DI PALMA MATTIA 0124002448.

TRACCIA 1.

QUESITO 1.

DESCRIZIONE DEL PROBLEMA:

Il problema posto nel quesito 1 riguarda la realizzazione di una struttura dati per insiemi generici basata su alberi **RED BLACK.**

La struttura dati deve poter effettuare le operazioni di **UNIONE, INTERSEZIONE e DIFFERENZA** trattando ogni albero come un insieme di numeri interi.

I numeri vengono letti da un file di testo in cui gli elementi di uno stesso insieme si trovano sulla stessa riga separati da uno spazio (righe diverse corrispondono a diversi insiemi).
Successivamente i numeri letti vengono inseriti come chiavi dei nodi dell'albero RED BLACK, tutto questo verrà spiegato più dettagliatamente nella sezione "DESCRIZIONE STRUTTURA DATI" e nella sezione "DESCRIZIONE ALGORITMO".

L'output del problema consiste nel mostrare i risultati delle suddette operazioni dotando il programma di un menu da cui l'utente può scegliere tramite tastiera la funzione da svolgere.

DESCRIZIONE STRUTTURA DATI:

La struttura dati trattata è quella degli alberi **RED BLACK**, ovvero un **albero binario di ricerca self-balancing** la cui altezza, grazie a particolari condizioni rimane limitata permettendo una complessità logaritmica nei metodi di inserimento ed eliminazione (metodi analizzati in seguito) tramite un bilanciamento "automatico" della struttura, evitando quindi la degradazione al caso peggiore ovvero la trasformazione in una lista.
Di seguito troviamo le caratteristiche che un albero RED BLACK deve soddisfare.

1. Ogni nodo di un albero RB contiene:
    - La chiave (in questo caso numero intero).
    - Puntatore al padre.
    - Puntatore al figlio destro.
    - Puntatore al figlio sinistro.
    - Colore del nodo (ROSSO/**NERO**).

2. Se **X** è la radice di un albero RB e Y rappresenta il sottoalbero sinistro allora **Key[X] >=key[Y]**
3. Se **X** è la radice di un albero RB e *Y rappresenta il sottoalbero sinistro allora **Key[X]<key[*Y]**
4. Ogni nodo è ROSSO o **NERO**.
5. Le chiavi sono mantenute solo nei nodi interni.

6. Le foglie sono NULL e sono sempre di colore **NERO**.
7. La radice dell'albero è sempre **NERA**
8. Se un nodo è ROSSO allora i suoi figli sono **NERI**.
9. Ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi **NERI**(B-ALTEZZA).
10. Ogni albero RB con radice x ha almeno $2^{bh(x)} - 1$ nodi interni, dove bh(x) è l'altezza nera del nodo x.
11. L'altezza nera bh(x) è il numero di nodi **NERI** in un cammino semplice dal nodo x ad una foglia (escluso x).
12. In un albero RB almeno la metà dei nodi dalla radice ad una foglia deve essere **NERA**, possiamo anche avere il caso in cui tutti nodi i nodi sono **NERI**.
13. In un albero RB nessun percorso da un nodo *v* ad una foglia è lungo più del doppio del percorso da *v* ad un'altra foglia.
14. In un albero RB con N nodi interni abbiamo un'altezza massima **2 ($\log_2$(N+1)).**

DIMOSTRAZIONE PUNTO 8.

Dimostrazione per induzione:

Caso base -> bh(x) = 0 quindi x è una foglia ed il sottoalbero $2^{bh(x)} - 1 = 1-1 = 0$ nodi interni.

Passo induttivo -> nodo x con due figli.
Figlio rosso = bh(x) e figlio nero = bh(x) - 1.

Ogni figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni quindi $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 =$ **$2^{bh(x)} - 1$ nodi interni.**

DIMOSTRAZIONE PUNTO 12.

Dimostrazione:

Sia h = altezza per la proprietà **6** (se un nodo è ROSSO non può avere figli ROSSI) e per la proprietà **8** sappiamo che almeno la metà dei nodi interni devono essere neri quindi la B-ALTEZZA deve essere uguale nel caso peggiore **h/2.**

$N >= 2^{h/2} - 1$ N -> numero nodi.

$N + 1 = 2^{h/2}$

$\log_2 (N + 1) >= h/2$

$2\log_2 (N + 1) >= h$

$2\log_2 (N + 1)$ -> ALTEZZA MASSIMA DELL'ALBERO RB CON 'N' NODI INTERNI.

DATI INPUT/OUTPUT:

L'input del programma come descritto nella traccia è rappresentato da un file di testo in cui gli elementi (in questo caso numeri interi) di uno stesso insieme si trovano sulla stessa riga separati da uno spazio (righe successive corrispondono a diversi insiemi).
La lettura avviene tramite il metodo **LETTURA_FILE** che salva l'intero letto in una variabile passata come parametro al metodo invocato per l'inserimento nell'albero RB, ovvero **INSERIMENTO_RB.**
L'output del programma consiste nella risoluzione delle operazioni di **UNIONE**, **INTERSEZIONE** e **DIFFERENZA** eseguite sugli alberi RB in cui ogni albero rappresenta un insieme.
Tutte le funzioni citate verranno analizzate nel dettaglio nella sezione "DESCRIZIONE ALGORITMO".

