1.6 Pseudo Codes

```
LEFT-ROTATE(T, x)
1  y = x.right
2  x.right = y.left           // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil          // if y's left subtree is not empty ...
4      y.left.p = x           // ... then x becomes the parent of the subtree's root
5  y.p = x.p                  // x's parent becomes y's parent
6  if x.p == T.nil            // if x was the root ...
7      T.root = y             // ... then y becomes the root
8  elseif x == x.p.left       // otherwise, if x was a left child ...
9      x.p.left = y           // ... then y becomes a left child
10 else x.p.right = y         // otherwise, x was a right child, and now y is
11 y.left = x                 // make x become y's left child
12 x.p = y
```

```
RB-INSERT(T, z)
1  x = T.root                 // node being compared with z
2  y = T.nil                  // y will be parent of z
3  while x ≠ T.nil            // descend until reaching the sentinel
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y                    // found the location—insert z with parent y
9  if y == T.nil
10     T.root = z             // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
14 z.left = T.nil             // both of z's children are the sentinel
15 z.right = T.nil
16 z.color = RED              // the new node starts out red
17 RB-INSERT-FIXUP(T, z)      // correct any violations of red-black properties
```

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left           // is z's parent a left child?
3          y = z.p.p.right            // y is z's uncle
4          if y.color == RED          // are z's parent and uncle both red?
5              z.p.color = BLACK      ⎫
6              y.color = BLACK        ⎬ case 1
7              z.p.p.color = RED      ⎪
8              z = z.p.p              ⎭
9          else
10             if z == z.p.right
11                 z = z.p            ⎫ case 2
12                 LEFT-ROTATE(T, z)  ⎭
13             z.p.color = BLACK      ⎫
14             z.p.p.color = RED      ⎬ case 3
15             RIGHT-ROTATE(T, z.p.p) ⎭
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

```
RB-DELETE(T, z)
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)      // replace z by its right child
6  elseif z.right == T.nil
7      x = z.left
8      RB-TRANSPLANT(T, z, z.left)       // replace z by its left child
9  else y = TREE-MINIMUM(z.right)        // y is z's successor
10     y-original-color = y.color
11     x = y.right
12     if y ≠ z.right                    // is y farther down the tree?
13         RB-TRANSPLANT(T, y, y.right)  // replace y by its right child
14         y.right = z.right             // z's right child becomes
15         y.right.p = y                 //     y's right child
16     else x.p = y                      // in case x is T.nil
17     RB-TRANSPLANT(T, z, y)            // replace z by its successor y
18     y.left = z.left                   // and give z's left child to y,
19     y.left.p = y                      //     which had no left child
20     y.color = z.color
21 if y-original-color == BLACK       // if any red-black violations occurred,
22     RB-DELETE-FIXUP(T, x)          //     correct them
```

```
RB-DELETE-FIXUP(T, x)
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left                 // is x a left child?
3          w = x.p.right                // w is x's sibling
4          if w.color == RED
5              w.color = BLACK          ⎫
6              x.p.color = RED          ⎬ case 1
7              LEFT-ROTATE(T, x.p)      ⎪
8              w = x.p.right            ⎭
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED            ⎫ case 2
11             x = x.p                  ⎭
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK ⎫
15                 w.color = RED        ⎬ case 3
16                 RIGHT-ROTATE(T, w)   ⎪
17                 w = x.p.right        ⎭
18             w.color = x.p.color      ⎫
19             x.p.color = BLACK        ⎪
20             w.right.color = BLACK    ⎬ case 4
21             LEFT-ROTATE(T, x.p)      ⎪
22             x = T.root               ⎭
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

2 B Tree

2.1 Pseudo Codes

```
B-TREE-SPLIT-CHILD(x, i)
y = x.c_i                          // full node to split
z = ALLOCATE-NODE()                // z will take half of y
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1                 // z gets y's greatest keys ...
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t                 // ... and its corresponding children
        z.c_j = y.c_{j+t}
y.n = t - 1                        // y keeps t - 1 keys
for j = x.n + 1 downto i + 1       // shift x's children to the right ...
    x.c_{j+1} = x.c_j
x.c_{i+1} = z                      // ... to make room for z as a child
for j = x.n downto i               // shift the corresponding keys in x
    x.key_{j+1} = x.key_j
x.key_i = y.key_t                  // insert y's median key
x.n = x.n + 1                      // x has gained a child
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

```
B-TREE-SPLIT-ROOT(T)
s = ALLOCATE-NODE()
s.leaf = FALSE
s.n = 0
s.c_1 = T.root
T.root = s
B-TREE-SPLIT-CHILD(s, 1)
return s
```

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf                              // inserting into a leaf?
    while i ≥ 1 and k < x.key_i        // shift keys in x to make room for k
        x.key_{i+1} = x.key_i
        i = i - 1
    x.key_{i+1} = k                    // insert key k in x
    x.n = x.n + 1                      // now x has 1 more key
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i       // find the child where k belongs
        i = i - 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t - 1               // split the child if it's full
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i                 // does k go into x.c_i or x.c_{i+1}?
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t - 1
    s = B-TREE-SPLIT-ROOT(T)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

2.2 Rules

Node	Min	Max Deg	Min	Max Keys
Root	0	$2t$	1	$2t-1$
Internal	t	$2t$	$t-1$	$2t-1$
Leaf	0		$t-1$	$2t-1$

2.3 Insertion

Start from root, split any full nodes. Then we can directly insert.

2.4 Deletion

Case 1 At leaf — delete

Case 2 Found in internal.

- Case 2a Preceding child has t keys: steal.
- Case 2b Succeeding child has t keys: steal.
- Case 2c Adjacent children have $t-1$ keys: merge into a $(2t-1)$-key node, and recurse.

Case 3 Not found yet — ensure next node has t node for safe Deletion

- Case 3a Preceding sibling has at least t keys: steal.
- Case 3b Succeeding child has at least t keys: steal.
- Case 3c Adjacent children have $t-1$ keys: merge into a $(2t-1)$-key node, and recurse.

In case 2c and 3c, root can be empty after operation. We remove it.

3 Disjoint Set

Method	Description	MAKE-SET	FIND	UNION
Array	Keep set ID	?	$O(1)$	$O(n)$
Tree	Keep tree	?	$O(n)$	$O(n)$
Linked-list	Keep head	$O(1)$	$O(1)$	$O(n)$
Array Small-to-Large		?	$O(1)$	$O(n)$ worst case, $O(\log n)$ amortized
Linked-list Small-to-Large		?	$O(1)$	$O(n)$ worst case, $O(\log n)$ amortized
Tree Small-to-Large		?	$O(\log n)$	$O(\log n)$
Full		$O(1)$	$O(\log n)$ worst case, $O(\alpha(n))$ amortized	$O(\log n)$ worst case, $O(\alpha(n))$ amortized

4 Hashing

- **Direct-address tables**: Array ($h(x) = x$).
- ***Uniform* hash function**: Probability of any probing sequence is the same (on slide).
- ***Simple uniform* hash function**: $P(h(x) = a) = |U|^{-1}$
- **Load factor** α: keys / slots
- **Division method**: $h(x) \equiv k \mod m$
- **Multiplication method**: $h(x) = \lfloor m(kA \mod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
- **Quadratic probing**: $h(x, i) = (h'(x) + ai + bi^2) \mod m$
- **Double hashing**: $h(x, i) = (h_1(x) + ih_2(x)) \mod m$
- **Primary clustering**: Consecutive filled slots produced by open addressing probing
- **Secondary clustering**: IDK
- **Dynamic hashing using directories**: (left) When overflow occurs, duplicate table with unchanged pointers. Lazy resolve correct new hash until touched.
- **Directoryless Dynamic hashing**: (right) Lazy resolve collision, branch cell from 0 to full, and starts from 0 again (space doubled every scan), collision is solved only when the cell is duplicated.