Algorithm 1 INSERT,DELETE,SEARCH INTERVAL TREES

```
1: procedure LEFT-ROTATE(T,x)
2:     y ← right[x]
3:     right[x] ← left[y]
4:     if left[y] ≠ NIL then
5:         p[left[y]] ← x                                   ▷ p is parent
6:     p[y] ← p[x]
7:     if p[x] ← NIL then
8:         root[T] ← y
9:     else if x = left[p[x]] then
10:        left[p[x]] ← y
11:    else
12:        right[p[x]] ← y
13:    left[y] ← x
14:    p[x] ← y
15:    max[x] = MAX(max[left[x]], max[right[x]], max[x])   ▷ reconfiguring the augmented values
16:    max[y] = MAX(max[left[y]], max[right[y]], max[y])
17:    max[p[y]] = MAX(max[left[p[y]]], max[right[p[y]]], max[p[y]])
18:
19: procedure RIGHT-ROTATE(T,y)                            ▷ analogous to LEFT-ROTATE
20:    x ← left[y]
21:    left[y] ← right[x]
22:    if right[x] ≠ NIL then
23:        p[right[x]] ← y
24:    p[x] ← p[y]
25:    if p[y] ← NIL then
26:        root[T] ← x
27:    else if y = right[p[y]] then
28:        right[p[y]] ← x
29:    else
30:        left[p[y]] ← x
31:    right[x] ← y
32:    p[y] ← x
33:    max[x] = MAX(max[left[x]], max[right[x]], max[x])
34:    max[y] = MAX(max[left[y]], max[right[y]], max[y])
35:    max[p[x]] = MAX(max[left[p[x]]], max[right[p[x]]], max[p[x]])
36:
37: procedure BST-INSERT(T,z)
38:    y ← NIL
39:    x ← root[T]
40:    while x ≠ NIL do
41:        y ← x
42:        if low[z] < low[x] then
43:            x ← left[x]
44:        else
45:            x ← right[x]
46:    p[z] ← y
47:    if y = NIL then
48:        root[T] ← z                                      ▷ z is the only node
49:    else if low[z] < low[y] then                         ▷ setting up the pointers to z
50:        left[y] ← z
51:    else
52:        right[y] ← z
53:    max[z] = high[z]                                     ▷ setting up augmented value of inserted node
```

```
55: procedure RB-INSERT(T,x)                          ▷ inserting a node in interval(Red Black) tree
56:     BST − INSERT(T, x)                            ▷ First insert x as normally inserted into BST and color it red
57:     color[x] ← RED
58:     while x ≠ root[T] and color[p[x]] = RED do
59:         if p[x] = left[p[p[x]]] then
60:             y ← right[p[p[x]]]                                                    ▷ uncle
61:             if color[y] = RED then                                                ▷ Case a
62:                 color[p[x]] ← BLACK
63:                 color[y] ← BLACK
64:                 color[p[p[x]]] ← RED
65:                 x ← p[p[x]]                                                       ▷ Change x to grandparent
66:             else if x = right[p[x]] then                                          ▷ Case b:Left Right Case
67:                 x ← p[x]
68:                 LEFT − ROTATE(T, x)
69:                 color[p[x]] ← BLACK                                               ▷ Follow Case b: Left Left Case
70:                 color[p[p[x]]] ← RED
71:                 RIGHT − ROTATE[T, p[p[x]]]
72:             else                                                                  ▷ Case b: Left Left Case
73:                 color[p[x]] ← BLACK
74:                 color[p[p[x]]] ← RED
75:                 RIGHT − ROTATE[T, p[p[x]]]
76:         else
77:             (do the same thing in then in line 94 clause with "right" and "left" swapped)   ▷ Case b: Right
    Left and Right Right Case
78:     color[root[T]] ← BLACK                                                        ▷ Since root is always black
79:
80: procedure RB-DELETE(T,z)                          ▷ Deleting a node in RB-Tree
81:     if left[z] = nil[T] or right[z] = nil[T] then
82:         y ← z
83:     else
84:         y ← RB − SUCCESSOR(z)
85:     if left[y] ≠ nil[T] then
86:         x ← left[y]
87:     else
88:         x ← right[y]
89:     p[x] ← p[y]
90:     if p[y] = nil[T] then
91:         root[T] ← x
92:     else if y = left[p[y]] then
93:         left[p[y]] ← x
94:     else
95:         right[p[y]] ← x
96:     if y ≠ z then
97:         low[z] ← low[y]
98:     if color[y] = BLACK then
99:         RB − DELETE − FIXUP(T, x)
        return y
100: procedure RB-SUCCESSOR(x)                        ▷ helper for finding successor for a node in tree
101:     if right[x] ≠ NIL then return RB − MINIMUM(right[x])
102:     y ← p[x]
103:     while y ≠ NIL and x = right[y] do
104:         x ← y
105:         y ← p[y]
        return y
```

```
111: procedure RB-DELETE-FIXUP(T,x)
112:     while x ≠ root[T] and color[x] = BLACK do
113:         if x = left[p[x]] then
114:             w ← right[p[x]]
115:             if color[x] = RED then
116:                 color[w] ← BLACK
117:                 color[p[x] ← RED
118:                 LEFT − ROTATE(T, p[x])
119:                 w ← right[p[x]]
120:             if color[left[w]] = BLACK and color[right[w]] = BLACK then
121:                 color[w] ← RED
122:                 x ← p[x]
123:             else if color[right[w]] = BLACK then
124:                 color[left[w] ← BLACK
125:                 color[w] ← RED
126:                 RIGHT − ROTATE(T, w)
127:                 w ← right[p[x]]
128:             else
129:                 color[w] ← color[p[x]]
130:                 color[p[x]] ← BLACK
131:                 color[right[w]] ← BLACK
132:                 LEFT − ROTATE(T, p[x])
133:                 x ← root(T)
134:         else
135:             (same as then clause with "right" and "left" exchanged)
136:     color[x] ← BLACK
137:
```

```
138: procedure SEARCH(root,interval)          ▷ interval to be searched has attributes low and high
139:     if root = NULL then return NULL
140:     if Interval[root].low ≤ interval.high and interval.low ≤ Interval[root].high then          ▷ Checking for overlaps return Interval[root]
141:     if left[root] ≠ NULL and max[left[root]] ≥ interval.low then return SEARCH(left[root], interval)▷ interval may overlap with an interval in left subtree
         return SEARCH(right[root], interval)          ▷ Otherwise recur for right subtree
```