1. Basic Function
   1. 红黑树
      1. 类与接口：
         1. RBTreeInterface<Interface>

接口，包含红黑树的相关操作的方法：插入、删除、单个搜索、范围搜索、遍历打印

* + - 1. Node<Class>

节点类；

属性：颜色、key、value、父节点、左孩子、右孩子；

方法：所有属性的getter和setter

* + - 1. RBTree<Class>

继承RBTreeInterface接口，实现其中的方法；

属性：root节点、size；

全局变量：NIL，是每一个叶节点的虚拟子节点；

方法：实现了RBTreeInterface接口中的方法

* + 1. 方法的实现：
       1. RBInsert（Node node）：

参数：将要插入的节点对象；

返回值：int类型，返回1表示replace，返回0表示insert；

内部实现：

1. 根据红黑树的原理，每次将要插入的节点颜色都为红色。
2. 首先找到节点是否存在，如果已经存在，将节点的值设为新节点的值，返回1；如果不存在，找到节点应该插入的位置。
3. 插入通过改变父节点与子节点的指针实现。
4. 插入之后，修复：

如果插入节点的父亲是左孩子：

Case 1：插入节点的uncle是红色的，换颜色，父节点的节点是新的需要修复的节点

Case 2：插入节点的uncle是黑色但是插入节点是右孩子，左旋，转换为case 3

Case 3：插入节点是黑色的但是插入节点是左孩子，右旋

如果插入节点的父亲是右孩子：

与上面情况左右相反

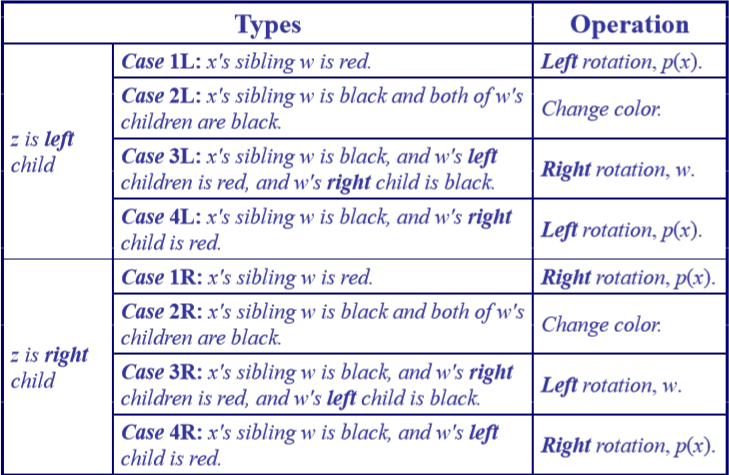
* + - 1. RBDelete（String key）：

参数：删除的key值；

返回值：int类型，key不存在返回-1，key已成功删除返回0；

内部实现：

1. 首先搜索红黑树，判断该节点是否存在，如果不存在，返回，如果存在，删除，如果前驱为红色，用前驱替换，否则，用后继替换
2. 修复：



* + - 1. RBSearch（Node x，String key）：

参数：搜索开始的节点x，搜索的key值；

返回值：Node类型，搜索到的节点，如果不存在，返回NIL；

内部实现：递归，自上而下搜索：

如果遍历结束仍然没有找到，返回此时的节点NIL；

如果找到，返回该节点，结束；

否则判断应该往左搜索还是往右搜索，递归调用

* + - 1. RBSearch（Node root，String start，String end）：

参数：搜索开始的节点，范围开始的key值，范围结束的key值；

返回值：ArrayList<Node>类型，包含搜索范围内的所有节点；

内部实现：

1. 首先判断树是否为空，如果为空，返回null
2. 如果start为空，将start设为树中最小值，如果end为空，将end设为树中最大值
3. 如果start与end相同，调用search（Node,String）方法，并将结果存入ArrayList中返回
4. 如果start小于end，in-order遍历，遍历时从start开始，start节点只遍历右支，end节点只遍历左支，将遍历到的节点add进ArrayList
5. 如果start大于end，调用search（root，end，start）
   * + 1. RBPreorderTreeWalk（Node root）：

