LUCIANO DI CRISTO

13/05/2022 | università degli studi di napoli “parthenope” | informatica

Algoritmi & Strutture Dati

Compendio degli algoritmi e la loro complessità

Sommario

[Algoritmi e complessità 3](#_Toc103681448)

[Algoritmi di sort: 3](#_Toc103681449)

[Insertion Sort: 3](#_Toc103681450)

[Bubble Sort: 3](#_Toc103681451)

[Merge Sort: 4](#_Toc103681452)

[Quick Sort: 5](#_Toc103681453)

[Heap Sort 6](#_Toc103681454)

[Bucket/Bin Sort: 7](#_Toc103681455)

[Counting Sort: 7](#_Toc103681456)

[Radix Sort: 7](#_Toc103681457)

[Programmazione Dinamica: 8](#_Toc103681458)

[Fibonacci: 8](#_Toc103681459)

[LCS: 9](#_Toc103681460)

[Distanza di Levenshtein: 9](#_Toc103681461)

[Matrix Chain Order e Multiply: 10](#_Toc103681462)

[Zaino 0-1: 11](#_Toc103681463)

[Fastest Way: 11](#_Toc103681464)

[Algoritmi con Memoization: 12](#_Toc103681465)

[Algoritmi Greedy: 14](#_Toc103681466)

[Greedy Activity Selector: 14](#_Toc103681467)

[Fractional Knapsack: 15](#_Toc103681468)

[Huffman: 15](#_Toc103681469)

[Grafi: 16](#_Toc103681470)

[BFS: 16](#_Toc103681471)

[DFS: 16](#_Toc103681472)

[Kruskal: 18](#_Toc103681473)

[Primm: 19](#_Toc103681474)

[Hashing 19](#_Toc103681475)

[Hash Insert: 19](#_Toc103681476)

[Hash Search: 19](#_Toc103681477)

[Hash Delete: 20](#_Toc103681478)

[Grafi cammini minimi 20](#_Toc103681479)

[Bellman-Ford: 20](#_Toc103681480)

[Dijkstra: 21](#_Toc103681481)

[Algoritmi visita e ricerca albero binario 21](#_Toc103681482)

[Tree Search: 21](#_Toc103681483)

[Tree Iterative Search: 22](#_Toc103681484)

[Tree Minimum: 22](#_Toc103681485)

[Tree Maximum: 22](#_Toc103681486)

[Tree Successor: 22](#_Toc103681487)

[Tree Predecessor: 23](#_Toc103681488)

[Tree Insert: 23](#_Toc103681489)

[Transplant: 23](#_Toc103681490)

[Tree Delete: 24](#_Toc103681491)

[Alberi Red and Black: 24](#_Toc103681492)

[Left Rotate: 24](#_Toc103681493)

[RB Insert: 25](#_Toc103681494)

[RB Transplant: 25](#_Toc103681495)

[RB Delete: 26](#_Toc103681496)

[RB Delete Fixup: 27](#_Toc103681497)

# Algoritmi e complessità

## Algoritmi di sort:

### Insertion Sort:

Insertion Sort(**A**)

**for** j ←2 **to** A.length

key ←A[j]

i ← j – 1

**While** i > 0 AND A[i] > key

A[i+1] ← A[i]

i ← i – 1

A[i + 1] ← key

Complessità nel caso peggiore:

### Bubble Sort:

BubbleSort(**A, n**)

**for** i ← 1 **to** n – i – 1

**for** j ← 1 **to** n – i – 2

**if** A[j] > A[j + 1]

**Scambia** A[j] ↔ A[j + 1]

Complessità nel caso peggiore:

### Merge Sort:

Merge Sort(**A, p, r**)

**if** p < r

q ← (p + r)/2

Merge Sort(**A, p, q**)

Merge Sort(**A, q+1, r**)

Merge(**A, p, q, r**)

Merge(**A, p, q, r**)

n1 ← q – p + 1

n2 ←r – q

L[1…n1 + 1], R[1…n2­­ + 1]

**for** i ← 1 **to** n1

L[i] ← A[i + p – 1]

