

# Linux 实验报告

姓名	庞竹	班级	硬件二班
学号	161403223	日期	2019.6.3

## 一、实验目的

- 1、了解 Linux 编程环境；掌握在 Linux 环境下编程的常用工具，例如：shell, vim, make, gedit, git 及各种语言的集成开发环境。
- 2、了解 Linux 系统的内存、进程、线程、同步、通信的基本原理和其在实际程序实际中的应用。
- 3、掌握 Linux 环境下的应用程序设计、开发和项目管理。
- 4、通过 16 学时的实验，在 Linux 环境下设计并实现一个代码量不小于 2000 的项目。

## 二、实验要求

在 Linux 环境下使用 Java 语言结合数据结构相关知识完成以下项目的具体实现：

- 1、堆
- 2、栈
- 3、红黑二叉树
- 4、B+树

## 三、实验内容

### 1、堆

(1) 大顶堆与小顶堆的具体实现（以小顶堆为例，实验代码均可实现）  
(HeapSort.java)

\*实现堆排序需解决两个问题：

- (1) 如何将  $n$  个待排序的数建成堆；
- (2) 输出堆顶元素后，怎样调整剩余  $n-1$  个元素，使其成为一个新堆。

\*建堆方法（小顶堆）：

对初始序列建堆的过程，就是一个反复进行筛选的过程。

- (1)  $n$  个结点的完全二叉树，则最后一个结点是第  $n/2$  个结点的子树。
- (2) 筛选从第  $n/2$  个结点为根的子树开始（ $n/2$  是最后一个有子树的结点），

使该子树成为堆。

- (3) 之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

\*调整堆的方法（小顶堆）：

(1) 设有  $m$  个元素的堆，输出堆顶元素后，剩下  $m-1$  个元素。将堆底元素送入堆顶，堆被破坏，其原因仅是根结点不满足堆的性质。

(2) 将根结点与左、右子树中较小元素的进行比较。

(3) 若与左子树交换：如果左子树堆被破坏，则重复方法(2)。

(4) 若与右子树交换，如果右子树堆被破坏，则重复方法(2)。

(5) 继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。

小顶堆运行结果：

```
os@os-VirtualBox:~/桌面/堆$ java -cp ./bin HeapSort
```

```
初始堆开始
待调整结点为: array[3] = 97
将与子孩子 array[7] = 49 进行比较
子孩子均比其小, 调整结束
待调整结点为: array[2] = 65
将与子孩子 array[6] = 27 进行比较
子孩子均比其小, 调整结束
待调整结点为: array[1] = 38
将与子孩子 array[3] = 97 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[7] = 49 进行比较
子孩子比其大, 交换位置
没有子孩子了, 调整结束
待调整结点为: array[0] = 49
将与子孩子 array[1] = 97 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[4] = 76 进行比较
子孩子比其大, 交换位置
没有子孩子了, 调整结束
初始堆结束
待调整结点为: array[0] = 38
```

```
将与子孩子 array[1] = 76 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[3] = 49 进行比较
子孩子比其大, 交换位置
没有子孩子了, 调整结束
待调整结点为: array[0] = 27
将与子孩子 array[2] = 65 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[5] = 13 进行比较
子孩子均比其小, 调整结束
待调整结点为: array[0] = 13
将与子孩子 array[1] = 49 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[4] = 49 进行比较
子孩子比其大, 交换位置
没有子孩子了, 调整结束
待调整结点为: array[0] = 13
将与子孩子 array[1] = 49 进行比较
子孩子比其大, 交换位置
继续与新的子孩子进行比较
将与子孩子 array[3] = 38 进行比较
```

```

孩子比其大，交换位置
没有孩子了，调整结束
待调整结点为：array[0] = 13
将与孩子 array[1] = 38 进行比较
孩子比其大，交换位置
没有孩子了，调整结束
待调整结点为：array[0] = 27
将与孩子 array[1] = 13 进行比较
孩子均比其小，调整结束
待调整结点为：array[0] = 13
13 27 38 49 49 65 76 97 os@os-VirtualBox:~/桌面/堆$