参数：遍历开始的节点；

返回值：void；

内部实现：如果root不为空，打印root，遍历左，遍历右，一直递归直到遇到NIL

对每一个节点的相关信息的处理：level：从当前节点向上直至遇到root；child：直接判断是父节点的左孩子还是右孩子

* 1. B+树：
     1. 类与接口：
        1. BTreeInterface<Interface>

接口，包含B+树的相关操作的抽象方法：插入、删除、单个搜索、范围搜索、遍历打印

* + - 1. Entry<Class>

词条类；

属性：key、value；

方法：属性的getter和setter

* + - 1. Node<Class>

节点类；

属性：isLeaf、isRoot、节点内部的对象（备注：ArrayList类型，如果为叶节点，存Entry对象，否则存的是key），子节点列表，父节点，前节点，后节点（备注：因为所有叶节点组成一个链表，所以只有叶节点有前节点和后节点）

方法：属性的getter和setter

* + - 1. BTree<Class>

继承BTreeInterface接口，实现其中的方法；

属性：degree（≥2）、root、size；

方法：实现BTreeInterface接口中的方法

* + 1. 方法的实现：
       1. BInsert（Entry entry）：

参数：将要插入词条的对象；

返回值：int 类型，insert返回0，replace返回1；

内部实现：

1. 首先判断root是否为空，如果为空，new Node
2. 之后判断是否只有root节点且root节点是否可以直接插入，如果可以，插入后返回相应的值；
3. 找到应当插入的节点，插入，如果只是replace，返回1，结束；否则，继续；
4. 如果插入后，该节点的对象的数大于2倍的degree，通过分裂节点树向上增长修复，需要改变的指针有子节点、父节点、前节点、后节点，如果分裂出的父节点是root节点，还需要改变tree的root属性
   * + 1. BDelete（String key）：

参数：需要删除的key；

返回值：int 类型，如果没有找到该词条，返回-1，否则返回0；

内部实现：

1. 首先搜索词条，如果不存在，返回-1，否则继续；
2. 找到词条所在的叶节点，从下向上遍历，将所有需要merge的节点和最上面的merge节点、以及它父节点、父节点的兄弟节点存起来；
3. 首先修复最后存起来的节点，如果最上面的节点不是叶节点的父节点，并且不是root节点，通过旋转修复；
4. 对之前存起来的需要merge的节点处merge
5. 之后对叶节点操作，首先从叶节点中删除词条，如果叶节点是root，直接删除词条，如果删除之后所含有词条数为0，root设为null，结束；
6. 判断叶节点的size是否小于degree-1，如果小于，则通过merge修复，此时的merge与上面不同的是，因为父节点中的值在子节点中都会出现，所以父节点中对应的值不需要拉下来，此时搜索含有key的内部节点，如果存在，则将原来的key值改成merge后新生成节点的第一个entry的key值，如果不存在，结束；
7. 否则通过旋转修复，搜索含有key的内部节点，如果存在，若兄弟节点在叶节点前面，将key值改为向兄弟借的节点的key，否则改为当前兄弟节点的第一个词条的key
   * + 1. BSearch（Node x，String key）：

参数：搜索开始的节点，搜索的key值；

返回值：Entry类型；

内部实现：首先从上向下找到可能存在的叶节点，之后判断该叶节点中是否存在该词条，如果存在，返回该词条，否则返回null

* + - 1. BSearch（Node root，String key，String end）：

参数：搜索开始的节点，确定搜索范围的两个key值；

返回值：ArrayList<Entry>类型，包含范围内的所有词条

内部实现：

1. 首先搜索start和end是否存在，如果不存在返回null，如果start为空，设为树中最小的key值，如果end为空，设为树中最大的key值
2. 如果start与end相等，通过start在Node所处位置的index与end所处位置的index限制词条准确的边界范围，调用search（Node，start），插入ArrayList中返回
3. 如果end小于start，调用search（root，end，start）
4. 从叶节点组成的链表中往后遍历，首先找到start所在的叶节点，从此处开始往后遍历，直至遇到end，将遍历到的所有节点加入ArrayList中返回
5. 实现边界模糊搜索：调用new Node.search（String key）方法获取start下一位的index，end上一位的index一次确定准确范围：如果在该节点中存在该key值，那么返回其对应的index，否则返回的是其应当插入的位置（如果插入的话），因此每次在start对应的Node中遍历前，都需要得到index后，减去该节点的size，end则在减去之后再加一
   * + 1. BPreorderTreeWalk（Node root）：