**for** j ← 1 **to** n2

R[j] ← A[j – q]

L[n1 + 1] ←

R[n2 + 1] ←

i ← j ← 1

**for** k ← p **to** r – 1

**if** L[i] <= R[j]

A[k] ← L[i]

i ← i + 1

**else**

A[k] ← R[j]

j ← j + 1

Complessità nel caso peggiore:

### Quick Sort:

Quick Sort(**A, p, r**)

**if** p < r

q ← Partition(A, p, r)

Quick Sort(A, p, q – 1)

Quick Sort(A, q + 1, r)

Partition(**A, p, r**)

x ← A[r]

i ← p – 1

**for** j ← p **to** r – 1

**if** A[j] <= x

i ← i + 1

**Scambia** A[i] ↔ A[j]

**Scambia** A[i + 1] ↔ A[r]

Return i + 1

Randomized Quick Sort(**A, p, r**)

**if** p < r

q ← Randomized Partition(A, p, r)

Randomized Quick Sort(A, p, q – 1)

Randomized Quick Sort(A, q + 1, r)

Randomized Partition(**A, p, r**)

i ← Random(p, r)

**Scambia** A[i] ↔ A[r]

Partition(A, p, r)

Complessità di tempo nel caso peggiore:

### Heap Sort

Max Heapify(**A, i**)

l ← Left(i)

r ← Right(i)

**if** l <= A.HeapSize **AND** A[l] > A[i]

Max ← l

**else** Max ← i

**if** r <= A.HeapSize **AND** A[r] > A[Max]

Max ← r

**if** Max ≠ i

**Scambia** A[i] ↔ A[Max]

Max Heapify(A, Max)

Build Max Heap(**A**)

A.HeapSize ← A.length

**for** i ← A.length/2 **down to** 1

Max Heapify(A, i)

Heap Sort(**A**)

Build Max Heap(A)

**for** i ← A.HeapSize **down to** 2

**Scambia** A[i] ↔ A[1]

A.HeapSize ← A.HeapSize – 1

Max Heapify(A, 1)

Complessità nel caso peggiore:

Heap Maximum(**A)**

**Return** A[1]

Heap extract-Max(**A**)

**if** A.HeapSize < 1

**error:** *“Underflow dell’Heap”*

max ← A[1]

A[1] ← A.HeapSize

A.Heapsize ← A.HeapSize – 1

Max Heapify(A,1)

**Return** max

Heap Increase Key(**A, i, key**)

**if** key < A[i]

**error:** *“Invalid Data”*

A[i] ← key

**While** i > 1 **AND** A[parent(i)] < A[i]

**Scambia** A[i] ↔ A[parent(i)]

i ← parent(i)

Complessità nel caso peggiore:

### Bucket/Bin Sort:

Bucket Sort(**A**)

B[0…n – 1]

n ← A.length

**for** i ← 0 **to** n – 1

B[i] ← **new list**

**for** i ← 1 **to** n

Push(B[i], A[i])

**for** j ← 0 **to** n – 1

Insertion Sort(B[i])

Concatena le liste B[0] – B[n-1] in ordine

Complessità nel caso peggiore:

### Counting Sort:

CountingSort(**A**)

n ← A.length

C[0...n]

**for** i ← 0 **to** n

C[i] ← 0

**for** j ←1 **to** n

C[A[j]] ← C[A[j]] +1

**for** i ← 1 **to** n

C[i] ← C[i] + C[i - 1]

**for** j ← n **down to** 1

B[C[A[j]]] ← A[j]

C[A[j]] ← C[A[j]] – 1

Complessità nel caso peggiore:

Se il valore di allora la complessità sarà

### Radix Sort:

Radixsort(**A,d**)

**for** i ← 1 **to** d

n ← A.length

exp ←

*// Usa BubbleSort su sort A e specificare da input la i*

**for** j ← 1 **to** n

**for** k ← j + 1 **to** n

**if** (A[j] / exp) \* |10| > (A[k]/exp) \* |10|

**Scambia** A[i] ↔ A[j]

Complessità caso peggiore:

## Programmazione Dinamica:

### Fibonacci:

Fibonacci\_R(**i**) *//versione ricorsiva*

**if** i = 0 **OR** i = 1

**Return** 1

Numero ← Fibonacci\_R(i – 1) + Fibonacci\_E(i – 2)

**Return** Numero

Numero\_Fibonacci(**i**) *//versione Programmazione Dinamica*

**for** j ← 1 **to** MAX

KnFibonacci[j] ← 0

Fibonacci(i)

Fibonacci(**i**)

**if** KnFibonacci(i) ≠ 0

**Return** KnFibonacci(i)

**if** i = 0 **OR** i = 1

**Return** 1

KnFibonacci[i] ← Fibonacci(i – 1) + Fibonacci(i – 2)

**Return** KnFibonacci[i]

Complessità nel caso peggiore:

### LCS:

LCS(**x, y**)

n ← x.length

m ← y.length

c[0…n, 0…m]

**for** i ← 0 **to** n

c[i, 0] ← 0

**for** j ← 0 **to** m

c[0, j] ← 0

**for** i ← 1 **to** n

**for** j ← 1 **to** m

**if** x[i] = y[j]

c[i, j] ← c[i – 1, j – 1] + 1

**else**

c[i, j] ← **max**(c[i – 1, j], c[i, j – 1])

**Return** c[n, m]

Complessità nel caso peggiore:

### Distanza di Levenshtein:

Distanza Levenshtein(**x, y**)

n ← x.length

m ← y.length

d[0…n, 0…m]

**for** i ← 0 **to** n

d[i, 0] ← i

**for** j ← 0 **to** m

d[0, j] ← j

**for** i ← 1 **to** n

**for** j ← 1 **to** m

**if** x[i] = y[j]

dist\_sub ← 0

**else** dist\_sub ← 1

d[i, j] ← **min**(d[i – 1, j – 1) + dist\_sub, d[i, j – 1) + 1, d[i – 1, j) + 1)

Return d[n, m]

Complessita di tempo nel caso peggiore:

### Matrix Chain Order e Multiply:

Matrix Multiply(**A, B**)

**if** A.Columns B.Rows

**error:** *“Dimensioni non compatibili”*

**else** C[1…A.Rows, 1…B.Columns]

**for** i ← 1 **to** A.Rows

**for** j ← 1 B.Columns

C[i, j] ← 0

**for** k ← 1 **to** A.Columns

C[i, j] ← C[i, j] + A[i, k] + B[k, j]

**Return** C

Tempo: dato dai prodotti scalari **p\*q\*r**

Costo: **dipende alla parentesizzazione**

Matrix Chain Order(**p**)

n ← p.length

m[1…n, 1…n]

s[1…n – 1, 2…n]

**for** i ← 1 **to** n

m[i, i] ← 0

**for** l ← 2 **to** n

**for** i ← 1 **to** n – 1 + l

j ← i + l – 1

m[i, j] ←

**for** k ← i **to** j – 1

q ← m[i, k] + m[k+1, j] + p[i – 1]\*p[k]\*p[j]

**if** q m[i, j]

m[i, j] ← q

s[i, j] ← k

**Return** m, s

Complessità nel caso peggiore:

Recursive Matrix Chain Order(**P, i, j**)

**if** i = j

**Return** 0

m[i, j] ←

**for** k ← i **to** j – 1

q ← Recursive Matrix Chain Order(P, i, k) +

Recursive Matrix Chain Order(P, k + 1, j) +

P[i – 1]\*P[k]\*P[j]

**if** q < m[i, j]

m[i, j] ← q

**Return** m[i, j]

Complessità nel caso peggiore:

### Zaino 0-1:

Knapsack 0-1(**v,w,W**)

n ← v.length

V[0…n, 0…W]

**for** i ← 0 **to** v

V[i, 0] ← 0

**for** j ← 0 **to** W

V[0, j] ← 0

**for** i ← 1 **to** v

**for** j ← 1 **to** W

**if** j < w[i]

V[i, j] ← V[i – 1, j]

**else** V[i, j] ← **max**(V[i – 1,j], V[i – 1, j – w[i]] + v[i])

**Return** V[n,W]

Complessità nel caso peggiore:

### Fastest Way:

Fastest Way(a, e, x, n, t)

**for** j ← 2 to n

**if**

**else**

**if**

**else**

**if**

**else**

Complessità nel caso peggiore:

### Algoritmi con Memoization:

Memoized LCS(**x, y**)

n ← x.length

m ← y.length

c[0…n, 0…m]

**for** i ← 1 **to** n

**for** j ← 1 **to** m

c[i, j] ←

**Return** Lookup LCS(x,y,c,n,m)

Lookup LCS(**x, y, c, i, j**)

**if** c[i, j] >

**Return** c[i, j]

**else if** i = 0 OR j = 0

c[i, j] ← 0

**else**

**if** x[i] = y[j]

c[i, j] ← Lookup LCS(x, y, c, i – 1, j – 1) + 1

**else**

c[i, j] ← **max**(Lookup LCS(x, y, c, i – 1, j), Lookup LCS(x, y, c, i, j – 1))

**Return** c[i, j]

Complessità nel caso peggiore:

Memoized Levenshtein(**x, y**)

n ← x.length

m ← y.length

d[0…n, 0…m]

**for** i ← 1 **to** n

**for** j ← 1 **to** m

c[i, j] ←

**Return** Lookup Levenshtein(x,y,d,n,m)

Lookup Levenshtein(**x, y, d, i, j**)

**if** d[i, j] <

**Return** d[i, j]

**else if** i = 0

d[i, j] ← j

**else if** j = 0

d[i, j] ← i

**else**

**if** x[i] = y[j]

dist\_sub ← 0

**else** dist\_sub ← 1

d[i, j] ← **min**(Lookup Levenshtein(x, y, c, i – 1, j – 1) + dist\_sub ,

Lookup Levenshtein(x, y, c, i – 1, j) + 1,

Lookup Levenshtein(x, y, c, i, j – 1) + 1)

**Return** d[i, j]

Complessità nel caso peggiore:

Memoized Knapsack(**v, w, W**)

n ← v.length

V[0…n, 0…m]

**for** i ← 1 **to** n

**for** j ← 1 **to** m

V[i, j] ←

**Return** Lookup Knapsack(v, w, V, n, W)

Lookup Knapsack(**v, w, V, i, j**)

**if** V[i, j] <

**Return** V[i, j]

**else if** i = 0 OR j = 0

V[i, j] = 0

**else**

**if** j < w[i]

V[i, j] ← Lookup Knapsack(v, w, V, i – 1, j)

**else** V[i, j] ← **max**(Lookup Knapsack(v, w, V, i – 1, j),

Lookup Knapsack(v, w, V, i – 1, j – w[i]) + v[i])

**Return** V[i, j]

Complessità nel caso peggiore:

Memoized MCO(**p**)

n p.length

m[1…n,1…n]

**for** i 1 **to** n

**for** j 1 **to** n

m[i, j]

**Return** Lookup MCO(p, m , 1, n)

Lookup MCO(**p, m, i, j**)

**if** m[i, j]

**Return** m[i, j]

**else if**

m[i, j] 0

**else**

**for** k i **to** j – 1

q Lookup MCO(p, m, i, k) + Lookup MCO(p, m, k+1, j) + p[i – 1]\*p[k]\*p[j]

**if** q < m[i, j]

m[i, j] q

**Return** m[i, j]

Complessità nel caso peggiore:

## Algoritmi Greedy:

### Greedy Activity Selector:

Greedy Activity Selector(s, f)

n ← s.length

A ← A U {a1}

k ← 1

**for** j ← 2 **to** n

**if** s[j] >= f[k]

A ← A U {aj}

k ← j

**Return** A

Complessità nel caso peggiore:

### Fractional Knapsack:

Fractional Knapsack(**v, w, X, TOT**)

n ← v.length

**for** i ← 1 **to** n

X[i] ← 0.0

Cap ← TOT

val ← 0.0

i ← 1

**While** i <= n **AND** Cap > 0.0

**if** w[i] <= Cap

X[i] ← 1.0

val ← val + v[i]

