朱奕新
19307090029

Project 1 实验报告

数据结构
2021 年 11 月 16 日

1 项目说明

1. 本次实验使用 Python 语言，实现了红黑树与 B-树。
2. 程序分为普通版和 UI 版。普通版直接把结果打印在控制台；UI 版则用一个 UI 界面与用户互动，具体使用方法请参见 Readme.md 文件。两个版本核心算法思想与实现没有本质不同，仅仅是函数返回结果的形式有差异。
3. 本实验报告以普通版代码为例进行设计的介绍。

2 红黑树的设计思路与实现细节

2.1 结点与 NIL 结点

1. 一个结点需要包括其颜色、父亲和左右孩子结点的指针，还需要包含它本身的数据，即 key, part, frequency 的值。
2. NIL 结点的颜色为黑色，其他属性并不重要。

代码 1: 红黑树结点类

```
class RBnode:
        color = BLACK
        key = None
        left = None
        right = None
        p = None
        part = None
        frequency = None

def __init__(self, newKey=None, newPart=None, newFrequency=None):
        self.key = newKey
        self.part = newPart
        self.frequency = newFrequency
```

代码 2: NIL 结点作为红黑树类的成员变量

```
class RBtree:
    root = RBnode()
```

```
3       __nil = RBnode()
```

2.2 旋转

1. 旋转用来调整红黑树的结构，分为左旋与右旋。一般用来帮助其他函数的实现。

代码 3: 左旋函数

```
1   def leftRotate(self, x):
2       y = x.right
3       x. right = y.left
4       if(y.left != self.__nil):
5           y.left.p = x
6       y.p = x.p
7       if(x.p == self.__nil):
8           self.root = y
9       elif(x == x.p.left):
10          x.p.left = y
11      else:
12          x.p.right = y
13      y.left = x
14      x.p = y
```

代码 4: 右旋函数

```
1   def rightRotate(self, y):
2       x = y.left
3       y. left = x.right
4       if(x.right != self.__nil):
5           x.right.p = y
6       x.p = y.p
7       if(y.p == self.__nil):
8           self.root = x
9       elif(y == y.p.left):
10          y.p.left = x
11      else:
12          y.p.right = x
13      x.right = y
14      y.p = x
```

2.3 插入

1. 要插入结点 z，就从根节点开始向下找到它要插入的位置，认适当的结点做父亲；然后设置自己的两个孩子为 NIL，设置自己为红色；最后对自己调用辅助程序 insertFixup 函数重新着色和旋转，使红黑树整体的性质得以保存。

2. 当父亲结点为红色时，这是不行的，就需要继续执行。假定父亲是爷爷的左孩子，insertFixup() 函数分三种情况来修正。

3. 如果叔叔为红色，就让父亲和叔叔变黑、让爷爷变红，并前往爷爷结点处继续递归。

4. 如果叔叔为黑色，且我是右孩子，则前往父亲处左旋。这种情况会转化为下一种情况。

5. 如果叔叔为黑色，且我是左孩子，则使父亲变黑，爷爷变红，并在爷爷处右旋。

6. 如果父亲是爷爷的右孩子，以上左右互换。

7. 最后，将根结点染黑。

代码 5: 插入函数

```
    def insertNode(self, z):
        y = self.__nil
        x = self.root
        while(x != self.__nil):
            y = x
            if (z.key < x.key):
                x = x.left
            elif(z.key > x.key):
                x = x.right
            else:
                print("Key", z.key, "conflict", )
                return
        z.p = y
        if(y == self.__nil):
            self.root = z
        elif(z.key < y.key):
            y.left = z
        else:
            y.right = z
```

```
        z.left = self.__nil
        z.right = self.__nil
        z.color = RED
        self.insertFixup(z)

    def insert(self, k, p, f):
        z = RBnode(k, p, f)
        self.insertNode(z)
```

