Algoritmi di Alberi e Grafi

Alberi Binari

Visita Inorder

```
procedure Visita_InOrder(T)
    if T != NIL then
        Visita_InOrder(T.left)
        Visita(T)
        Visita_InOrder(T.right)
end procedure
```

Visita Preorder

```
procedure Visita_PreOrder(T)
    if T != NIL then
        Visita(T)
        Visita_PreOrder(T.left)
        Visita_PreOrder(T.right)
end procedure
```

Visita Postorder

```
procedure Visita_PostOrder(T)
    if T != NIL then
        Visita_PostOrder(T.left)
        Visita_PostOrder(T.right)
        Visita(T)
end procedure
```

DFS / BFS su albero binario

```
procedure BFS(T)
    if T == NIL then return
    Q = {T}
    while Q != {} do
        x = Dequeue(Q)
        Visita(x)
        if x.left != NIL then
            Q = Enqueue(Q, x.left)
        if x.right != NIL then
            Q = Enqueue(Q, x.right)
```

```
end procedure

procedure Search(T, k)
    if T != NIL then
        if T.key < k then
            return Search(T.right, k)
        else if T.key > k then
            return Search(T.left, k)
        else
            return T
    return NIL
end procedure
```

Alberi AVL

Inserimento

```
procedure Insert_AVL(T, k)
    if T == NIL then
        T = Alloca_Nodo_AVL()
        T.key = k
        T.left = NIL
        T.right = NIL
        T.h = 0
    else if T.key < k then
        T.right = Insert_AVL(T.right, k)
        T = Bilancia_Destra(T)
    else if T.key > k then
        T.left = Insert_AVL(T.left, k)
        T = Bilancia_Sinistra(T)
    return T
end procedure

procedure Bilancia_Sx(T)
    if T != NIL then
        if Altezza(T.left) - Altezza(T.right) > 1 then
            if Altezza(T.left.left) >= Altezza(T.left.right) then
                T = Rotazione_Sx(T)
            else
                T = Rotazione_Doppia_Sx(T)
        else
            T.h = max(Altezza(T.left), Altezza(T.right)) + 1
    return T
end procedure

procedure Bilancia_Dx(T)
    if T != NIL then
        if Altezza(T.right) - Altezza(T.left) > 1 then
            if Altezza(T.right.right) >= Altezza(T.right.left) then
                T = Rotazione_Dx(T)
            else
```

```
                T = Rotazione_Doppia_Dx(T)
        else
            T.h = max(Altezza(T.left), Altezza(T.right)) + 1
    return T
end procedure
```

Cancellazione

```
procedure Delete_AVL(T, k)
    if T == NIL then return NIL
    if k < T.key then
        T.left = Delete_AVL(T.left, k)
        T = Bilancia_Dx(T)      // sinistra si accorcia → possibile
sbilanciamento destro
    else if k > T.key then
        T.right = Delete_AVL(T.right, k)
        T = Bilancia_Sx(T)      // destra si accorcia → possibile
sbilanciamento sinistro
    else
        T = Delete_Root(T)
    return T
end procedure

procedure Delete_Root(T)
    if T.left == NIL then return T.right
    if T.right == NIL then return T.left
    // due figli: sostituisci con successore
    tmp = Stacca_Min(T.right, T)
    T.key = tmp.key
    free(tmp)
    T = Bilancia_Sx(T)
    return T
end procedure

procedure Stacca_Min(T, P)  // restituisce il nodo minimo (per copia
chiave)
    if T.left != NIL then
        minNode = Stacca_Min(T.left, T)
        T.left = Bilancia_Dx(T.left)   // bilancia sottoalbero sinistro
modificato
        return minNode
    else
        // T è il minimo
        if P.left == T then
            P.left = T.right
        else
            P.right = T.right
        return T
end procedure
```

Rotazione Semplice a Sinistra

```
procedure Rotazione_Sx(T)
    newT = T.left
    T.left = newT.right
    newT.right = T
    T.h = max(Altezza(T.left), Altezza(T.right)) + 1
    newT.h = max(Altezza(newT.left), Altezza(newT.right)) + 1
    return newT
end procedure
```