Cap ← Cap – w[i]

**else**

X[i] ← Cap/w[i]

val ← val + v[i] \* X[i]

Cap ← 0.0

i ← i + 1

Complessità nel caso peggiore:

### Huffman:

Huffman(**A**)

n ← |A|

Q ← A

**for** i ← 1 **to** n – 1

z ← new\_nodo

x ← z.left ← EXTRACT\_MIN(Q)

y ← z.right ← EXTRACT\_MIN(Q)

f|z| ← f|x| + f|y|

INSERT(Q, z)

**Return** EXTRACT\_MIN(Q)

Complessità nel caso peggiore:

## Grafi:

### BFS:

BFS(**G, s**)

**for each** v G.V

v.d ←

|v| ← NIL

v.color ← WHITE

s.d ← 0

s.color ← GRAY

Q ← NIL

ENQUEUE(Q, s)

**While** Q

u ← DEQUEUE(Q)

**for each** v G.Adj[u]

**if** v.color ← WHITE

|v| ← u

v.d ← u.d + 1

v.color ← GRAY

ENQUEUE(Q, v)

u.color ← BLACK

Complessità nel caso peggiore:

Complessità nel caso peggiore:

### DFS:

DFS(**G**)

**for each** v G.V

|v| ← NIL

v.color ← WHITE

time ← 0

**for each** v G.V

**if** v.color = WHITE

DFS\_visit(G, v)

DFS\_visit(**G, v**)

v.d ← time ← time + 1

v.color ← GRAY

**for each** u G.Adj[v]

**if** u.color ← WHITE

|u| ← v

DFS\_visit(G, u)

v.f ← time ← time + 1

v.color ← BLACK

Complessità nel caso peggiore:

Complessità nel caso peggiore:

DFS(**G**)

L ← new stack

**for each** v G.V

|v| ← NIL

v.color ← WHITE

time ← 0

**for each** v G.V

**if** v.color = WHITE

DFS\_visit(G, v, L)

Crea GT

**While** L

u ← L.pop()

**if** u.color = WHITE

DFS\_visit\_1(GT, u)

SCC ← predecessor subgraph

DFS\_visit(**G, v, L**)

v.d ← time ← time + 1

v.color ← GRAY

**for each** u G.Adj[v]

**if** u.color ← WHITE

|u| ← v

DFS\_visit(G, u)

v.f ← time ← time + 1

v.color ← BLACK

Push(L, v)

Complessità nel caso peggiore:

Complessità nel caso peggiore:

### Kruskal:

MakeSet(**u**)

u.rank ← 0

← u

FindSet(**u**)

**if**

← FindSet()

Return |u|

Union(**u, v**)

Link(FindSet(u), FindSet(v))

Link(u, v)

**if** u.rank > v.rank

|v| ← u

**else**

|u| ← v

**if** v.rank = u.rank

v.rank ← v.rank + 1

Kruscal(**G, w**)

A ← NIL

**for each** v G.V

MakeSet(v)

Sort(w, 1, |G.E|)

**for each** (u, v) G.E

**if** FindSet(u) FindSet(v)

A ← A U {u, v}

Union(u, v)

**Return** A

Complessità nel caso peggiore:

### Primm:

Primm(G, v, w)

**for each** v G.V

v.d ←

|v| ← NIL

s.d ← 0

Q ← G.V

**While** Q

u ← EXTRACTMIN(Q)

**for each** v G.V

**if** v Q **AND** v.d > w(u, v)

v.d ← w(u, v)

|v| ← u

Complessità nel caso peggiore:

## Hashing

### Hash Insert:

Hash\_Insert(**T, k**)

La procedura prende come input una Tabella Hash T ed una chiave k;

Restituisce j se la cella j contiene la chiave k oppure NIL se la chiave k non si trova nella tabella T

i ← 0

**repeat**

j ← h(k, i)

**if** T[j] = **NIL** then

T[j] ← k

**Return** j

else i ← i + 1

**until** i = m

**error** *“Overflow Tabella Hash”*

Caso peggiore:

### Hash Search:

Hash Search(**T, k**)

i ← 0

**repeat**

j ← h(k, i)