代码 6: insertFixup 函数

```
    def insertFixup(self, z):
        while(z.p.color == RED):
            y = self.uncle(z)
            if(y.color == RED):
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else:
                if(z.p == z.p.p.left):
                    if(z == z.p.right):
                        z = z.p
                        self.leftRotate(z)
                    z.p.color = BLACK
                    z.p.p.color = RED
                    self.rightRotate(z.p.p)
                else:
                    if(z == z.p.left):
                        z = z.p
                        self.rightRotate(z)
                    z.p.color = BLACK
                    z.p.p.color = RED
                    self.leftRotate(z.p.p)
        self.root.color = BLACK
```

2.4 删除

1. 要删除结点 z，要用到辅助函数 transplant()，用 v 子树来替换 u 子树。

2. 删除操作类似于普通搜索树的删除，分为三种情况进行。但是删除之后要使用 deleteFixup() 函数，通过改变颜色和执行旋转来恢复红黑性质。

3. deleteFixup() 函数分四种情况来修正。当我是父亲的左子时：

4. 如果兄弟红，就染黑它，染红父亲，并在父亲左旋。去新的兄弟处递归执行。

5. 如果兄弟黑，且其两孩子都黑，则染红兄弟。去父亲递归执行。

6. 如果兄弟 w 黑，且其子左红右黑，则使其与其左子交换颜色，并在 w 处右旋。此时进入第四种情况。

7. 取新的兄弟 w。若兄弟之右子红，则染黑父亲，染黑兄弟右子，在父亲处左旋。

8. 如果我是父亲的右孩子，以上左、右互换。

9. 最后，将自己染黑。

代码 7: 删除函数

```
    def delete(self, i):
        z = self.find(i)
        if(z == self.__nil):
            print('Key', i, 'missing')
            return
        else:
            y = z
            yOriginalColor = y.color
            if(z.left == self.__nil):
                x = z.right
                self.transplant(z, z.right)
            elif(z.right == self.__nil):
                x = z.left
                self.transplant(z, z.left)
            else:
                y = self.minimum(z.right)
                yOriginalColor = y.color
                x = y.right
                if(y.p == z):
                    x.p = y
```

```
21                 else:
22                     self.transplant(y, y.right)
23                     y.right = z.right
24                     y.right.p = y
25                 self.transplant(z, y)
26                 y.left = z.left
27                 y.left.p = y
28                 y.color = z.color
29         if(yOriginalColor == BLACK):
30             self.deleteFixup(x)
```

代码 8: deleteFixup 算法

```
1     def deleteFixup(self, x):
2         while(x != self.root and x.color == BLACK):
3             if(x == x.p.left):
4                 w = x.p.right
5                 if(w.color == RED):
6                     w.color = BLACK
7                     x.p.color = RED
8                     self.leftRotate(x.p)
9                     w = x.p.right
10                if(w.left.color == BLACK and w.right.color == BLACK):
11                    w.color = RED
12                    x = x.p
13                else:
14                    if(w.right.color == BLACK):
15                        w.left.color = BLACK
16                        w.color = RED
17                        self.rightRotate(w)
18                        w = x.p.right
19                    w.color = x.p.color
20                    x.p.color = BLACK
21                    w.right.color = BLACK
22                    self.leftRotate(x.p)
23                    x = self.root
24            else:
25                w = x.p.left
26                if(w.color == RED):
27                    w.color = BLACK
```

```
                    x.p.color = RED
                    self.rightRotate(x.p)
                    w = x.p.left
                if(w.right.color == BLACK and w.left.color == BLACK):
                    w.color = RED
                    x = x.p
                else:
                    if(w.left.color == BLACK):
                        w.right.color = BLACK
                        w.color = RED
                        self.leftRotate(w)
                        w = x.p.left
                    w.color = x.p.color
                    x.p.color = BLACK
                    w.left.color = BLACK
                    self.rightRotate(x.p)
                    x = self.root
        x.color = BLACK
```

代码 9: transplant 函数

```
    def transplant(self, u, v):
        if(u.p == self.__nil):
            self.root = v
        elif(u == u.p.left):
            u.p.left = v
        else:
            u.p.right = v
        v.p = u.p
```