```

(2) 堆插入及排序(HeapSort\_delete.java):

```

//堆排序
public int[] heapSort(int[] array){
    array = buildMaxHeap(array); //初始建堆，array[0]为第一趟值最大的元素
    for(int i=array.length-1;i>1;i--){
        int temp = array[0]; //将堆顶元素和堆底元素交换，即得到当前最大元素正确的排序位置
        array[0] = array[i];
        array[i] = temp;
        adjustDownToUp(array, 0,i); //整理，将剩余的元素整理成堆
    }
    return array;
}

//插入操作:向大根堆array中插入数据data
public int[] insertData(int[] array, int data){
    array[array.length-1] = data; //将新节点放在堆的末端
    int k = array.length-1; //需要调整的节点
    int parent = (k-1)/2; //双亲节点
    while(parent >=0 && data>array[parent]){
        array[k] = array[parent]; //双亲节点下调
        k = parent;
        if(parent != 0){
            parent = (parent-1)/2; //继续向上比较
        }else{ //根节点已调整完毕，跳出循环
            break;
        }
    }
    array[k] = data; //将插入的结点放到正确的位置
    return array;
}

```

运行结果:

```

os@os-VirtualBox:~/桌面/堆2$ java -cp ./bin HeapSort
构建大根堆: 122 87 78 45 17 65 53 9 32
删除堆顶元素: 87 45 78 32 17 65 53 9 -99999
插入元素63:87 63 78 45 17 65 53 9 32
大根堆排序: 9 17 32 45 53 63 65 78 87 os@os-VirtualBox:~/桌面/堆2$

```

## 2、栈 (EvaluateExpression.java)

输入运算式并以#结尾，得到运算结果。

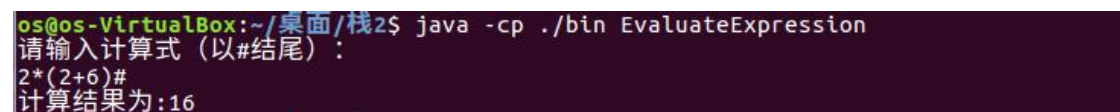
具体过程：

- \* 使用两个栈，分别是操作数栈（存储数字）和操作符栈（存储运算符）；
- \* 读入表达式时，如果是操作数，则入操作数栈；如果是运算符，则入操作符栈；
- \* 当运算符入栈时，和操作符栈栈顶元素比较优先级；
- \* 如果优先级比栈顶元素高，则入栈，并接收下一个字符；
- \* 如果和栈顶元素相同，则脱括号，并接收下一个字符（因为相同只有（）括号）；
- \* 如果小于栈顶元素优先级，则操作数出栈，操作符出栈，并计算运算结果再入栈。

\*\*循环的退出条件 直到运算全部结束，即当前栈顶元素和读入的操作符均为#。

```
while (c != '#' || stackOPR.peek() != '#') {
    if (isOPN(c)) { //判断是否是操作数
        //是 则入操作数栈，读入下一个
        stackOPN.push(c-48); //ascii码 '0'转为int是48 A是65 a是97
        c = (char) System.in.read();
    } else {
        switch (isPrior(c)) {
            case '>': //优先级比栈顶元素大，则入栈，并接收下一个字符
                stackOPR.push(c);
                c = (char) System.in.read();
                break;
            case '=': //优先级和栈顶元素相同
                stackOPR.pop(); //脱去括号
                c = (char) System.in.read();
                break;
            case '<': //优先级比栈顶元素小，则操作数出栈，操作符出栈，运算之后入栈。即意思是优先级高的先运算
                int b = stackOPN.pop(); //顺序靠后的操作数
                int a = stackOPN.pop(); //顺序靠前的操作数
                stackOPN.push(Option(a, b, stackOPR.pop()));
                break;
        }
    }
}
```

