|  |  |
| --- | --- |
| 实验名称：Linux大作业 | 班级：硬件3班 |
| 学号：161403324 | 姓名：彭依玲 |

**一、实验目的**

（1）理解堆、栈、B+树、红黑树这四种数据结构的基本原理。

（2）用C语言实现这四种数据结构，并且每种数据结构至少完成一种其对应的功能。

**二、实验内容**

**1. 堆（Heap）**

代码已附在./code/Heap文件夹中，还有在linux下生成的可执行文件heap.exe。

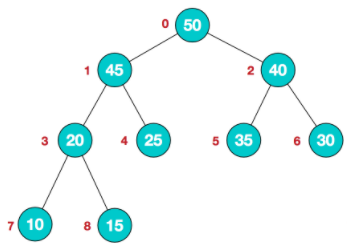
以下是我写在博客里的内容：

（博客地址：<https://blog.csdn.net/weixin_44861366/article/details/90710361）>

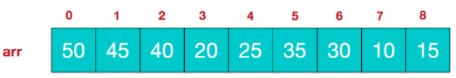
（1）堆的性质

堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆。

在数据结构中，我们将堆的逻辑结构映射到数组中存储，如下图：



在数组中存储的样子如下：



于是在数组中，堆中节点的索引有如下定义：  
****大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]  
小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]****

**这个定义在进行堆排序时需要经常用到。**

**（2）堆排序的性质**

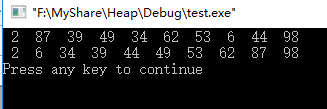
**堆排序的主要思想如下：**

**将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余n-1个元素重新构造成一个堆，这样会得到n个元素的次小值。如此反复执行，便能得到一个有序序列了**

（3）堆排序的实现步骤

1. 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。在我的代码中，用PercDown(int A[], int i, int N)构成堆，从数组的第一个数开始，设其下标为i，那么其左孩子节点的下标则为2i+1，右孩子节点下标为2i+2，找到两个孩子中最大的，若这个最大的孩子值比该节点大，则将最大孩子的值与该节点值交换。这样循环下去，可以得到堆。
2. 将堆顶元素与末尾元素进行交换，使末尾元素最大。交换后，末尾元素，即值最大的这个元素，逻辑上脱离该数组，就是以后的调整结构中我们不再考虑它。
3. 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序（不包括）。在我的代码中用HeapSort(int A[], int N)函数来实现这个功能。

（4）运行结果



**2. 栈（Stack）**

代码已附在./code/Stack文件夹中，还有在linux下生成的可执行文件stack.exe。

以下是我记录在博客的内容：

<https://blog.csdn.net/weixin_44861366/article/details/90169634>

（1）栈的性质

栈也是一种特殊的线性表，但不同的是，栈的操作与传统的线性表不同。传统的线性表可以完成随机位置存取，而栈的结构决定了它进行操作的特点：仅仅在表尾进行插入或删除操作（后进先出）。表尾端称作栈顶，而与之相对的，表头端称作栈底。可以将栈的结构与子弹弹夹进行类比，后压入的子弹先被发射，正如栈的后进先出特点一般。

栈的顺序表示基本结构如下，要求利用一组地址连续的内存单元来存储信息，元素的数据按顺序由栈底依次存储到栈顶，再由栈顶开始依次向下取出或添加元素。

在栈的顺序表示之中，先定义栈的类名为SqStack,需要设置名为base与top的两个指针，分别指示栈底的内存与栈顶的内存，同时定义int类型的变量用于设置栈的内存。

（2）用栈实现算术解析表达式

栈在解决算术解析表达式时常用，使用方法是：

A. 将算术解析表达式（中缀表达式）转换成后缀表达式

B. 使用栈进行求值。

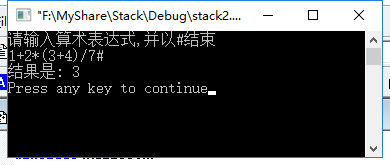
（3）代码分析

整个代码中，值得一提的是比较运算符的优先级的函数Precede(SElemType a,SElemType b)和运算函数Operate(SElemType a,SElemType theta,SElemType b)，以及将这两个函数综合起来的EvaluateExpression()函数。

