

Linux 实验报告

数据结构

姓名：赵天凤 学号：161403122 班级：硬件一班

【实验题目】

- 1、编程实现堆排序，并了解堆排序的方法。
- 2、编程实现 B+树。
- 3、编程实现红黑树。
- 4、编程实现利用栈求出算术表达式的值。

【实验要求】

实验代码能够运行，并在 Linux 下编译和运行，学会使用 gcc 编译程序，并进行运行。

【实验内容】

1、堆排序

(1) 堆排序是利用堆的性质进行的一种选择排序。堆实际上是一棵完全二叉树，其任何一非叶节点满足性质

$Key[i] \leq key[2i+1] \&\& Key[i] \leq key[2i+2]$ 或者
 $Key[i] \geq Key[2i+1] \&\& key \geq key[2i+2]$

即任何一非叶节点的关键字不大于或者不小于其左右孩子节点的关键字。

(2) 堆分为大顶堆和小顶堆，满足

$Key[i] \geq Key[2i+1] \&\& key \geq key[2i+2]$ 称为大顶堆，满足

$Key[i] \leq key[2i+1] \&\& Key[i] \leq key[2i+2]$ 称为小顶堆。

由上述性质可知大顶堆的堆顶的关键字肯定是所有关键字中最大的，小顶堆的堆顶的关键字是所有关键字中最小的。利用大顶堆(小顶堆)堆顶记录的是最大关键字(最小关键字)这一特性，使得每次从无序中选择最大记录(最小记录)变得简单。

(3) 堆排序其实也是一种选择排序，是一种树形选择排序。只不过直接选择排序中，为了从 $R[1...n]$ 中选择最大记录，需比较 $n-1$

次，然后从 $R[1...n-2]$ 中选择最大记录需比较 $n-2$ 次。事实上这 $n-2$ 次比较中有很多已经在前面的 $n-1$ 次比较中已经做过，而树形选择排序恰好利用树形的特点保存了部分前面的比较结果，因此可以减少比较次数。对于 n 个关键字序列，最坏情况下每个节点需比较 $\log_2(n)$ 次，因此其最坏情况下时间复杂度为 $n \log n$ 。堆排序为不稳定排序，不适合记录较少的排序。

2、B+树

B+ 树是一种树数据结构，通常用于数据库和操作系统的文件系统中。**B+** 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。**B+** 树元素自底向上插入，这与二叉树恰好相反。

在 **B+** 树中的节点通常被表示为一组有序的元素和子指针。如果此 **B+** 树的序数（order）是 m ，则除了根之外的每个节点都包含最少一个元素最多 $m-1$ 个元素，对于任意的节点有最多 m 个子指针。对于所有内部节点，子指针的数目总是比元素的数目多一个。因为所有叶子都在相同的高度上，节点通常不包含确定它们是叶子还是内部节点的方式。

每个内部节点的元素充当分开它的子树的分离值。例如，如果内部节点有三个子节点（或子树）则它必须有两个分离值或元素 a_1 和 a_2 。在最左子树中所有的值都小于等于 a_1 ，在中间子树中所有的值都在 a_1 和 a_2 之间($(a_1, a_2]$)，而在最右子树中所有的值都大于 a_2 。

三个功能：

查找

查找以典型的方式进行，类似于二叉查找树。起始于根节点，自顶向下遍历树，选择其分离值在要查找值的任意一边的子指针。在节点内部典型的使用是二分查找来确定这个位置。

插入

节点要处于违规状态，它必须包含在可接受范围之外数目的元素。首先，查找要插入其中的节点的位置。接着把值插入这个节点中。如果没有节点处于违规状态则处理结束。

如果某个节点有过多元素，则把它分裂为两个节点，每个都有最小数目的元素。在树上递归向上继续这个处理直到到达根节点，如果根节点被分裂，则创建一个新根节点。为了使它工作，元素的最小和最大数目典型的必须选择为使最小数不小于最大数的一半。

删除

首先，查找要删除的值。接着从包含它的节点中删除这个值。

如果没有节点处于违规状态则处理结束。

如果节点处于违规状态则有两种可能情况：

它的兄弟节点，就是同一个父节点的子节点，可以把一个或多个它的子节点转移到当前节点，而把它返回为合法状态。如果是这样，在更改父节点和两个兄弟节点的分离值之后处理结束。

它的兄弟节点由于处在低边界上而没有额外的子节点。在这种情况下把两个兄弟节点合并到一个单一的节点中，而且我们递归到父节点上，因为它被删除了一个子节点。持续这个处理直到当前节点是合法状态或者到达根节点，在其上根节点的子节点被合并而且合并后的节点成为新的根节点。

