普通的二叉搜索树作为数据存储工具有着很多优点：（1）可以快速找到一个给定关键字的数据项。（2）可以快速的插入和删除数据项。其他的数据结构，如数组、有序数组、链表，执行这些操作比较慢。二叉搜索树的缺点：如果树中插入的是随机数据，则执行效果很好，但是如果插入的是有序的数据，或者逆序的数据，速度就会比较慢。因为当插入的数据时有序时，二叉树就是非平衡的。而对于非平衡二叉树，快速查找、删除和插入指定数据项的优点就没有了。

红黑树是解决非平衡二叉树的方法，极端情况下，当数没有分支时，非平衡二叉树就是一个链表。数据的排列是一维的而不是二维的，时间复杂度降低到 O(N)，而不是平衡树的 O(logN)。在一棵有 10000 个数据项的非平衡树中搜索数据项平均需要比较 5000 次，而在随机插入生成的平衡树中搜索数据项只需要 14 次比较。

为了能够以较快的时间 O(logN)来搜索一棵树，需要保证树总是平衡的，或者大部分是平衡的，即对树中的每个节点，在它左边的后代数目和在它右边的后代数目应该大致相等。红黑树的平衡是在插入的过程中进行的，对一个要插入的数据项，插入时要检查不会破坏树一定的特征。如果破坏了，程序会进行纠正，根据需要更改树的结构，通过维持树的特征，保持了树的平衡。

红黑树特征：（1）节点都有颜色（2）在插入和删除的过程中，要遵循保持这些颜色的不同排列的规则。

在红黑树中，每一个节点或者是黑色的或者是红色的，也可以是任意两种颜色，颜色只是用于标记。在节点类中增加一个数据字段，来表示颜色的信息。

当插入或者删除一个新节点时，必须要遵循一定的规则：（1），每一个节点不是黑色的就是红色的。（2）根总是黑色的。（3）如果节点是红色的，则它的子节点必须是黑色的，反之则不一定成立。（4）从根到叶节点的每条路径上，必须包含相同数目的黑色节点。

红黑树的查找、插入和删除的时间复杂度 O(logN)。在红黑树的查找和在普通二叉树中的查找时间应该几乎完全一样，因为在查找的过程中并没有应用到红黑树的特征。额外的开销只是每一个节点的存储空间增加了一个来存储红黑颜色的 Boolean 变量。

AVL 平衡树是最早的平衡树，在 AVL 树中每一个节点存储一个额外数据，左子树和右子树的高度差不会超过 1 层。插入之后检查新节点插入所在子树的根。如果它的子节点的高度相差大于 1，执行旋转使得高度差相等。然后算法向上移动，检查节点，必要时均衡高度。这个检测所有路径一直向上，直到根为止。AVL 树查找的时间复杂度是 O(logN)，因为树一定是平衡的。但是插入一个节点时，需要扫描两趟树，一次向下查找插入点，一次向上平衡树，AVL 没红黑树效率高，也没红黑树常用。

特点：（1）一次颜色变换把一个黑色节点和它的两个红色子节点改成一个红色节点和两个黑色子节点。（2）在一次旋转中，指定一个节点为顶端节点。（3）右旋把顶端节点移动到右子节点的位置，并把顶端节点的左子节点移动到顶端节点的位置。（4）左旋把顶端节点移

动到左子节点的位置，并把顶端节点的右子节点移动到顶端节点的位置。（5）当顺着树向下查找新节点的插入位置时，应用颜色变换，并且有时应用旋转。颜色变换通过简单的方法就使得树在插入后恢复成正确的红黑树。（6）新节点插入后再检测红红冲突，如果发现有违背红黑规则的现象，执行适当的旋转使得树恢复红黑特性。（7）调整之后使得树大致上是平衡的。（8）在二叉树中加入红黑平衡对平均执行效率只有很小的负面影响，但是能够避免对有序的数据操作的最坏的性能。

实现优先级队列的另一种结构，堆。堆是一种树，由堆实现的优先级队列的插入和删除时间复杂度都是 O(logN)。堆是完全二叉树，除了最后一层节点不是满的，其他的每一层从左到右都是满的。它常常用一个数组实现，堆中的每个节点都满足堆的条件，即每一个节点的关键字都大于或者等于这个节点的所有节点的关键字。移除堆中关键字最大的节点，这个节点总是根节点，

堆中元素的移除：移除堆中关键字最大的节点，这个节点总是根节点，根在堆数组中的索引总是 0。一旦移除了根节点，数组里会空一个数据单元，为了将数据单元填充，将数组中的所有数据项都向前移动一个单元。（1）移除根。（2）把最后一个节点移动到根的位置。（3）一直向下筛选这个节点，直到它在一个大于它的节点之下，小于它的节点之上为止。最后一个节点是树最底层的最右端的数据项，对应数组中最后一个填入的数据单元，把此节点直接复制到根节点为止，即 arr[0]=arr[n-1]。

堆中元素的插入：插入节点使用向上筛选方法，节点初始时插入到数组最后一个空着的单元中，数组容量大小加 1。如果新插入的节点大于它的父节点，则会破坏堆的条件。因为父节点在堆的底端，可能很小。需要向上筛选新节点，直到它在一个大于它的节点之下，在一个小于它的节点之上。

用数组表示一棵树时，如果数组中节点的索引为 x。则父节点的下标为(x-1)/2，左子节点的下标为 2*x+1，右子节点的下标为 2*x+2。

外部存储特指某类磁盘系统，目前为止的数据结构都是假设存储在主存中的，即 RAM，随机访问存储。但是很多情况下要处理的文件数量太大，不能都存储在主存中，则需要另外一种存储方法。磁盘文件存储器一般比主存大的多，使得存储每个字节的花费比较低。磁盘的另一个好处就是持久性，关掉计算机之后，在主存中的数据会丢失，磁盘文件存储器突然断电后还可以保存数据。外部存储的缺点是比主存慢，由于速度不同，因此需要不同的技术进行管理。B 树用在外部存储上的数据时，起到很大的作用。多叉树是指节点的子节点多于两个并且数据项多于一个。

对于外部数据需要使用和内部数据不一样的树，为多叉树，每个节点需要有更多的数据项，称之为 B-树。B 树通常更强调每个节点包含更多的子节点。

B-树的查找，在记录中按关键字查找和在内存的 2-3-4 树中查找很类似。首先，含有根的块读入到内存中，然后搜索算法开始在节点中查找，如果块不是满的，则检查所有的块。

从 0 开始，当记录的关键字比较大的时候，需要找在这条记录和前一条记录之间的那个子节点。持续这个过程直到找到正确的节点。如果到达叶节点还没有找到那条记录，则查找不成功。

B-树的插入过程像 2-3 树，而不是 2-3-4 树。2-3-4 树有很多节点不满，实际上有的只有一个数据项。尤其是每次节点分裂时总会产生两个每个有一个数据项的节点。尽可能让 B-树满很重要，这样每次存取磁盘时，读取整个节点，可以获得最大数量的数据。为了达到这个目的，B-树的插入过程和 2-3-4 树的插入过程有不同如下：（1）节点分裂时数据项平分，一半到新创建的节点中去，一半保留在原来的节点中。（2）节点分裂像 2-3 树从底向上，而不是自顶向下。（3）原节点内，中间数据项不上移，而是加上数据项后所组成的节点数据项序列的中间数据项上移。

B 树的分裂规则导致了 B 树上的所有节点要求至少是半满的。每个节点有很多记录，每层有很多节点，在 B 树上的操作比较快。节点中记录越多，树的层数越少，树的层数越少，访问效率越高。

