PSEUDOCODE

Breadth-First Search

```
BFS(G, s)
    for each vertex u ∈ G.V − {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                ENQUEUE(Q, v)
        u.color = BLACK
```

Depth-First Search

```
DFS(G)
    for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    for each vertex u ∈ G.V
        if u.color == WHITE
            DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v. π = u
            DFS-VISIT(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

```
STRONGLY-CONNECTED-COMPONENTS(G)
1   call DFS(G) to compute finishing times u.f for each vertex u
2   compute G^T
3   call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing
    u.f (as computed in line 1)
4   output the vertices of each tree in the depth-first forest formed in line 3 as a separate
    strongly connected component
```

Heaps

```
PARENT(i)
    return ⌊i/2⌋
```

```
LEFT(i)
    return 2i
```

```
RIGHT(i)
    return 2i + 1
```

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length / 2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size – 1
        MAX-HEAPIFY(A, 1)
```

```
HEAP-MAXIMUM(A)
    return A[1]
```

```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size – 1
    MAX-HEAPIFY(A, 1)
    return max
```

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = –∞
    HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Disjoint-Set Forests

```
MAKE-SET(x)
    x.p = x
    x.rank = 0

```

```
FIND-SET(x)
    if x ≠ x.p
        x.p = FIND-SET(x.p)
    return x.p
```

```
LINK(x, y)
    if x.rank > y.rank
        y.p = x
    else x.p = y
        if x.rank == y.rank
            y.rank = y.rank + 1
```

```
UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))
```

Minimum Spanning Trees

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ G.V
        MAKE-SET(v)
    sort the edges of G.E into nondecreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
    return A
```

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u. π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if  v ∈  Q and w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```

Shortest Paths in Weighted Graphs

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Hash Tables Using Open Addressing

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL or T[j] == DELETED
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

```
HASH-DELETE(T, k)
i = HASH-SEARCH(T, k)
if i ≠ NIL
    T[i] = DELETED
```

Red-Black Trees

```
LEFT-ROTATE(T, x)
    y = x.right
    x.right = y.left
    if y.left ≠ T.nil
        y.left.p = x
    y.p = x.p
    if x.p == T.nil
    T.root = y
    elseif x == x.p.left
    x.p.left = y
    else x.p.right = y
    y.left = x
    x.p = y
```

```
RB-TRANSPLANT(T, u, v)
    if u.p == T.nil
        T.root = v
    elseif u == u.p.left
        u.p.left = v
    else u.p.right = v
    v.p = u.p
```

```
RB-INSERT(T, z)
    y = T.nil
    x = T.root
    while x ≠ T.nil
        y = x
        if  z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == T.nil
        T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
    while z.p.color == RED
        if z.p == z.p.p.left
            y = z.p.p.right
            if y.color == RED
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else if z == z.p.right
                    z = z.p
                    LEFT-ROTATE(T, z)
                z.p.color = BLACK
                z.p.p.color = RED
                RIGHT-ROTATE(T, z.p.p)
        else (same as then clause with “right” and “left” exchanged)
    T.root.color = BLACK
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            LEFT-ROTATE(T, x.p)
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                RIGHT-ROTATE(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            LEFT-ROTATE(T, x.p)
            x = T.root
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```