运行结果：



```
os@os-VirtualBox:~/桌面/栈2$ java -cp ./bin EvaluateExpression
请输入计算式（以#结尾）：
2*(2+6)#
计算结果为:16
```

## 3、红黑二叉树 (DRBTree.java)

实现红黑二叉树，并进行删除操作。

红黑树二叉树的四大性质：

- \*红黑树节点的颜色非红即黑；
- \*红色节点的两个子节点必须是黑色；
- \*叶子都为黑色（这里的叶子节点是指 NULL 节点）；
- \*每个节点到叶子节点的所有路径包含的黑色节点个数要相同。

根据以上性质完成红黑二叉树的实现及删除等操作。

\*插入时需要重新进行红黑树平衡的三种情况，假设父节点是左孩子，叔父节点是右孩子，反之为对称情况好理解：

(1) 父亲节点是红色，叔父节点也是红色

需要转化为：父节点涂黑，叔父节点也涂黑，祖父节点涂红，把祖父节点赋值为当前节点，然后变为下面 2, 3 情况一种

(2) 当前节点为父节点的右孩子，叔父节点为黑色

需要转化为：把父节点左旋，当前节点赋值为父节点，父节点赋值为当前节点，此时变为第 3 种情况

(3) 当前节点是父节点的左孩子，叔父节点为黑色

需要转化为：父节点涂黑，祖父节点涂红并右旋，此时红黑树变为平衡状态

```
public void insertFixUp(RBNode<T> node) {
    RBNode<T> parent, gparent;
    while(((parent = parentOf(node)) != null) && isRed(parent)) {
        gparent = parentOf(parent);
        if(gparent.leftChild == parent) {
            RBNode<T> uncle = gparent.rightChild;
            if(isRed(uncle)) {
                setBlack(parent);
                setBlack(uncle);
                setRed(gparent);

                node = gparent;
                continue;
            } else {
                if(parent.rightChild == node) {
                    leftRotate(parent);
                    RBNode<T> temp = node;
                    node = parent;
                    parent = temp;
                }

                setBlack(parent);
            }
        }
    }
}
```



```

        setRed(gparent);
        rightRorate(gparent);

    }
} else {

    RBNode<T> uncle = gparent.leftChild;
    if(isRed(uncle)) {
        setBlack(parent);
        setBlack(uncle);
        setRed(gparent);

        node = gparent;
        continue;
    } else {
        if(parent.leftChild == node) {
            rightRorate(parent);
            RBNode<T> temp = node;
            node = parent;
            parent = temp;
        }
        setBlack(parent);

        setRed(gparent);
        leftRorate(gparent);
    }
}

if(mroot == node) {
    setBlack(node);
}
}

```

\*删除需要进行的四种情况(删除的节点可以由后继节点来填补,后继节点一般是大于被删除节点的最小节点,所以真正需要平衡转化的地方是后继节点那里)

假设当前节点( )是左孩子,兄弟节点是右孩子,反之为对称情况好理解

(1) 当前节点为黑色包含空的情况(大多数为空),兄弟节点为红色

需要转化为:兄弟涂黑,父亲涂红并左旋,如果变成2情况,则不会转化3,4情况,重新循环,如果变化为3情况,则会转化为第4种情况,如果变化为4情况,红黑树重新平衡并跳出循环

(2) 当前节点为黑色包含空的情况(大多数为空),兄弟节点为黑,并且兄弟节点的左孩子和右孩子都为黑

需要转化为:兄弟节点涂红,父节点赋值为当前节点,重新循环

(3) 当前节点为黑色包含空的情况(大多数为空),兄弟节点为黑,并且兄弟左孩子为红色,右孩子为黑色

需要转化为：兄弟节点涂红，左孩子涂黑，并把兄弟节点右旋 则转化为第 4 种情况

(4) 当前节点为黑色包含空的情况（大多数为空），兄弟节点为黑色，并且兄弟右孩子为红色，左孩子为任一颜色

需要转化为：兄弟节点涂成父节点的颜色，父节点涂黑，兄弟右孩子涂黑，并把父节点左旋，红黑树达到平衡状态

```
public void deleteFixUp(RBNode<T> node, RBNode<T> parent) {

    RBNode<T> other;
    while(isBlack(node)&&node!=this.mroot) {

        if(parent.leftChild == node) {
            other = parent.rightChild;
            if(isRed(other)) {
                setRed(parent);
                setBlack(other);
                leftRotate(parent);
                continue;
            }else {
                if(isBlack(other.leftChild)&&isBlack(other.rightChild)) {
                    setRed(other);
                    node = parent;
                    parent = parentOf(node);

                }else if(isRed(other.leftChild)&&isBlack(other.rightChild)) {
                    setRed(other);

                    node = parent;
                    parent = parentOf(node);

                }else if(isRed(other.leftChild)&&isBlack(other.rightChild)) {
                    setRed(other);
                    setBlack(other.leftChild);
                    rightRotate(other);
                }else if(isRed(other.rightChild)) {
                    setColor(other, colorOf(parent));
                    setBlack(parent);
                    setBlack(other.rightChild);
                    leftRotate(parent);
                    break;
                }
            }
        }else {
            other = parent.leftChild;
            if(isRed(other)) {
                setBlack(other);
```





#### 4、B+树 (BplusTree.java、Test.java、Product.java)

实现 B+数，对过程进行详细分析。

\*首先考虑的问题是数据类型，用来作为 B+树索引的键的肯定是某个拥有很多个属性的对象，那么数据类型应该使用泛型；

\*类型确定用泛型之后，接下来应该思考 B+树实现应该有哪些类，按照写二叉树的经验，首先考虑到的就是节点类。每个节点应该有一系列的键，键的数量取决于多种因素，那么最好采用数组。其实还应该有指向父节点和子节点的指针，其中父节点只需要有一个，而子节点有多个，同样需要采用数组。最好还有一个变量来方便得记录子节点和键的数量，这样获取节点数量时比较方便。

\*在确定了节点的属性之后，要考虑节点类会有哪些方法，首先构造方法肯定需要有的，在构造方法中完成相关属性的初始化。

节点分为叶节点和非叶节点，叶节点需要额外存储数据，所以数据结构不太一样，非叶节点也有自己的查询和插入逻辑，所以应该把节点类作为抽象类，叶节点和非叶节点都继承这个类。

非叶节点的查询是遍历整个数组，找到对应的子节点然后进行递归查询。非叶节点的插入同样也是找到应该插入的子树进递归插入。

叶节点需要专门定义一个数组用来存储键对应的值，还需要实现具体的查询和插入方法。查询可以顺序查询，这里采用二分搜索来节约点时间。插入就比较复杂了，因为插入叶节点时需要保证整个 B+树的平衡。

叶节点进行插入时，首先找到数组里面应该插入的数据的正确位置，然后把数组挪一挪把数据放进来。挪完数组后，就需要判断数据数量是否超过上限了，如果超过上限，则需要对当前节点进行分裂，奇数时左边比右边少一个数据，偶数时都一样。分裂成两个节点并且完成数据复制后，还没有结束，需要把新产生的节点插入父节点，所以非叶节点需要有一个方法来处理这种情况。因为插入父节点的同时还需要更新子节点指针，所以干脆把两个节点作为参数传过去。同时，父节点键也有很多，为了精准地进行插入，我们还需要传入旧键来弄清楚插入的两个节点应该放到哪里。