**if** T[j] = k then

Return j

i ← i + 1

**until** T[j] = NIL or i = m

**Return NIL**

Caso peggiore:

### Hash Delete:

Hash Delete(**T, k**)

i ← 0

**repeat**

j ← h(k, i)

**if** T[j] = k then

temp ← T[j]

T[j] ← DELETE

Return temp

i ← i + 1

**until** T[j] = NIL or i = m

**Return NIL**

Caso peggiore:

## Grafi cammini minimi

### Bellman-Ford:

Init single source(**G, s**)

**for each** v G.V

|v| ← NIL

v.d ←

s.d ← 0

Relax(**u, v, w**)

**if** v.d > u.d + w(u, v)

v.d ← u.d + w(u, v)

|v| ← u

Bellman-Ford (**G, s, w**)

Init single source(G,s)

**for** i ← 1 **to** |G.V| – 1

**for each** {u, v} G.E

Relax(u, v, w)

**for each** {u, v} G.E

**if** v.d > u.d + w(u, v)

**Return** FALSE

**Return** TRUE

Caso peggiore:

Caso migliore:

### Dijkstra:

Init Single Source(**G,s**)

**for each** v G.V

|v| ← NIL

v.d ← inf

s.d ← 0

Relax(**u,v,w**)

**if** v.d > u.d + w(u, v)

v.d ← u.d + w(u, v)

|v| ← u

Dijkstra(**G,s,w**)

Init single source(G,s)

A ← NIL

Q ← G.V

**While** Q NIL

u ← Extractmin(Q)

A ← A U {u}

**for each** v G.Adj[u]

Relax(u, v, w)

Caso peggiore:

## Algoritmi visita e ricerca albero binario

### Tree Search:

TreeSearch(**x, k**)

**if** x = NIL **OR** x.key = k

**Return** x

**if** x.key > k

**Return** TreeSearch(x.left, k)

**else Return** TreeSearch(x.right, k)

Complessità :

### Tree Iterative Search:

Tree-Iterative-Search(**x, k**)

**While** xNIL **AND** k Key[x]

if k < key[x]

x ← Left[x]

else x ← Right[x]

**Return** x

Complessità:

### Tree Minimum:

Tree-Minimum(**x**)

**While** Left[x] NIL

x ← Left[x]

**Return** x

Complessità:

### Tree Maximum:

Tree-Maximum(**x**)

**While** Right[x] NIL

x ← Right[x]

**Return** x

Complessità:

### Tree Successor:

Tree-Successor(**x**)

**if** Right[x] NIL

**Return** Tree-Minimum(Right[x])

y ←p[x]

**While** y NIL **AND** x = Right[y]

x ← y

y ← p[y]

**Return** y

Complessità:

### Tree Predecessor:

Tree-Predecessor(**x**)

**if** Left[x] NIL

**Return** Tree-Mamimum(Left[x])

y ←p[x]

**While** y NIL **AND** x = Left[y]

x ← y

y ← p[y]

**Return** y

Complessità:

### Tree Insert:

Tree-Insert(**T, z**)

y←NIL

x← root[T]

**While** x NIL

y←x

**if** key[z] < key[x]

x←Left[x]

**else**

x←Right[x]

p[z]←y

**if** y = NIL

root[T]←z

**else** if key[z] < key[y]

Left[y] ← z

**else** Right[y] ← z

Complessità:

### Transplant:

TRANSPLANT(**T,u,v**)

**if** p[u] = NIL

root[T] ← v

**else if** u = left[p[u]]

left[p[u]] ← v

**else** right[p[u]] ← v

**if** v NIL

p[v] ← p[u]

Complessità:

### Tree Delete:

Tree-Delete(T,z)

**if** Left[z] = NIL

TRANSPLANT(T, z, Right[z])

**else if** Right[z] = NIL

TRANSPLANT(T,z, Left[z])

**else** y ← Tree-Minimum(Right[z])

**if** p[y] z

TRANSPLANT(T,y, Right[y])

Right[y] ← Right[z]

p[Right[y]] ← y

TRANSPLANT(T, z, y)