我设定了两个栈，一个符号栈OPTR，一个数字栈OPND。

比较优先级函数中有两个参数，被比较者a和比较者b，这个顺序要求很严格，因为被比较者a一般是已在栈中且在OPTR栈顶的元素，而b要与其进行优先级比较才能决定是入栈还是运算，如果a<b，则入栈；若a>b，则a出栈，且在OPND栈中取出两个数，进行运算；若a=b，则只将OPTR的栈顶的符号出栈。

（4）运行结果



**3. B+树**

代码已附在./code/Stack文件夹中，还有在linux下生成的可执行文件stack.exe。

以下是我记录在博客中的内容：

<https://blog.csdn.net/weixin_44861366/article/details/90716576>

（1）B+树的性质

要想理解B+树，必须先知道什么是二叉排序树。

**二叉排序树：**或者是一课空树；或者是具有下列性质的二叉树：

a. 若它的左子树不空，则左子树上所有结点的值均小于它的根节点的值；

b. 若它的右子树不空，则右子树上所有结点的值均大于它的根节点的值；

c. 它的左、右子树也分别为二叉排序树。

**平衡二叉树（又称AVL树）：**它或者是一课空树，或者是具有下列性质的二叉树：

a. 它的左子树和右子树都是平衡二叉树；

b. 左子树和右子树的深度之差的绝对值不超过1。

**B树（也即B-树，也称平衡多叉查找树）：**它或者是一课空树，或者是具有下列性质的二叉树：

a. 树中每个结点至多有m棵子树；

b. 若根节点不是叶子节点，则至少有两棵子树；

c. 除根外的所有非终端结点至少有[m/2]棵子树；

d. 所有的非终端结点中包含下列信息数据

(n，A0，K1，A1，K2，A2，…，Kn，An)

PS：每个节点至多有m-1个关键字，至多有m个儿子节点

**B+树：**一棵 m阶的B\*树和m阶的B-树的差异在于:

a. 有n棵子树的结点中含有n个关键字。

b. 所有的叶子结点中包含了全部关键字的信息,及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

c. 所有的非终端结点可以看成是索引部分,结点中仅含有其子树(根结点)中的最大(或最小)关键字。

值得一提的是插入算法：

a. 首先执行查找算法，找出被插结点的父亲结点。

b. 判断被插结点是其父亲结点的左、右儿子。将被插结点作为叶子结点插入。

c. 若二叉树为空。则首先单独生成根结点。

注意：新插入的结点总是叶子结点。

如果关键字满了，兄弟节点没满，就把最靠近兄弟节点的值过给它；如果兄弟节点满了，就分裂。如果分裂导致父节点的子节点数超过了M，也满了，那么父节点就也得分裂，分裂得到的两个父节点各得一般的儿子数。

（2）代码分析

整个代码中最重要的是InsertElement这个函数，其包含两种插入功能：，一个是插入关键字，另外就是插入节点。

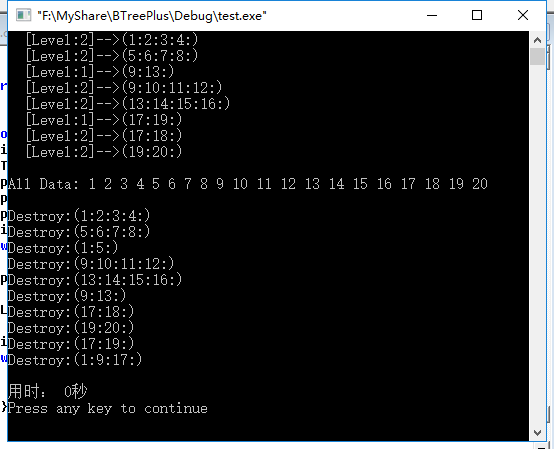
还有一个MoveElement函数，功能是当一个节点关键字满了，而其兄弟节点没满时，将自己的关键字移动到兄弟节点中，这种移动也分两种，一是树叶节点的移动，另一个则是中间节点的移动。