父节点的插入方法拿到了旧键和需要插入的两个节点，那么首先根据旧键找到了应该插入的位置，然后新的两个键取代旧的一个键，新的两个指针取代旧的指针，又是疯狂地挪位置。完成插入之后，和叶节点的插入类似，也需要判断是否超出上限了，如果超出了上限那么同样需要进行拆分，拆分也和子节点拆分类似，拆完了递归调用自身，这样就能够保证 B+树始终是平衡的。

```

public class BplusTree <T, V extends Comparable<V>>{
    //B+树的阶
    private Integer bTreeOrder;
    //B+树的非叶子节点最小拥有的子节点数量 (同时也是键的最小数量)
    //private Integer minNumber;
    //B+树的非叶子节点最大拥有的节点数量 (同时也是键的最大数量)
    private Integer maxNumber;

    private Node<T, V> root;

    private LeafNode<T, V> left;

    //无参构造方法, 默认阶为3
    public BplusTree(){
        this(3);
    }

    //有参构造方法, 可以设定B+树的阶
    public BplusTree(Integer bTreeOrder){
        this.bTreeOrder = bTreeOrder;
        //this.minNumber = (int) Math.ceil(1.0 * bTreeOrder / 2.0);
        //因为插入节点过程中可能出现超过上限的情况, 所以这里要加1

        this.maxNumber = bTreeOrder + 1;
        this.root = new LeafNode<T, V>();
        this.left = null;
    }

    //查询
    public T find(V key){
        T t = this.root.find(key);
        if(t == null){
            System.out.println("不存在");
        }
        return t;
    }

    //插入
    public void insert(T value, V key){
        if(key == null)
            return;
        Node<T, V> t = this.root.insert(value, key);
        if(t != null)

```

```

        this.root = t;
        this.left = (LeafNode<T, V>)this.root.refreshLeft();

        System.out.println("插入完成,当前根节点为:");
        for(int j = 0; j < this.root.number; j++) {
            System.out.print((V) this.root.keys[j] + " ");
        }
        // System.out.println();
    }

    /**
     * 节点父类,因为在B+树中,非叶子节点不用存储具体的数据,只需要把索引作为键就可以了
     * 所以叶子节点和非叶子节点的类不太一样,但是又会公用一些方法,所以用Node类作为父类,
     * 而且因为要互相调用一些公有方法,所以使用抽象类
     *
     * @param <T> 同BPlusTree
     * @param <V>
     */
    abstract class Node<T, V> extends Comparable<V>>{
        //父节点
        protected Node<T, V> parent;
        ---
    }

```

之后是关于节点父类、叶子节点类、非叶子节点类的具体实现,详情见代码。

运行结果:

```

os@os-VirtualBox: ~/桌面/B+树2
os@os-VirtualBox:~/桌面/B+树2$ java -cp ./bin Test
叶子节点,插入key: 0
插入完成,当前节点key为:
0
叶子节点,插入key: 0,不需要拆分
插入完成,当前根节点为:
0 叶子节点,插入key: 1
插入完成,当前节点key为:
0 1
叶子节点,插入key: 1,不需要拆分
插入完成,当前根节点为:
0 1 叶子节点,插入key: 2
插入完成,当前节点key为:
0 1 2
叶子节点,插入key: 2,不需要拆分
插入完成,当前根节点为:
0 1 2 叶子节点,插入key: 3
插入完成,当前节点key为:
0 1 2 3
叶子节点,插入key: 3,不需要拆分
插入完成,当前根节点为:
0 1 2 3 叶子节点,插入key: 4
插入完成,当前节点key为:
0 1 2 3 4
叶子节点,插入key: 4,需要拆分
叶子节点,插入key: 4,父节点为空 新建父节点
非叶子节点,插入key: 1 4
非叶子节点,插入key: 1 4直接插入
插入完成,当前根节点为:
1 4 非叶子节点查找key: 5
叶子节点,插入key: 5
插入完成,当前节点key为:
2 3 4 5

```

## 四、实验总结

本次实验使用 Java 实现了数据结构四种经典构造及其应用,加深了我对数据结构的理解及对 Java 语言的使用,也让我对 Linux 环境更为熟悉,提高了我的学习能力及动手能力。