3、红黑树

首先红黑树是一棵二叉搜索树，它在每个结点上增加了一个存储位来表示结点的颜色，可以是 RED 或者 BLACK。通过对一条从根节点到 NIL 叶节点（指空结点或者下面说的哨兵）的简单路径上各个结点在颜色进行约束，红黑树确保没有一条路径会比其他路径长出 2 倍，因而是近似平衡的。

用途

红黑树和 AVL 树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。对于查找、插入、删除、最大、最小等动态操作的时间复杂度为 $O(\lg n)$ 。常见的用途有以下几种：

STL（标准模板库）中在 set map 是基于红黑树实现的。

Java 中在 TreeMap 使用的也是红黑树。

epoll 在内核中的实现，用红黑树管理事件块。

linux 进程调度 Completely Fair Scheduler, 用红黑树管理进程控制块。

4、栈

分为三个步骤：

1. 检查输入是否有误（因为输入其他的非预期字符，程序就会崩溃，我就试着加了一个检查输入的函数）
2. 先将正常的中缀表达式转换为后缀表达式
3. 再进行求值

根据后缀表达式求值比较简单，因为后缀表达式已经有了优先级。

比较难懂的是将中缀表达式转换为后缀表达式，需要考虑很多情况：

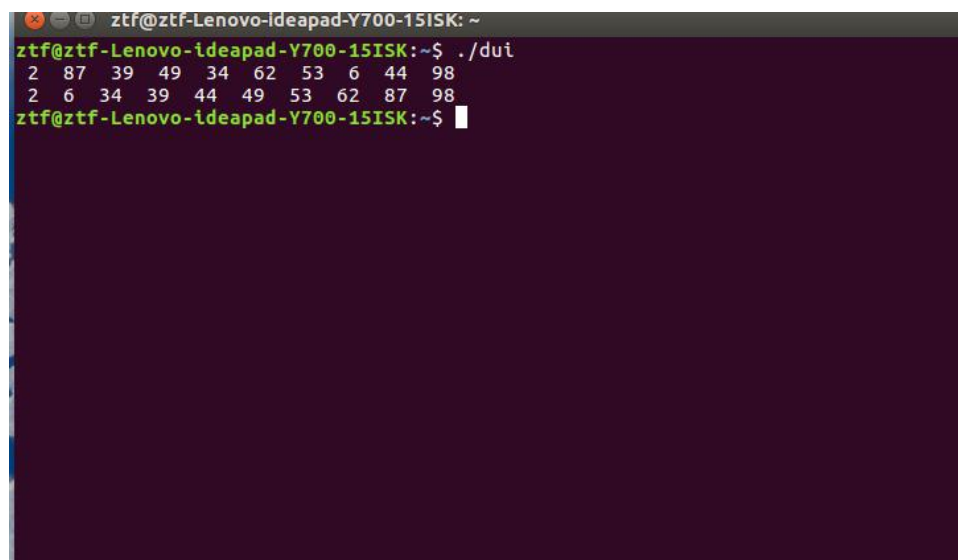
1. 如果字符是 '('，就直接入操作符栈，因为越内层的括号优先级越高，所以不用考虑什么。
2. 如果字符是 ')'，就要结束一个左括号了，将前面的操作符出栈送入 postexp，直到遇到 '('。

3. 如果字符是 '-' 或 '+'，就要将遇到的第一个 '(' 之前的所有操作符出栈送入 postexp。因为相对来说，后面的 '+' 和 '-' 优先级是最低的，低于前面的所有操作符。最后出栈完，再将它压入栈。

4. 如果字符是 '*' 或 '/'，先判断前面操作符栈底是否为 '*' 或 '/'，是的话就要将栈里的符号出栈送入 postexp。因为 '*' 或 '/' 的优先级低于前面的 '*' 或 '/'，是高于前面的 '+' 和 '-' 的。