参数：遍历开始的节点；

返回值：void；

内部实现：循环+递归遍历，首先打印当前节点的信息，之后从左往右递归打印每一个子树中的信息

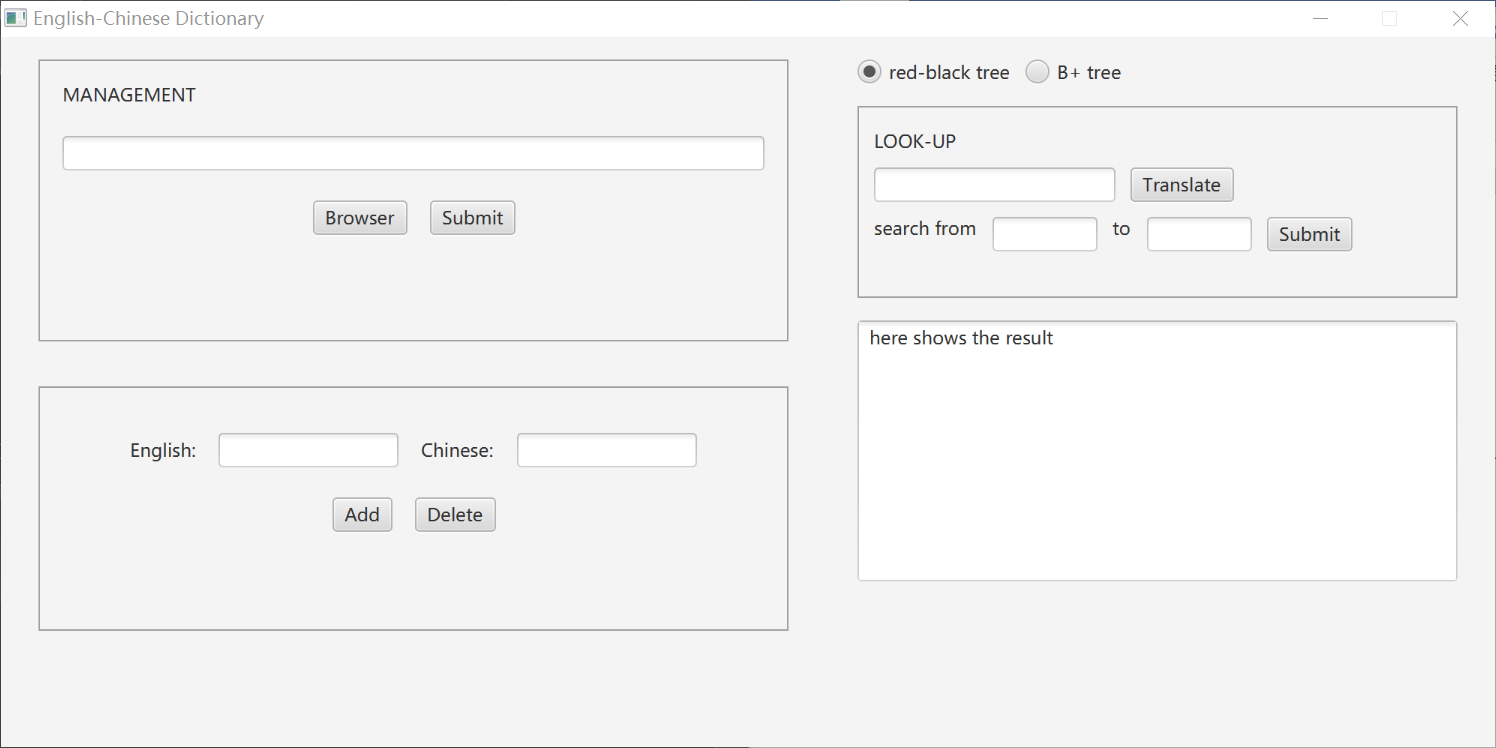
对每一节点的信息处理：level：从下往上直至遇到root节点；child：得到当前节点是父节点的第几个孩子；内部的值，如果是叶节点，遍历每一项，得到key和value，否则只得到每一项的key

1. User Interface：
   1. 类与接口：

Entrance类：

继承Application类，也是所有程序的入口

全局变量：time：操作词条的数目、RBTree的对象、BTree的对象、是否为红黑树

* 1. 实现：主要是JavaFX的用法，实现需求中的相关要求。
  2. 

1. Analysis Work：
   1. 红黑树相关数据：
      1. 插入：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 次数 | 时间（ns） | 次数 | 时间（ns） | 次数 | 时间（ns） |
| 0 | 15566528 | 17 | 388481 | 34 | 433841 |
| 1 | 9831829 | 18 | 387020 | 35 | 414962 |
| 2 | 14740381 | 19 | 385888 | 36 | 419493 |
| 3 | 11676312 | 20 | 429310 | 37 | 402879 |
| 4 | 11377646 | 21 | 394194 | 38 | 481415 |
| 5 | 426667 | 22 | 385510 | 39 | 424779 |
| 6 | 415339 | 23 | 374561 | 41 | 440637 |
| 7 | 430819 | 24 | 383622 | 42 | 414207 |
| 8 | 386265 | 25 | 377959 | 43 | 450454 |
| 9 | 373428 | 26 | 402123 | 44 | 439882 |
| 10 | 399480 | 27 | 398348 | 45 | 464425 |
| 11 | 373050 | 28 | 403646 | 46 | 483304 |
| 12 | 382490 | 29 | 429687 | 47 | 509357 |
| 13 | 411563 | 30 | 394949 | 48 | 472732 |
| 14 | 393439 | 31 | 409298 | 49 | 511244 |
| 15 | 379846 | 32 | 414584 |  |  |
| 16 | 442147 | 33 | 411941 |  |  |

平均值：1664121ns

* + 1. 删除：（第一列指是对第几个100删除）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 次数/100 | 实验一（ns） | 实验二 （ns） | 实验三（ns） | 实验四（ns） | 实验五（ns） |
| 0 | 8307 | 4908 | 6419 | 3399 | 6419 |
| 1 | 1442359 | 611303 | 1001722 | 627161 | 993793 |
| 2 | 6419 | 6797 | 6797 | 6419 | 6419 |
| 3 | 6797 | 5664 | 7552 | 3398 | 5664 |

平均值：238385ns（每删除100个词条所需的时间）

* + 1. 单个搜索：（对最小单词、中间单词、最大单词、不存在单词搜索测试）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单词 | 实验一 | 实验二 | 实验三 | 实验四 | 实验五 | 实验六 |
| Agony | 6419 | 6419 | 6041 | 5663 | 4154 | 7552 |
| gradation | 13215 | 17747 | 21522 | 14726 | 17746 | 13216 |
| Molest | 3775 | 3021 | 3775 | 3399 | 3776 | 3021 |
| sheathe | 15103 | 25676 | 13593 | 13971 | 15859 | 17369 |
| Yokel | 7929 | 5286 | 4153 | 5664 | 6041 | 6419 |
| s | 18124 | 17369 | 38513 | 15481 | 16991 | 12838 |

平均值：9742ns（Agony、Molest、Yokel三个的平均值5139ns）

* + 1. 范围搜索：（对所有的遍历搜索）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab1 | Lab2 | Lab3 | Lab4 | Lab5 | Lab6 |
| 34529019 | 25923572 | 18618892 | 32220489 | 17626987 | 17324545 |

平均值：24373917ns

* 1. B+树相关数据：

Degree=2：

* + 1. 插入：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 次数 | 时间（ns） | 次数 | 时间（ns） | 次数 | 时间（ns） |
| 0 | 18134078 | 17 | 408542 | 34 | 447056 |
| 1 | 6133047 | 18 | 494631 | 35 | 378336 |
| 2 | 7213683 | 19 | 678890 | 36 | 405522 |
| 3 | 10213186 | 20 | 914121 | 37 | 370785 |
| 4 | 5881201 | 21 | 862773 | 38 | 405145 |
| 5 | 1555256 | 22 | 485569 | 39 | 665298 |
| 6 | 758937 | 23 | 419870 | 41 | 531256 |
| 7 | 613191 | 24 | 410053 | 42 | 444035 |
| 8 | 536165 | 25 | 473486 | 43 | 446679 |
| 9 | 516531 | 26 | 386265 | 44 | 413829 |
| 10 | 432707 | 27 | 376070 | 45 | 328496 |
| 11 | 444791 | 28 | 429687 | 46 | 366254 |
| 12 | 405522 | 29 | 444791 | 47 | 393439 |
| 13 | 517664 | 30 | 366631 | 48 | 400613 |
| 14 | 405900 | 31 | 375315 | 49 | 314148 |
| 15 | 435351 | 32 | 370407 |  |  |
| 16 | 437616 | 33 | 375315 |  |  |

平均值：1412614ns

* + 1. 删除：（第一列指是对第几个100删除）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 次数/100 | 实验一（ns） | 实验二 （ns） | 实验三（ns） | 实验四（ns） | 实验五（ns） |
| 0 | 6419 | 3399 | 5664 | 2643 | 5286 |
| 1 | 3290996 | 1417816 | 1796908 | 1103291 | 2202430 |
| 2 | 5664 | 3776 | 3021 | 3021 | 3398 |
| 3 | 7174 | 4154 | 4154 | 3020 | 3375 |

平均值为：483780ns（每删除100个词条需要的时间）

* + 1. 单个搜索：（对最小单词、中间单词、最大单词、不存在单词搜索测试）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单词 | 实验一 | 实验二 | 实验三 | 实验四 | 实验五 | 实验六 |
| Agony | 22272 | 9062 | 9440 | 9062 | 8084 | 9817 |
| gradation | 18502 | 13593 | 16614 | 14348 | 13970 | 17369 |
| Molest | 9062 | 7929 | 6796 | 9440 | 9439 | 6419 |
| sheathe | 12838 | 14384 | 17746 | 15858 | 16991 | 15859 |
| Yokel | 7174 | 7551 | 5663 | 9062 | 7174 | 6796 |
| S | 13251 | 15103 | 15481 | 17746 | 17369 | 17746 |

平均值为：11610ns（Agony、Molest、Yokel三个的平均值：8902ns）

* + 1. 范围搜索：（对所有的遍历搜索）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab1 | Lab2 | Lab3 | Lab4 | Lab5 | Lab6 |
| 1006631 | 1704778 | 1856943 | 1512967 | 1828625 | 593934 |

平均值为：1249541ns

* 1. 对比分析：

1. 实验数据的分析：（红黑树与B+树同等的测试环境）

插入：B+树稍快于红黑树

删除：红黑树明显快于B+树

单个搜索：红黑树较快

范围搜索：红黑树远快于B+树

1. 可能导致产生这种差异的原因：
2. 删除：B+树的删除可能会导致三部分的变化，叶节点中的变化，含有索引的内部节点的变化，还有向下merge的变化，而红黑树的删除只是对删除节点附近产生影响，因此B+树的删除会很慢
3. 搜索：当只分析agony、molest、yokel三个单词搜索的时间时，发现红黑树明显快于B+树，这是因为红黑树的搜索并不在于对在树中靠上的节点搜索较快，而在于对每一个节点搜索的时间几乎是一样的，而红黑树对于越靠上的节点，搜索越快，这点从测试的有限的数据就可以看出来，因此，如果只测试靠近最中间的key和最小最大节点时，红黑树的优势是无法被体现的，因此如果测试更多的key值时，会发现B+树的优势会有所体现
4. 范围搜索：因为红黑树的树高降低了，并且红黑树在IO读取的时候更占优势，因此当对大范围的值搜索时，B+树占有明显的优势。
5. 关于异常值的一点猜想：
6. 删除第二个100个key的之后，发现所用的时间特别长，因此我猜想这是由于在第二个100的值中，所有值的位置都比较靠下的位置，因此每次遍历的时间就会比较长
7. 插入前面的词条的时候，用的时间较长，这是因为当树的size较小时，每插入一条，树中都会发生较大的变化
   1. 仍然可以优化的地方：

B+树的delete，在此实验中，我的算法需要从上遍历一遍找到叶节点的位置，之后从叶节点向上遍历，找到所有需要预处理的节点并保存起来，之后在对保存的值从上往下遍历，一个一个处理，最后再处理叶节点。这样子会导致多次重复的遍历。

优化方法设想：当第一次从上向下遍历时，就把需要预处理的节点先进性处理，这样会大大减少遍历次数，减少了重复的遍历。