Class Red-Black Binary Search Tree Pseudo Code

Five data fields: (C# pseudo code)

① color :: System.String; → added field for additional sorting
② key :: System.object; Special note: Since the key is of type object, the user defined type of the key must have some kind of data field the is sortable either alphabetically, numerically, or based on memory address.

[In C or C++ this field would be of type void *.]

③ left :: RedBlack;
④ right :: Red Black;
⑤ parent :: Red Black;

15 Methods That Operate on the Tree; (including two constructors)

① Inorder-Tree-Walk(RedBlack x)
{
if (x ≠ null) // x can be at the root of [Tree] or any subtree node.
{ //recursively call
Inorder-Tree-Walk(left reference of x);
print key data field of x;
Inorder-Tree-Walk(right reference of x);
}
}

the return type of this method is Void.

② Tree-Search([root of Tree] RedBlack x, System.object [key to look for] k)
{
if (x = null (comparison) or k = key datafield of x)
then x has the key we're looking for (return x);
if (k < key data field of x)
then return Tree-Search(left reference of x, k);
else
{ return Tree-Search(right reference of x, k); } }

return type is RedBlack

③ Iterative-Tree-Search(root of Tree RedBlack x, node with key system.object k to look for)
return type is RedBlack

```
{
    while (x ≠ null and k ≠ key datafeild of X)
    {
        if (k < [key datafield of X] -> x.key
            x = [left reference of X;] -> x.left
        else
            x = [right reference of X;] -> x.right
    }
return x; // This search will alway be more efficient since it is not making recursive calls, except just moving memory references.
}
```

④ Tree-Minimum(any given subtree node RedBlack x)

Return Type is RedBlack

```
{
    while (x.left ≠ null)
        x = x.left;        // The key with the minimum value is always found by following the left child references.
    return x;
}
```

⑤ Tree-Maximum(any given subtree node RedBlack x)

Return Type is RedBlack

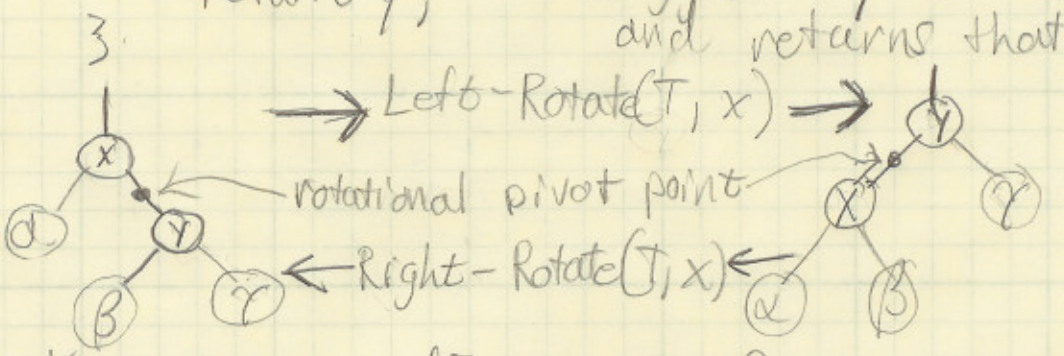
```
{
    while (x.right ≠ null)   // The key with the maximum value is always found by following the right child references.
        x = x.right;
    return x;
}
```

⑥ [Return type RedBlack]
Tree-Successor(any node RedBlack x)

```
{
    if (x.right ≠ null)
        return [Tree-Minimum(x.right)];
    RedBlack y = x.parent;
    while (y ≠ null and x == y.right)
    {
        x = y;
        y = y.parent;
    }
    return y;    // finds the smallest key greater than x.key and returns that node
}
```

⑦ [Return type RedBlack]
Tree-Predecessor(any node RedBlack x)
{
if(x.left ≠ null)
return [Tree-Maximum(x.left)];
RedBlack y = x.parent;
while(y ≠ null and x == y.left)
{
x = y;
y = y.parent;
}
return y;
}

// finds the node with the biggest key less than x.key and returns that node.



⑧ [Return type void] → moving references in memory
Left-Rotate(root or head of RedBlack T, RedBlack node in that tree[T] ⇒ x to rotate to the left)
{
RedBlack y = x.right;
x.right = x.left;
y.left.parent = x;
y.parent = x.parent;
if(x.parent == null)
root of tree T = y;
else if(x == x.parent.left)
x.parent.left = y;
else
x.parent.right = y;
y.left = x;
x.parent = y;
}

(9) [Return type void] → moving references only
RightRotate(root or head of RedBlack tree T,
node to rightrotate RedBlack x)

```
{
    RedBlack y = x.left;
    x.left = x.right;
    y.right.parent = x;
    y.parent = x.parent;
    if(x.parent == null)
        root of tree T = y;
    else if (x == x.parent.right);
        x.parent.right = y;
    else
        x.parent.left = y;
    y.right = x;
    x.parent = y;
}
```