Rotazione Semplice a Destra

```
procedure Rotazione_Dx(T)
    newT = T.right
    T.right = newT.left
    newT.left = T
    T.h = max(Altezza(T.left), Altezza(T.right)) + 1
    newT.h = max(Altezza(newT.left), Altezza(newT.right)) + 1
    return newT
end procedure
```

Rotazione Doppia Left-Right

```
procedure Rotazione_Doppia_Sx(T)
    T.left = Rotazione_Dx(T.left)
    T = Rotazione_Sx(T)
    return T
end procedure
```

Rotazione Doppia Right-Left

```
procedure Rotazione_Doppia_Dx(T)
    T.right = Rotazione_Sx(T.right)
    T = Rotazione_Dx(T)
    return T
end procedure
```

Alberi Red-Black

Inserimento

```
procedure Insert_RB(T, k)
    if not IS_NIL(T) then
        if T.key < k then
            T.right = Insert_RB(T.right, k)
            T = Bilancia_Destra_RB(T)
        else if T.key > k then
            T.left = Insert_RB(T.left, k)
            T = Bilancia_Sinistra_RB(T)
    else
        T = NewNodeRB(k)
        T.color = RED
    T.color = BLACK   // @copilot radice sempre nera
    return T
end procedure

procedure Bilancia_Sinistra_RB(T)
    if !nil(T.left) && T.left.color == RED then
        v = Tipo_Violazione_Sinistra(T.left, T.right)
        switch v do
            case 1: T = Caso1(T)
            case 2: T = Caso2(T)
            case 3: T = Caso3(T)
    return T
end procedure

procedure Tipo_Violazione_Sinistra(S, D)
    v = 0
    if S.color == RED then
        if D.color == RED then
            // CASO 1: Padre rosso e Zio rosso
            if S.left.color == RED || S.right.color == RED then
                v = 1
        else
            // Lo zio D è NERO (o NIL, che è considerato nero)
            if S.right.color == RED then
                v = 2  // CASO 2: Nodo "interno" (figlio destro del figlio
sinistro)
            else if S.left.color == RED then
                v = 3  // CASO 3: Nodo "esterno" (figlio sinistro del
figlio sinistro)
    return v
end procedure

procedure Caso1(T)
    T.right.color = BLACK
    T.left.color = BLACK
    T.color = RED
    return T
end procedure

procedure Caso2(T)
    T.left = Rotazione_Dx(T.left)
    T = Caso3(T)
```

```
    return T
end procedure

procedure Caso3(T)
    T = Rotazione_Sx(T)
    T.color = BLACK
    T.right.color = RED
    return T
end procedure
```