再就是分裂函数SplitNode，功能像上面描述的，当一个节点关键字满了，其兄弟节点也都满了，所以它得分裂。分裂就分两种，一是树叶节点的分裂，这种分裂影响比较小，只需要生成一个新节点，将自己的一半关键字分裂给该新节点，然后将新节点接到父节点下就行。另一种是中间节点的分裂，这种分裂生成的新节点需要过继原节点一半的孩子。

与以上提到的函数功能相对的有删除函数RemoveElement、合并函数MergeElement，这些便不赘述了。

另外深度遍历函数的实现相对于B+树本身功能的实现来说很简单，仅仅只是指针的递归输出，没什么好说的。

（3）运行结果



**4. 红黑树**

（1）红黑树的性质

红黑树就是为了解决二叉查找树不平衡发明的，所以它是一种平衡的树。除了符合二叉查找树的特性之外，还有下列特性：

a. 节点是红色或者黑色

b. 根节点是黑色

c. 每个叶子的节点都是黑色的空节点（NULL）

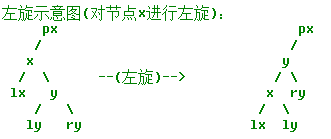
d. 每个红色节点的两个子节点都是黑色的。

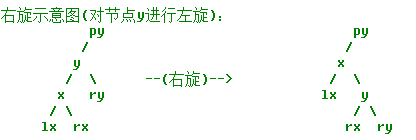
e. 从任意节点到其每个叶子的所有路径都包含相同的黑色节点。

条条框框很多，所以插入删除都很容易影响红黑树的平衡。于是我们有变色和旋转两种恢复平衡的方法。

变色很容易，能使红黑树节点平衡就行，但是有时候变色无法使红黑树完全平衡，反而引起一系列的不平衡，这时候，就需要旋转了。

旋转有两种，左旋转和右旋转。左旋转就是逆时针旋转两个节点，使父节点被自己的右孩子取代，而自己成为自己的左孩子；右旋转就是顺时针旋转两个节点，使得自己的父节点被左孩子取代，而自己成为自己的右孩子。





（2）代码分析

其实红黑树代码看起来长，对于它的理解相较于B+树来说比较简单。

因为红黑树就是二叉查找树，所以对它的前后中序遍历、（非）递归查找、最大最小值节点查找、后继节点和前驱节点的查找等与二叉树一般无二，而且很简单，在这里就不细说了，而重难点也在红黑树平衡的实现上。

红黑树平衡的实现在本代码中主要用插入和删除函数。

插入平衡相关函数有新建结点函数insert\_rbtree、结点插入函数rbtree\_insert、插入修正函数rbtree\_insert\_fixup。

删除函数： 结点删除函数 rbtree\_delete、删除修正函数rbtree\_delete\_

Fixup。

其中比较难理解，在这里我要提几句的是rbtree\_insert\_fixup函数。它实现的功能是：

当插入的节点的父节点存在且为红色

A。父节点是祖父节点的左孩子：

▶如果叔叔节点是红色，就将叔叔节点和父节点改成黑色，祖父节点改成红色；

▶如果叔叔节点是黑色，且当前节点是右孩子，就进行左旋转，并将父节点和该节点交换；

▶如果叔叔节点是黑色，且当前节点是左孩子，就将父节点改成黑色，祖父节点改成红色，对根节点和祖父节点进行右旋转。

B。父节点是祖父节点的右孩子

▶如果叔叔节点是红色，同上。

▶如果叔叔节点是黑色，且当前节点是左孩子，就进行右旋转，并将父节点和该节点交换；

▶如果叔叔节点是黑色，且当前节点是右孩子，就将父节点改成黑色，祖父节点改成红色，对根节点和祖父节点进行左旋转。

（3）运行结果