多叉树总结：（1）多叉树比二叉树有更多的关键字和子节点。（2）2-3-4 树是多叉树，每个节点最多有三个关键字和四个子节点。（3）多叉树中，节点中数据项按照关键字升序排列。（4）2-3-4 树中，所有插入都在叶节点上，所有的叶节点在同一层。（5）在 2-3-4 树中有三种可能的节点。2-节点有 1 个关键字和 2 个子节点，3-节点有 2 个关键字和 3 个子节点，4-节点有 3 个关键字和 4 个子节点。（6）2-3-4 树中没有 1-节点。（7）在 2-3-4 树中查找，检查每个节点的关键字。没有找到时，如果要查找节点的关键字比 key0 小，下一个节点就是 child0；如果要查找节点的关键字在 key0 和 key1 之间，下一个节点就是 child1；如果要查找节点的关键字在 key1 和 key2 之间，下一个节点就是 child2；如果要查找节点的关键字大于 key2，下一个节点就是 child3。（8）在 2-3-4 树中插入需要在查找插入点的过程中，顺着树的路径向下分裂路径上每个满的节点。(9)分裂根要创建两个新节点，分裂出另一个节点，创建一个新节点。（10）只有分裂根是 2-3-4 树的高度才会增长。（11）2-3-4 树和红黑树之间存在一对一的对应关系。（12）把 2-3-4 树转化为红黑树，需要把每个 2-节点变成黑色节点，把每个 3-节点变成一个黑色的父节点和一个红色的子节点，把每个 4-节点变成一个黑色父节点和两个红色子节点。（13）当 3-节点转化成一个父节点和子节点时，每个节点都可以做父节点。（14）2-3-4 树中分裂节点和在红黑树中进行颜色变换一样。（15）红黑树中的旋转对应于转化 3-节点时在两种可能的倾斜方向间变化。（16）2-3-4 树的高度为 $\log_2 N$。（17）查找时间和高度成正比。（18）2-3-4 树浪费空间，很多节点不满一半。（18）2-3 树类似于 2-3-4 树，除了只能有 1 个或 2 个数据项以及 1 个或者 2 个或者 3 个子节点。（19）在 2-3 树中插入，需要找到合适的叶节点，然后从叶节点开始向上分裂，直到找到不满的节点。（20）外部存储的意思是在主存外面保存数据，通常是在磁盘上。（21）外部存储比主存大，但是每个字节比较便宜，速度比较慢。（22）外部存储中的数据通常需要在主存间来回传送，一

次传送一块。（23）在外部存储器中的数据可以按照关键字顺序有序排列，查找比较快，插入和删除比较慢。（24）B-树是多叉树，每个节点可以有几十个或者上百个关键字和子节点。（25）B-树中子节点的个数总是比关键字的个数多 1。（26）为了达到最好的性能，B-树通常在一个节点中保存一块的数据。

B+树由 B 树和索引顺序访问方法（ISAM）演化，B+树是为磁盘或者其他直接存取辅助设备设计的一种平衡查找树。在 B+树中，所有记录节点都是按照键值的大小顺序存放在同一层的叶子节点上，由各叶子节点指针进行连接。如果从最左边的叶子节点进行顺序遍历可以得到所有键值的顺序排序。

B+树索引的本质就是 B+树在数据库中的实现，B+树在数据库中有一个特点是高扇出性。因此在数据库中，B+树的高度一般在 2-4 层，查找某一个键值的行记录时最多只需要 2-4 次 IO。一般的机械磁盘每秒至少可以做 100 次 IO，2-4 次意味着查询时间只需要 0.02-0.04 秒。

数据库中的 B+树索引可以分为聚集索引和辅助索引，其内部都是 B+树，即高度平衡的，叶子节点存放着所有的数据。聚集索引和辅助索引的不同时，叶子节点存放的是否是一整行的信息。

页是 InnoDB 存储引擎管理数据库的最小磁盘单位，页大小为 16KB，且不可更改。MySQL 的 varchar 数据类型可以存放 65535 个字节，实际只能存储 65532 个。InnoDB 是 B+树结构，因此每个页中至少应该有两个行记录，否则变成了链表。一行记录的最大长度的阈值是 8098。在 InnoDB 存储引擎中，每个数据页中有两个虚拟的行记录，用来限定记录的边界。

查询 B+树索引的流程：首先通过 B+树索引找到叶节点，再找到对应的数据页，然后将数据页加载到内存中，通过二分查找 PageDirectory 中的 slot，查找出一个粗略的目录，然后根据槽的指针，指向链表中的行记录，之后在链表中依次查找。B+树索引不能找到具体的一条记录，只能找到对应的页。把页从磁盘装入到内存中，再通过 PageDirectory 进行二分，同时二分也可能找不到具体的行记录，只能找到一个接近的链表中的点，再从这个点开始遍历链表进行查找。

聚集索引是按照每张表的主键构造的一棵 B+树，并且叶子节点中存放着整张表的行记录数据，让聚集索引的节点成为数据页，这个特性决定了索引组织表中的数据也是索引的一部分。由于实际的数据页只能按照一棵 B+树进行排序，所以每张表只能拥有一个聚集索引。查询优化器倾向于采用聚集索引，其直接存储行数据，所以主键的排序查询和范围查找速度非常快。不是物理上的连续，而是逻辑上的，一开始数据顺序插入，是物理上连续，随着数据增删，物理上不再连续。