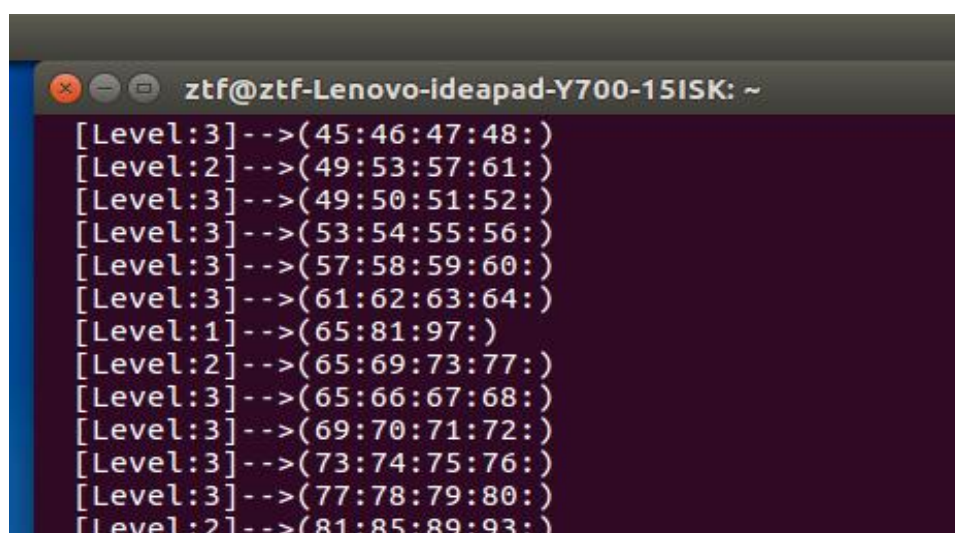
【运行结果】

1、堆排序，以 2,87,39,49,34,62,53,6,44,98 为例。

A terminal window with a dark purple background. The title bar reads 'ztf@ztf-Lenovo-Ideapad-Y700-15ISK: ~'. The prompt is 'ztf@ztf-Lenovo-ideapad-Y700-15ISK:~\$'. The user enters './dui'. The output shows two lines of numbers: '2 87 39 49 34 62 53 6 44 98' and '2 6 34 39 44 49 53 62 87 98'. The prompt returns to 'ztf@ztf-Lenovo-ideapad-Y700-15ISK:~\$' with a cursor.

```
ztf@ztf-Lenovo-Ideapad-Y700-15ISK: ~
ztf@ztf-Lenovo-ideapad-Y700-15ISK:~$ ./dui
2 87 39 49 34 62 53 6 44 98
2 6 34 39 44 49 53 62 87 98
ztf@ztf-Lenovo-ideapad-Y700-15ISK:~$
```

2、B+树

A terminal window with a dark purple background. The title bar reads 'ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~'. The prompt is 'ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~'. The output shows a B+ tree structure with levels and pointers in parentheses.

```
ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~
[Level:3]-->(45:46:47:48:)
[Level:2]-->(49:53:57:61:)
[Level:3]-->(49:50:51:52:)
[Level:3]-->(53:54:55:56:)
[Level:3]-->(57:58:59:60:)
[Level:3]-->(61:62:63:64:)
[Level:1]-->(65:81:97:)
[Level:2]-->(65:69:73:77:)
[Level:3]-->(65:66:67:68:)
[Level:3]-->(69:70:71:72:)
[Level:3]-->(73:74:75:76:)
[Level:3]-->(77:78:79:80:)
[Level:2]-->(81:85:89:93:)
```



```

[Level:3]-->(81:82:83:84:)
[Level:3]-->(85:86:87:88:)
[Level:3]-->(89:90:91:92:)
[Level:3]-->(93:94:95:96:)
[Level:2]-->(97:99:)
[Level:3]-->(97:98:)
[Level:3]-->(99:100:)

Destroy:(1:2:3:4:)
Destroy:(5:6:7:8:)
Destroy:(9:10:11:12:)
Destroy:(13:14:15:16:)
Destroy:(1:5:9:13:)
Destroy:(17:18:19:20:)
Destroy:(21:22:23:24:)
Destroy:(25:26:27:28:)
Destroy:(29:30:31:32:)
Destroy:(17:21:25:29:)
Destroy:(33:34:35:36:)
Destroy:(37:38:39:40:)
Destroy:(41:42:43:44:)
Destroy:(45:46:47:48:)
Destroy:(33:37:41:45:)
Destroy:(49:50:51:52:)
Destroy:(53:54:55:56:)
Destroy:(57:58:59:60:)
Destroy:(61:62:63:64:)
Destroy:(49:53:57:61:)
Destroy:(1:17:33:49:)
Destroy:(65:66:67:68:)
Destroy:(69:70:71:72:)
Destroy:(73:74:75:76:)
Destroy:(77:78:79:80:)
Destroy:(65:69:73:77:)
Destroy:(81:82:83:84:)
Destroy:(85:86:87:88:)
Destroy:(89:90:91:92:)
Destroy:(93:94:95:96:)
Destroy:(81:85:89:93:)
Destroy:(97:98:)
Destroy:(99:100:)
Destroy:(97:99:)
Destroy:(65:81:97:)
Destroy:(1:65:)

用时： 4秒
ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~$