Left[y] ← Left[z]

p[Left[y]] ← y

### Tree Inorder Visit:

Tree inorder visit(**x**)

**if** x NIL

Tree inorder visit(x.left)

Print(x.key)

Tree inorder visit(x.right)

### Tree Preorder Visit:

Tree preorder visit(**x**)

**if** x NIL

Print(x.key)

Tree preorder visit(x.left)

Tree preorder visit(x.right)

### Tree Postorder Visit:

Tree postorder visit(**x**)

**if** x NIL

Tree postorder visit(x.left)

Tree postorder visit(x.right)

Print(x.key)

## Alberi Red and Black:

### Left Rotate:

Left Rotate(**T, x**)

y ← Right[x]

Right[x] ← Left[y]

**if** Left[y] NIL

p[Left[y]] ← x

p[y] ← p[x]

**if** p[x] = NIL

root[T] ← y

**else if** x = Left[p[x]]

Left[p[x]] ← y

**else** Right[p[x]] ← y

Left[y] ← x

p[x] ← y

Complessità nel caso peggiore:

### RB Insert:

RB Insert(**T, x**)

x.color ← RED

**While** x root[T] AND p[x].color = RED

**if** p[x] = Left[p[p[x]]] *//Then*

y ← Right[p[p[x]]]

**if** y.color = RED

p[x].color ← BLACK

y.color ← BLACK

p[p[x]].color ← RED

x ← p[p[x]]

**else if** x = Right[p[x]]

x ← p[x]

Left Rotate(T, x)

p[x].color ← BLACK

p[p[x]].color ← RED

Right Rotate(T, p[p[x]])

**else** *//Analogo al ramo then con left e right invertiti*

y ← Left[p[p[x]]]

if y.color = RED

p[x].color ← BLACK

y.color ← BLACK

p[p[x]].color ← RED

x ← p[p[x]]

else if x = Left[p[x]]

x ← p[x]

Right Rotate(T, x)

p[x].color ← BLACK

p[p[x]].color ← RED

Left Rotate(T, p[p[x]])

root[T].color ← BLACK

Complessità nel caso peggiore:

### RB Transplant:

RB Transplant(T, u, v)

**if** p[u] = NIL[T]

root[T] ← v

**else if** u = Left[p[u]]

Left[p[u]] ← v

**else** Right[p[u]] ← v

p[v] ← p[u]

Complessità nel caso peggiore:

### RB Delete:

RB Delete(**T, z**)

y ← z

y\_orig\_color ← y.color

**if** Left[z] = NIL[T]

x ← Right[z]

RB Transplant(T, z, Right[z])

**else if** Right[z] = NIL[T]

x ← Left[z]

RB Transplant(T, z, Left[z])

**else**

y ← Tree Minimum(Right[z])

y\_orig\_color ← y.color

x ← Right[y]

**if** p[y] = z

p[x] ← y *//Se x è = NIL[T]*

**else**

RB Transplant(T, y, Right[y])

Right[y] ← Right[z]

p[Right[y]] ← y

Transplant(T, z, y)

Left[y] ← Left[z]

p[Left[y]] ← y

y.color ← z.color

**if** y\_orig\_color = BLACK

RB Delete Fixup(T, x)

Complessità nel caso peggiore:

### RB Delete Fixup:

RB Delete Fixup(**T, x**)

**While** x root[T] AND x.color = BLACK

**if** x = Left[p[x]]

w ← Right[p[x]]

**if** w.color = RED

w.color ← BLACK

p[x].color ← RED

Left Rotate(T, p[x])

w ← Right[p[x]]

**if** Left[w].color = BLACK **AND** Right[w].color = BLACK

w.color ← RED

x ← p[x]

**else if** Right[w].color = BLACK

Left[w].color ← BLACK

w.color ← RED

Right Rotate(T, w)

w ← Right[p[x]]

w.color ← p[x].color

p[x].color ← BLACK

Right[w].color ← BLACK

Left Rotate(T, p[x])

x ← root[T]

else *//analogo al ramo then con right e left scambiati*

x.color ← BLACK

Complessità nel caso peggiore: