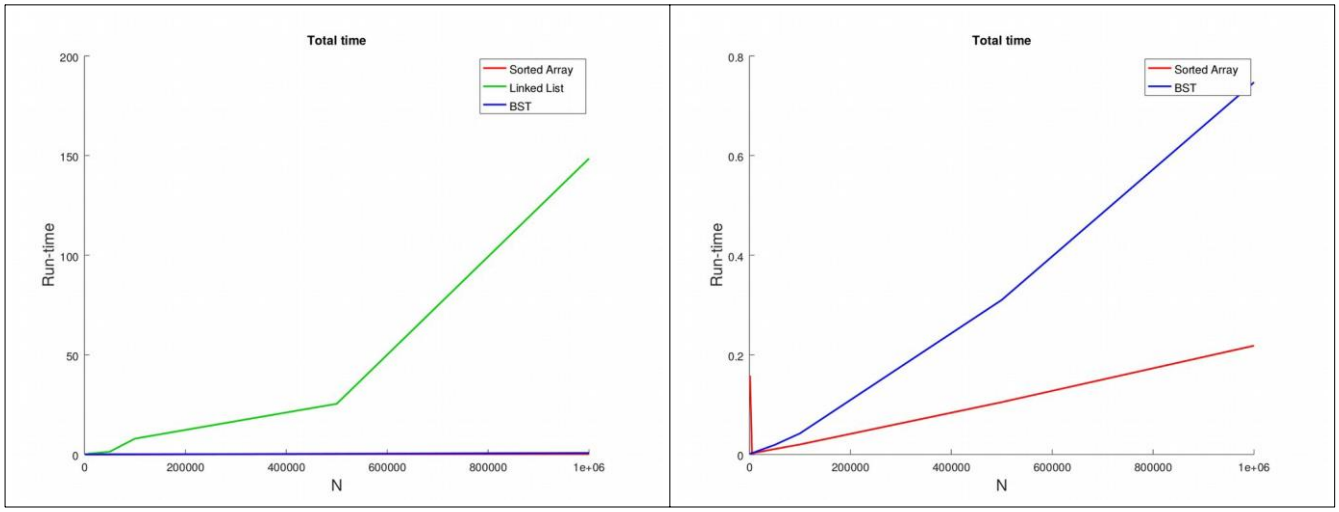
上面的图表显示了与我们预期一致的行为。***BST***和排序数组的表现相似（运行时间的曲线形状看起来相当接近，除了一些与管理***BST***和数组的开销有关的恒定因素）。我们还可以看到，***BST***的性能似乎没有显示出任何减缓的迹象，表明达到了搜索复杂度的最坏情况***O(N)***（在这种情况下，我们应该期望它向上述图中的链接列表的绿色曲线移动）。

结论。实践中的搜索时间与我们在章节中对复杂性的分析一致
以上。

总时间

下图显示了每个数据结构进行10,000次查询的总时间。它是设置时间和搜索时间的总和
。

左边的图表再次显示了所有三种数据结构的总时间，并清楚地表明，如果我们期望在我们的集合中进行大量的搜索，链接列表***不是***首选的数据结构。***BST***和排序数组的表现都很好--相似的曲线，其差异可能非常接近于一个常数，这是因为维护和遍历***BST的***开销，而数组中的任何条目几乎都可以立即访问。



还有一个观察结果。总时间曲线的形状表明，***BST***和排序数组的***总时间至少在***我们对数据库进行查询的数量上是***由设置时间主导的***。这并不令人惊讶。分类阵列和BST的设置时间都是***O(N Log(N))***（对于***BST***，请记住，这只是***平均时间***）。另一方面，搜索时间只有***O(m Log(N))***，其中***m***是我们在数据结构上运行的查询次数。由于***m=10,000***，搜索时间预计将小于最大集合的设置时间。

结论。总的搜索时间在实践中与我们对前几节的数据结构行为的分析一致。

最后的想法

我们已经花了一些时间仔细研究了如何使大型集合的搜索更有效率的问题。在这一过程中，我们研究了对算法、问题和数据结构的***复杂性进行***测量、描述和推理的问题。我们已经了解到，当我们实现这些解决方案时，我们从研究一个给定的解决方案中可以得出的复杂性结果可以清楚地直接转化为运行时间，我们已经思考了到目前为止我们所研究的不同的***ADT***应该如何使用以及何时使用。

请记住，对复杂性的理论分析只是找出执行某些任务的最有效方法这一复杂问题的一个层面。如果你真的想写出最好（最有效、最快）的软件来解决任何给定的问题，你需要学习计算机结构、现代CPU如何工作，以及如何优化程序以获得最大效率。当你学习计算机组织（B58）、算法和数据结构（B63）以及嵌入式系统（C85）时，不要忘记牢记你在这里学到的知识，因为这三门课程将使你对这一重要问题有更深的了解。

至于***最坏情况下的复杂度***。上面的分析可能会让你相信，如果你的数据结构具有良好的***平均***复杂度，你就不必担心这个问题。事实上，对于大多数应用

你会发现你可以通过使用数据结构（如***BST***）和排序算法（如*quicksort()）来*获得出色的结果。***然而。***对于安全关键型应用，或实时应用，其中的例子包括医疗设备的控制软件、发电、运输（飞机飞行控制）、机器人、制造业等等；***你不能使用一个最坏情况下性能很差的数据结构***。关键是，即使你需要非常不走运地碰到触发最坏情况或接近最坏情况的性能的输入，你也不能承担这种风险。这类应用需要从其软件中涉及的所有数据结构和算法中***保证性能界限***。因此，在你进入B63和C85时，请记住这一点，在那里将详细讨论这些问题。

额外的练习

Ex0 - 画出将下列键依次插入最初为空的BST的结果。键按字母顺序排列。

"钢铁侠3》、《黑豹》、《复仇者联盟》、《美国队长》、《钢铁侠2》、《水行侠》、《蝙蝠侠归来》、《雷神》、《蜘蛛侠：进入蜘蛛世界》。

例1--在我们删除 "水鬼 "后画出BST

例2--在我们删除 "钢铁侠3 "后画出BST

例3--通过对例2中的BST进行***后序遍历而***产生的电影列表是什么？

Ex4--
一个键值为电影标题的BST很可能最终会遇到这样的问题：有多个键值相同的条目。正如我们上面所讨论的，这确实不是一个好情况。请***给出至少3个不同的建议，说明我们如何建立一个BST，使条目按电影名称组织，但又不存在重复的键。***

Ex5 - 对于***大的N值***，以下哪项的复杂性***最低***？

a) 250000*Log(N)
b) .001*N
c) .000001*N^2
d) 5000*$Log(N)^2$

Ex6 - 对于***小的N值***，以下哪项的复杂性***最低***？

a) 5.25*N
b) 1.11*N
c) 5*Log(N)
d) 2.1*$Log(N)^2$

Ex7--回顾一下，在上面的讨论中，我们指定了对一个节点的键值的更新为

BST是不允许的，因为它可能破坏***BST***属性。然而，在实践中，我们可能会遇到这样的情况：我们*必须更新一个用作键的数据字段*。在下面的空白处列出我们可以采取的步骤来完成这个任务，同时不破坏树中的***BS***T属性（伪代码就可以了，不需要为此写C代码）。*提示：*考虑所有可用的***BST***操作。

Ex8 - 根据P.16的描述实现*quicksort()*。你也可以参考在线资源。