```

3、红黑树

```

ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~$ ./rbtree
== 原始数据: 10 40 30 60 90 70 20 50 80
== 添加节点: 10

```

```

== 树的详细信息:
10(B) is root

== 添加节点: 40
== 树的详细信息:
10(B) is root
40(R) is 10's right child

== 添加节点: 30
== 树的详细信息:
30(B) is root
10(R) is 30's left child
40(R) is 30's right child

== 添加节点: 60
== 树的详细信息:
30(B) is root
10(B) is 30's left child
40(B) is 30's right child
60(R) is 40's right child

== 添加节点: 90
== 树的详细信息:
30(B) is root
10(B) is 30's left child
60(B) is 30's right child
40(R) is 60's left child
90(R) is 60's right child

== 添加节点: 70
== 树的详细信息:
30(B) is root
10(B) is 30's left child
60(R) is 30's right child
40(B) is 60's left child
90(B) is 60's right child
70(R) is 90's left child

== 添加节点: 20
== 树的详细信息:
30(B) is root
10(B) is 30's left child
20(R) is 10's right child
60(R) is 30's right child
40(B) is 60's left child
90(B) is 60's right child
70(R) is 90's left child

== 添加节点: 50
== 树的详细信息:
30(B) is root
10(B) is 30's left child
20(R) is 10's right child
60(R) is 30's right child
40(B) is 60's left child
50(R) is 40's right child
90(B) is 60's right child

70(R) is 90's left child

== 添加节点: 80
== 树的详细信息:
30(B) is root
10(B) is 30's left child
20(R) is 10's right child
60(R) is 30's right child
40(B) is 60's left child
50(R) is 40's right child
80(B) is 60's right child
70(R) is 80's left child
90(R) is 80's right child

```

```

== 前序遍历: 30 10 20 60 40 50 80 70 90
== 中序遍历: 10 20 30 40 50 60 70 80 90
== 后序遍历: 20 10 50 40 70 90 80 60 30
== 最小值: 10
== 最大值: 90
== 树的详细信息:
30(B) is root
10(B) is 30's left child
20(R) is 10's right child
60(R) is 30's right child
40(B) is 60's left child
50(R) is 40's right child
80(B) is 60's right child
70(R) is 80's left child
90(R) is 80's right child

== 删除节点: 10
== 树的详细信息:
30(B) is root
20(B) is 30's left child
60(R) is 30's right child
40(B) is 60's left child
50(R) is 40's right child
80(B) is 60's right child
70(R) is 80's left child
90(R) is 80's right child

== 删除节点: 40
== 树的详细信息:
30(B) is root
20(B) is 30's left child
60(R) is 30's right child
50(B) is 60's left child
80(B) is 60's right child
70(R) is 80's left child
90(R) is 80's right child

== 删除节点: 30
== 树的详细信息:
50(B) is root
20(B) is 50's left child
80(R) is 50's right child
60(B) is 80's left child
70(R) is 60's right child

```

```

90(B) is 80's right child

== 删除节点: 60
== 树的详细信息:
50(B) is root
20(B) is 50's left child
80(R) is 50's right child
70(B) is 80's left child
90(B) is 80's right child

== 删除节点: 90
== 树的详细信息:
50(B) is root
20(B) is 50's left child
80(B) is 50's right child
70(R) is 80's left child

== 删除节点: 70
== 树的详细信息:
50(B) is root
20(B) is 50's left child
80(B) is 50's right child

== 删除节点: 20
== 树的详细信息:
50(B) is root
80(R) is 50's right child

== 删除节点: 50
== 树的详细信息:
80(B) is root

== 删除节点: 80
== 树的详细信息:

```


4、栈实现算术表达式求值

```
ztf@ztf-Lenovo-ideapad-Y700-15ISK: ~  
ztf@ztf-Lenovo-ideapad-Y700-15ISK:~$ gcc -o struct struct.c  
ztf@ztf-Lenovo-ideapad-Y700-15ISK:~$ ./struct  
请输入以#结尾的表达式  
2+4*(2-1)#  
压入OPND栈 50  
压入OPTR栈 +  
压入OPND栈 52  
压入OPTR栈 *  
压入OPTR栈 (  
压入OPND栈 50  
压入OPTR栈 -  
压入OPND栈 49  
压入前0 0  
删除后OPND的头49 50  
运算后 45  
压入前1 2  
删除后OPND的头49 52  
运算后 42  
压入前1 4  
删除后OPND的头52 50  
运算后 43  
  
运算结果= 6ztf@ztf-Lenovo-ideapad-Y700-15ISK:~$
```