辅助索引的页级别不包含行的全部数据。叶子节点除了包含键值以外，每个页级别中的索引行中还包含一个书签，用来告诉 InnoDB 在哪找到与索引相对应的行记录，其中存储的

是聚集索引的键。辅助索引的存在并不影响数据在聚集索引的结构。InnoDB 会遍历辅助索引并通过页级别的指针获得指向聚集索引的键。然后通过聚集索引找到一个完整的行记录。如果只是需要辅助索引的值和聚集索引的值，只需要查找辅助索引即可以查询到要检索的数据，不需要再次去查聚集索引。

索引在创建或者删除的时候，MySQL 会先创建一个新的临时表，然后把数据导入到临时表，删除原表，再把临时表更改名称为原表名称。

MySQL 索引概述：所有 MySQL 列可以被索引，对相关列使用索引是提高 select 操作性能的最佳途径。根据存储引擎定义每个表的最大索引数和最大索引长度。所有存储引擎支持每个表至少 16 个索引，总索引长度至少为 256 字节。

FULLTEXT 索引可以用于全文搜索，只有 MyISAM 支持全文索引，并且只为 char、varchar 和 text 列。索引总是对整个列进行，不支持局部索引。也可以为空间列类型创建索引。只有 MyISAM 存储引擎支持空间类型，空间索引使用 R-树。默认 MEMORY 存储引擎使用 Hash 索引，也支持 B-树索引。

查询优化器不能使用 Hash 索引来加快 order by 操作，Hash 索引只用于使用=或者<=>操作符的等式比较。

主键就是唯一索引的一种，主键要求建表的时候指定，一般用 auto_increment 列，关键字是 primary key，索引列的值必须唯一，但是允许有空值，如果是组合索引，则列值的组合必须唯一。

全文索引是目前搜索引擎的关键技术，能够利用分词技术等多种算法智能分析出文本文字中关键词的频率及重要性，然后按照一定算法规则进行筛选。搜索选项必须在全文索引的索引列上，match 中指定的列必须在 fulltext 中指定过，MySQL5.6 之前只能应用在表引擎为 MyISAM 类型的表中，仅能在 char，varchar 和 text 类型的列上创建全文索引。可以在定义表的时候指定索引，也可以在创建表之后添加或者修改。搜索字符串必须是一个常量字符串，不能是表的列名。对于特别大数据的插入，向没有索引的表中插入数据后创建索引比向有索引的表中插入的过程要快。

使用了索引之后会按照索引的列进行排序，当不使用索引时，会直接按照默认的进行排序。可以使用 alter 添加数据的索引。（1）ALTER TABLE table_name ADD PRIMARY KEY(column_list)，添加一个主键，主键的索引值必须是唯一的，并且不能是 null。（2）ALTER TABLE table_name ADD UNIQUE index_name (column_list)，创建索引的值必须是唯一的，除了 null 之外，null 值可以出现多次。（3）ALTER TABLE table_name ADD INDEX index_name (column_list)，添加普通索引，索引值可以出现多次。（4）ALTER TABLE table_name ADD FULLTEXT index_name (column_list)，指定了索引为 FULLTEXT，用于全文索引。

索引提高了查询速度，但是却降低了更新速度，更新表的时候，MySQL 不仅要保存数据，还会保存一下索引文件。建立索引会占用磁盘空间的索引文件，如果在大表上创建了多

种组合索引，索引文件会比较大。

索引注意事项：（1）索引不会包含有 NULL 值的列，只要列中包含有 NULL 值，将不会被包含在索引中，复合索引中只要有一列含有 NULL 值，这一列对于复合索引就是无效的。在数据库设计时，不要让字段的默认值为 NULL。（2）对串列进行索引，如果可能，应该指定一个前缀长度。短索引不仅可以提高查询速度，还可以节省磁盘空间和 I/O 操作。如：如果有一个 CHAR(255)的列，如果在前 10 个或者 20 个字符内，多数值是唯一的，则不要对整个列进行索引。（3）索引列排序，MySQL 查询只使用一个索引，如果 where 子句中已经使用了索引，order by 中的列是不会使用索引的。因此数据库默认排序可以符合要求的话，不要使用排序操作，尽量不要包含多个列的排序，如果需要，最好给这些列创建复合索引。（4）一般情况下不要使用 like 操作，like “%A%”不会使用索引，而 like “A%”可以使用索引。（5）不要在列上进行计算，会导致索引失效而进行全表扫描。（6）不要使用 NOT IN 和 <>操作。

正向索引，搜索引擎将正向索引重新构建为倒排索引，把文件 ID 对应到关键词的映射转换为关键词到文件 ID 的映射，每个关键字都对应着一系列的文件。

倒排索引是实现“单词-文档矩阵”的一种具体存储形式，通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。主要组成：（1）单词词典（2）倒排文件

单词词典：搜索引擎的通常索引单位是单词，单词词典是由文档中出现过的所有单词构成的字符串集合，单词词典内每条索引项记载单词本身的一些信息以及指向“倒排列表”的指针。

倒排列表：记载了出现过的某个单词的所有文档列表及单词在该文档中出现的位置信息，每条记录称为一个倒排项。根据倒排列表，可知道哪些文档包含了这个单词。

倒排文件：所有单词的倒排列表顺序地存储在磁盘的某个文件里，这个文件称之为倒排文件，倒排文件是存储倒排索引的物理文件。

单词编号：与文档编号类似，搜索引擎内部以唯一的编号来表征某个单词，单词编号可以作为某个单词的唯一表征。

文档：一般搜索引擎的处理对象是互联网网页，文档的概念可以表现邮件、PDF 等多种概念，使用文档来表征文本信息。

文档集合：由若干文档组成的集合称之为文档集合。比如，海量的互联网网页或者大量电子邮件等。

文档编号：在搜索引擎内部，会将文档集合内每个文档赋予一个唯一的内部编号，作为该文档的唯一标识。

哈夫曼树：给定 N 个权值作为 N 个叶子节点，构造一棵二叉树，若树的带全路径长度达到最小，则这个树被称之为哈夫曼树。

哈夫曼树：（1）路径，在一棵树中，从一个结点往下可以达到的孩子或者孙子结点之间

的通路，称为路径。通路中分支的数目称为路径长度。若规定根结点的层数为 1，则从根结点到第 L 层结点的路径长度为 L-1。（2）若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。(3)树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为 WPL。

假设有 N 个权值，则构造出的哈夫曼树有 N 个叶子结点。N 个权值分别为 W1，W2，……，Wn，哈夫曼树的构造规则为：（1）将 W1，W2，……Wn 看成是有 N 棵树的森林（每棵树只有一个结点）（2）在森林中选出根结点的权值最小的两棵树进行合并，作为一棵新树的左子树、右子树，且新树的根结点权值为左、右子树根结点权值之和。（3）从森林中删除选取的两棵树，并将新树加入森林。（4）重复前两个步骤，直到森林中只有一棵树为止。该树即为所生成的哈夫曼树。

哈夫曼编码应用：哈夫曼进行通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。在发送端通过一个编码系统对传输数据预先编码，在接收端将传来的数据进行译码。对于双工信道，每端都需要一个完整的编码、译码系统。在数字通信中，经常需要将传送的文字转换成由二进制字符 0 和 1 组成的二进制串，称为编码。在传送电文时，希望电文代码尽可能短，采用哈夫曼编码构造的电文的总长度最短。

电文中每个字符出现的概率是不同的，如果采用不等长编码，让出现频率低的字符具有较长的编码，则可能缩短传送电文的总长度。如果对某一字符集进行不等长编码，则要求字符集中的任一字符的编码都不能是其他字符编码的前缀，符合此要求的编码叫做前缀编码。为了使得不等长编码也是前缀编码，可以用字符集中的每个字符作为叶子结点生成一颗编码二叉树，为了获得最短的电文长度，可以将每个字符出现的频率作为字符的权值赋予对应的叶子节点，求出此树的最小带权路径长度就是电文的最短编码。根据哈夫曼算法构造哈夫曼树，以字符集的中的字符为叶子结点，频率作为权值，构造一棵哈夫曼树。其中，每个结点分别对应一个字符，对树中的边做标记，把左分支记为“0”，右分支记为“1”。定义字符的编码是从根节点到该字符所对应的叶子结点的路径上，各条边上的标记所组成的序列就是哈夫曼编码。对于任意的字符集总是能够构造出这样的编码二叉树。由于在任何一条从根结点到一个叶子结点的路径上一定不会出现其他叶子结点。这种方法得到的编码一定是前缀编码，通过遍历二叉树，可以求出每个字符的编码。