2.5 其他函数的实现

1. 其他函数的实现大体基于以上核心函数，完成了各种功能。具体请参见源代码。

2. 另外还配套了文件读入等辅助函数，帮助实现本 PJ 要求的功能。

3 B-树的设计思路与实现细节

3.1 数据包 dic 类和结点类

1. 用 dic 类将一份数据打包起来，放在结点类的 dics 数组中。而结点类的 child 数组中，则放置它的孩子结点的指针。

代码 10: dic 类和结点类

```
class Dic:
    def __init__(self, nkey, npart, nfrequency):
        self.key = nkey
        self.part = npart
        self.frequency = nfrequency


class Bnode:
    def __init__(self):
        self.parent = None
        self.dics = []
        self.child = []
        self.child.append(None)

    def lookUp(self, k):
        r = -1
        for i in range(0, len(self.dics)):
            if(k >= self.dics[i].key):
                r = i
            else:
                break
        return r
```

3.2 插入

1. 要插入数据，需要用到关键码查找关键码所在结点的函数 find()。查找过程类似于二叉搜索树的查找，从根结点开始逐层深入，若在本结点中能成功找到，则成功返回；否则前往下一层的另一结点。

2. 要插入数据时，先调用 find(), 若找不到则可继续插入。用 insert() 确定正确插入的位置。若关键码总数依然合法，则插入操作随即完成。

3. 否则发生关键码数量上溢，调用 insertFixup() 来修正，将该结点以某个关键码为界左右分裂，并提升该关键码。首先，若父节点存在且可接纳一个关键码，则插入到指定位置后结束；否则，在父节点处递归调用 insertFixup() 继续向上；若果真抵达根节点，则被提升的关键码自成一结点，作为新的树根。

代码 11: find() 函数

```
    def find(self, k):
        v = self.root
        self.__last = None
        while(v is not None):
            r = v.lookUp(k)
            if(r >= 0 and k == v.dics[r].key):
                return v
            self.__last = v
            v = v.child[r+1]
        return None
```

代码 12: insert() 函数

```
    def insert(self, k, p, f):
        d = Dic(k, p, f)
        now = self.find(k)
        if(now is not None):
            print('Key', k, 'conflict')
            return False
        r = self.__last.lookUp(k)
        self.__last.dics.insert(r + 1, d)
        self.__last.child.insert(r + 2, None)
        self.insertFixup(self.__last)
        return True
```

代码 13: insertFixup() 函数

```
    def insertFixup(self, x):
        if(len(x.child) <= self.__order):
            return
```

```
4           r = self.__order // 2
5           u = Bnode()
6           for i in range(0, self.__order - r - 1):
7               u.child.insert(i, x.child[r+1])
8               x.child.pop(r + 1)
9               u.dics.insert(i, x.dics[r+1])
10              x.dics.pop(r + 1)
11          u.child[self.__order - r - 1] = x.child.pop(r + 1)
12          for j in range(0, self.__order-r):
13              if(u.child[j] is not None):
14                  u.child[j].parent = u
15          p = x.parent
16          if(p is None):
17              p = Bnode()
18              self.root = p
19              p.child[0] = x
20              x.parent = p
21          t = 1 + p.lookUp(x.dics[0].key)
22          p.dics.insert(t, x.dics[r])
23          x.dics.pop(r)
24          p.child.insert(t+1, u)
25          u.parent = p
26          if(len(p.child) > self.__order):
27              self.insertFixup(p)
```

3.3 删除

1. 要删除数据，还是要用函数 find()。若找不到则操作完成。若找到则可继续删除。用 lookUp() 确定正确删除的关键码。令该关键码与其后继交换位置并删除，这样发生删除操作的一定是在叶子结点。若关键码总数依然合法，则删除操作随即完成。

2. 否则发生关键码数量下溢，调用 deleteFixup() 来修正。首先，若左兄弟存在且可给出一个关键码，则本结点向父亲借一个关键码，父亲再向左兄弟借一个关键码;，若右兄弟存在且可给出一个关键码，则本结点向父亲借一个关键码，父亲再向右兄弟借一个关键码。否则，向父亲借一个关键码，与左兄弟或右兄弟合并为一个结点。