Cancellazione (fixed by @copilot)

```
procedure Delete_RB(T, k)
    if !nil(T) then
        if T.key < k then
            T.right = Delete_RB(T.right, k)
            T = Bilancia_Canc_Destra_RB(T)
        else if T.key > k then
            T.left = Delete_RB(T.left, k)
            T = Bilancia_Canc_Sinistra_RB(T)
        else
            T = Delete_Root_RB(T)
    return T
end procedure

procedure Delete_Root_RB(T)
    if !nil(T) then
        tmp = T
        if nil(T.left) then
            T = T.right
            if tmp.color == BLACK then
                Propagate_Black(T)
        else if nil(T.right) then
            T = T.left
            if tmp.color == BLACK then
                Propagate_Black(T)
        else
            tmp = Stacca_Min_RB(T.right, T)
            T.key = tmp.key
            T = Bilancia_Canc_Destra_RB(T)
        free(tmp)
    return T
end procedure

procedure Propagate_Black(T)
    if T.color == RED then
        T.color = BLACK
    else
        T.color = DOUBLE_BLACK
end procedure
```

```
procedure Stacca_Min_RB(T, P)
    if !nil(T.left) then
        ret = Stacca_Min_RB(T.left, T)
        T.left = Bilancia_Canc_Sinistra_RB(T.left)
        return ret
    else
        // T è il minimo
        if T.color == BLACK && !nil(T.right) then
            Propagate_Black(T.right)
        if P.left == T then
            P.left = T.right
        else
            P.right = T.right
        return T
end procedure

procedure Bilancia_Canc_Sinistra_RB(T)
    if !nil(T.right) then
        v = Violazione_Sx_Canc(T.left, T.right)
        switch v do
            case 1:
                T = Caso1_Canc(T)
                T = Bilancia_Canc_Sinistra_RB(T.left)
            case 2: T = Caso2_Canc(T)
            case 3: T = Caso3_Canc(T)
            case 4: T = Caso4_Canc(T)
    return T
end procedure

procedure Violazione_Sx_Canc(X, W)
    v = 0
    if !nil(X) && X.color == DOUBLE_BLACK then
        if W.color == RED then
            v = 1
        else if W.right.color == BLACK && W.left.color == BLACK then
            v = 2
        else if W.left.color == RED && W.right.color == BLACK then
            v = 3
        else
            v = 4
    return v
end procedure

procedure Caso1_Canc(T)
    T = Rotazione_Dx(T)
    T.color = BLACK
    T.right.color = RED
    return T
end procedure

procedure Caso2_Canc(T)
    T.right.color = RED
    T.left.color = BLACK
    Propagate_Black(T)
```

```
    return T
end procedure

procedure Caso3_Canc(T)
    T.right = Rotazione_Sx(T.right)
    T.right.color = BLACK
    T.right.right.color = RED
    T = Caso4_Canc(T)
    return T
end procedure

procedure Caso4_Canc(T)
    T = Rotazione_Sx(T)
    T.right.color = T.color
    T.color = T.left.color
    T.left.color = BLACK
    T.left.left.color = BLACK
    return T
end procedure
```

Grafi

Visita in Profondità (DFS)

```
procedure DFS(G)
    Init(G)
    for each v in V do
        if Color[v] == White then
            DFS_Visit(G, v)
end procedure

procedure DFS_Visit(G, s)
    Color[s] = Gray
    time = time + 1
    d[s] = time
    for each v in Adj(s) do
        if Color[v] == White then
            Pred[v] = s
            DFS_Visit(G, v)
    time = time + 1
    f[s] = time
    Color[s] = Black
end procedure
```

Visita in Ampiezza (BFS)

```
procedure BFS(G, s)
    if s == NIL then
```

```
        return
    for each x in V do
        Color[x] = White
        d[x] = infinity
        p[x] = NIL
    Q = {s}
    Color[s] = Gray
    d[s] = 0
    p[s] = NIL
    while Q != {} do
        x = Head(Q)
        for each v in Adj(x) do
            if Color[v] == White then
                Q = Enqueue(Q, v)
                Color[v] = Gray
                d[v] = d[x] + 1
                p[v] = x
        Q = Dequeue(Q)
        Color[x] = Black
end procedure
```

Dijkstra

```
procedure Dijkstra(G, w, s)
    Init(G, s)
    S = {}
    Q = V
    while Q != {} do
        u = Extract_Min(Q)
        Q = Q \ {u}
        S = S union {u}
        for each v in Adj[u] do
            Relax(u, v, w)
end procedure
```

Bellman-Ford

```
procedure Bellman_Ford(G, w, s)
    Init(G, s)
    for i = 1 to |V| - 1 do
        for each (u, v) in E do
            Relax(u, v, w)
    for each (u, v) in E do
        if d[v] > d[u] + w(u, v) then
            return false
    return true
end procedure
```

Rilevamento Cicli (Aciclicità)

```
procedure Init_Color(G)
    for each v in V do
        Color[v] = White
        Pred[v] = NIL
    time = 1
end procedure

procedure Aciclico(G)
    Init(G)
    for each v in V do
        if Color[v] == White then
            ret = Aciclico_Visit(G, v)
            if ret == false then
                return false
    return true
end procedure

procedure Aciclico_Visit(G, s)
    Color[s] = Gray
    for each v in Adj(s) do
        if Color[v] == White then
            ret = Aciclico_Visit(G, v)
            if ret == false then
                return false
        else if Color[v] == Gray then
            return false
    Color[s] = Black
    return true
end procedure
```