(10) [Return type void]
RB-Insert(root or head of RedBlack tree T,
node to insert RedBlack z)

```
{
    RedBlack y = null;
    RedBlack x = root or head of tree T;
    while (x ≠ null)                    // The key of node
    {                                   z must have al-
        y = x;                          ready been filled
        if(z.key < x.key)               in. (complete
            x = x.left;                 with data and
        else                            a sortable
            x = x.right;                attribute)
    }
    z.parent = y;
    .
    .
    .
```

(next page)

```
if(y = null)
    root or head of tree T = z;
else if(z.key < y.key)
    y.left = z;
else
    y.right = z;
z.left = null;
z.right = null;
z.color = "red";
RB-Insert-Fixup(root/head of tree T, z);
} // End of RB-Insert()
```

(11) [Return type void]
RB-Insert-Fixup(root/head of tree T, RedBlack node to fixup z)

```
{
while(z.parent.color == "red")
{
if(z.parent == z.parent.paret.left)
{
RedBlack y = z.parent.parent.right;
if(y.color == "red")
{
z.parent.color = "black";
y.color = "black";
z.parent.parent.color = "red";
z = z.parent.parent;
}
else if(z == z.parent.right)
{
z = z.parent;
Left-Rotate(root/head of tree T, z);
}
z.parent.color = "black";
z.parent.parent.color = "red";
Right-Rotate(root/head of tree T,
             z.parent.parent);
}
}
```

(next page)

```
else
{
    RedBlack y = z.parent.parent.left;
    if(y.color == "red")
    {
        z.parent.color = "black";
        y.color = "black";
        z.parent.parent.color = "red";
        z = z.parent.parent;
        z.parent.color = "black";
        z.parent.parent.color = "red";
        Left-Rotate(root/head of tree T,
                    z.parent.parent);
    }
    elseif(z == z.parent.left)
    {
        z = z.parent;
        Right-Rotate(root-head of tree T, z);
    }
} //end else clause
} //end while loop
(root/head of tree T).color = "black";
} //end RB-Insert-Fixup function call
```

(12) [Return type RedBlack]
RB-Delete(root/head of treeT, Node to delete RedBlack z)
{
if(z.left == null or z.right == null)
RedBlack y = z;
else
RedBlack y = Tree-Successor(z);
if(y.left ≠ null)
x = y.left;
else
x = y.right;

x.parent = y.parent;
if(y.parent == null)
root/head of tree = x;
else if(y == y.parent.left)
y.parent.left = x;
else
y.parent.right = x;
if(y ≠ z)
{
z.key = y.key;
// any additional data items for y.key must also be copied to z.key. In other words, there must be a deep copy between the two.
}
if(y.color == "black")
RB-Delete-Fixup(root/head of treeT, x);
return y;
} // end of RB-Delete

⑬ [Return type void]
RB-Delete-Fixup(root/head of tree T, node to fixup RedBlack x)
{
while(x ≠ root/head of tree T and x.color == "black")
if (x == x.parent.left)
{
RedBlack w = x.parent.right;
if(w.color == "red")
{
w.color = "black";
x.parent.color = "red";
Left-Rotate(root/head of tree T, x.parent);
w = x.parent.right;
}
if(w.left.color == "black" and
w.right.color == "black")
{
w.color = "red";
x = x.parent;
}
else if(w.right.color == "black")
{
w.left.color = "black";
w.color = "red";
Right-Rotate(root/head of tree T, w);
w = x.parent.right;
}
w.color = x.parent.color;
x.parent.color = "black";
w.right.color = "black";
Left-Rotate(root/head of tree T, x.parent);
x = root/head of tree T;
}
else
...

(next page)

```
else
{
    RedBlack w == x.parent.left;
    if (w.color == "red")
    {
        w.color = "black";
        x.parent.color = "red";
        Right-Rotate(root/head of tree T, x.parent);
        w = x.parent.left;
    }
    if(w.right.color == "black" and
       w.left.color == "black")
    {
        w.color = "red";
        x = x.parent;
    else if(w.left.color == "black")
    {
        w.right.color = "black";
        w.color = "red";
        Left-Rotate(root/head of TreeT, w);
        w = x.parent.left;
    }
    w.color = x.parent.color;
    x.parent.color = "black";
    w.left.color = "black";
    Right-Rotate(root/head of tree T, x.parent);
    x = root/head of tree T;
    } //end if
} //end long else clause
} //end long while loop
x.color = "black";
} // end of function call RB-Delete-Fixup()
```

(14) [Return type void]
RedBlack() no arguement default constructor
{
color = "black";
key = null;
left = null;
right = null;
parent = null;
}

(15) [Return type void] → Sigle arguement constructor ←
RedBlack(System.object k) //This will insert your key k into a tree where the node you call this through is the head of the tree.
{
RedBlack X = new RedBlack();
X.key = k;
RB-Insert(this, X);
}