3. 此后在父节点处递归修复。若果真到达根节点，则修复后以它唯一的孩子代替它作为

新的树根。

代码 14: delete() 函数

```
    def delete(self, k):
        v = self.find(k)
        if(v is None):
            print('Key', k, 'missing')
            return False
        r = v.lookUp(k)
        if(v.child[0] is not None):
            u = v.child[r+1]
            while(u.child[0] is not None):
                u = u.child[0]
            v.dics[r] = u.dics[0]
            v = u
            r = 0
        v.dics.pop(r)
        v.child.pop(r+1)
        self.deleteFixup(v)
        return True
```

代码 15: deleteFixup() 函数

```
    def deleteFixup(self, x):
        if((self.__order+1)//2 <= len(x.child)):
            return
        p = x.parent
        if(p is None):
            if(len(x.dics) == 0 and (x.child[0] is not None)):
                self.root = x.child[0]
                self.root.parent = None
                x.child[0] = None
            return
        r = 0
        while(p.child[r] != x):
            r = r + 1
        if(r > 0):
            lb = p.child[r-1]
            if((self.__order+1)//2 < len(lb.child)):
```

```
17                  x.dics.insert(0, p.dics[r-1])
18                  p.dics[r-1] = lb.dics.pop(len(lb.dics)-1)
19                  x.child.insert(0, lb.child.pop(len(lb.child)-1))
20                  if(x.child[0] is not None):
21                      x.child[0].parent = x
22          if(r < len(p.child)-1):
23              rb = p.child[r+1]
24              if((self.__order+1)//2 < len(rb.child)):
25                  x.dics.insert(len(x.dics), p.dics[r])
26                  p.dics[r] = rb.dics.pop(0)
27                  if(rb.child[0] is not None):
28                      rb.child[0].parent = x
29                  x.child.append(rb.child.pop(0))
30          if(r > 0):
31              lb = p.child[r-1]
32              lb.dics.append(p.dics.pop(r-1))
33              p.child.pop(r)
34              lb.child.append(x.child.pop(0))
35              if(lb.child[len(lb.child)-1] is not None):
36                  lb.child[len(lb.child)-1].parent = lb
37              while(len(x.dics) > 0):
38                  lb.dics.append(x.dics.pop(0))
39                  lb.child.append(x.child.pop(0))
40                  if(lb.child[len(lb.child)-1] is not None):
41                      lb.child[len(lb.child)-1].parent = lb
42          else:
43              rb = p.child[r+1]
44              rb.dics.insert(0, p.dics.pop(r))
45              p.child.pop(r)
46              rb.child.insert(0, x.child.pop(len(x.child)-1))
47              if(rb.child[0] is not None):
48                  rb.child[0].parent = rb
49              while(len(x.dics) > 0):
50                  rb.dics.insert(0, x.dics.pop(len(x.dics)-1))
51                  rb.child.insert(0, x.child.pop(len(x.child)-1))
52                  if(rb.child[0] is not None):
53                      rb.child[0].parent = rb
54          self.deleteFixup(p)
55          return
```

3.4 其他函数的实现

1. 其他函数的实现大体基于以上核心算法函数，完成了实验要求的各种功能。具体请参见源代码。

2. 另外还配套了文件读入等辅助函数，帮助实现本 PJ 要求的功能。

4 用户 UI 的设计与实现

1. 用户 UI 基于 python 的 tkinter 库。

2. 创建了输入输出框并绑定函数事件，让 UI 类拥有一棵树的对象，代用户操作这棵树。

3. UI 实现请参见源代码，不加赘述。

5 缓存加速搜索的设计思路

1. 设立一个新的“小树”类，小树类除了不再包含另一个小小树以外，可以和现有的树一样。在现有的树类中加入一个小树对象作为成员。

2. 当用户 search 了一个 key 值，就将这个关键码及其卫星变量加入小树中。如果调用 cacheSearch() 函数，会先在小树里找，然后再到大树里找。

3. 如果用户的需求是小范围地反复查找数据，则用 cacheSearch() 会更快（如某辅导员要在全校学生中时常查看本班学生数据）。否则调用一般的 search() 函数。