DESCRIZIONE ALGORITMO:

Il programma è basato sulla realizzazione di una struttura dati per la gestione di insiemi generici trattati come alberi **RED BLACK** precedentemente analizzati.
Utilizziamo il metodo **LETTURA_FILE** per salvare l'intero da passare come parametro al metodo **INSERIMENTO_RB** per la creazione del nodo da inserire nell'albero RB.
Sia per il metodo **INSERIMENTO_RB** che per il metodo **DELETE_RB** utilizziamo delle funzioni per evitare lo sbilanciamento dell'albero e conservare una complessità di tipo logaritmica.
Per INSERIMENTO_RB utilizziamo le funzioni **LEFT_ROTATE** e/o **RIGHT_ROTATE** mentre per DELETE_RB utilizziamo la funzione **DELETE_FIXUP** che a sua volta richiama LEFT_ROTATE e/o RIGHT_ROTATE**.**
Le operazioni insiemistiche vengono implementate tramite i metodi **UNIONE**, **INTERSEZIONE** e **DIFFERENZA** basate sulle funzioni INSERIMENTO_RB e DELETE_RB.
Il programma inoltre presenta una funzione **ELIMINA_DUPLICATI** per l'eliminazione di eventuali nodi con chiavi ripetute utilizzando i metodi **DELETE_RB** e **RICERCA_NODO.**
L'utente infine potrà scegliere tramite la tastiera se eseguire un'unica operazione insiemistica oppure eseguirle tutte.

PRESENTAZIONE PSEUDO_CODICE:

1. **LEFT_ROTATE (RB, x)**
2. y ← Right[x]
3. Right[x] ← left[y]
4. If left[y] != NIL
5. Padre[left[y]] ← x
6. Padre[y] ← Padre[x]
7. If Padre[x] = NIL
8. Root [RB] ← y
9. Else if x = left[padre[x]]
10. Left[padre[x]] ← y
11. Else
12. Right[padre[x]] ← y
13. Left[y] ← x
14. Padre[x] ←y

15. **RIGHT_ROTATE (RB, x)**
16. y ← left[x]
17. left[x] ← right[y]
18. If (right[y] != NIL)
19. Padre[right[y]] ← x
20. Padre[y] ← Padre[x]
21. If (Padre[x] = NIL)
22. Root [RB] ← y
23. Else if x = left[padre[x]]
24. Left[padre[x]] ← y
25. Else
26. Right[padre[x]] ← y
27. right[y] ← x
28. Padre[x] ←y

29. **MINIMO (x)**
30. WHILE (left[x] != NIL) do
31. x ←left[x]

31.1 Return x

32. **INSERIMENTO (RB, x, z)**

```
33. y ← NILL
34. x ← root [RB]
35. WHILE (x != NIL) do
36.         y ← x
37.         IF (key[z] < key[x]
38.                 x ← left[x]
39.          ELSE
40.                 x ← right[x]
41. Padre[z] ← y
42. IF (y = NIL) THEN
43.        Root [RB] ← z
44. ELSE IF (key[z] < key[y]) THEN
45.          Left[y] ← z
46. ELSE
47.          Right[y] ← z
48. Color[x] ← RED
49. WHILE (x != Root AND color[padre[x]] = RED) do
50.        IF padre[x] = left[padre[padre[x]]] THEN
51.                 y ← right[padre[padre[x]]]
52.                 IF (colore[y] = RED )THEN
53.                         Color[padre[x]] ← BLACK
54.                         Color[padre[padre[x]]] ← RED
55.                         x ←padre[padre[x]]
56.                 ELSE IF (x = right[padre[x]] THEN
57.                                 x ← padre[x]
58.                                 LEFT_ROTATE (*RB, x)
59.                  Color[padre[x]] ← BLACK
60.                  Color[padre[padre[x]]] ← RED
61.                  RIGHT_ROTATE (*RB, padre[padre[x]])
62.       ELSE IF (padre[x] = right[padre[padre[x]]]) THEN
63.                  y ← left[padre[padre[x]]]
64.                  IF (color[y] = RED) THEN
65.                          Color[padre[x]] ← BLACK
66.                          Color[y] ← BLACK
67.                          Color[padre[padre[x]]] ← RED
68.                          x ← padre[padre[x]]
69.                   ELSE IF (x = left[padre[x]]) THEN
70.                          x ← padre[x]
71.                          RIGHT_ROTATE (*RB, x)
72.                 Color[padre[x]] ← BLACK
73.                 Color[padre[padre[x] ← RED
74.                 LEFT_ROTATE (*RB, padre[padre[x]])
75. End while
76. Color [root [RB]] ← BLACK
```

77. TRANSPLANT (RB, u, v)

```
78.  If (padre[u] = NILL) then
79.          Root [RB] = v
80. else if (u = left[padre[u]] then
81.          Left[padre[u]] = v
82. Else
83.          Right[padre[u]] = v
84. If (v != NIL) then
85.          Padre[v] = padre[u]
```

86. **DELETE_FIXUP (RB, x)**

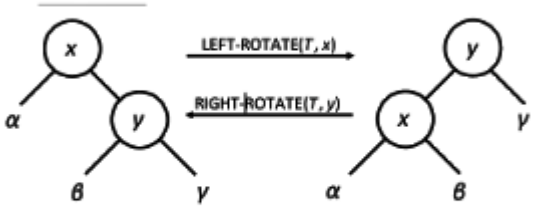
```
87. WHILE (x != Root [RB] AND color[x] = BLACK) DO
88.       IF (x = left[padre[x]]) THEN
89.            w ← right[padre[x]]
90.            IF (color[w] = RED) THEN
91.                 Color[w] ← BLACK
92.                 Color[padre[x]] ← RED
93.                 LEFT_ROTATE (RB, padre[x])
94.                 w ← right[padre[x]]
95.       IF (color[left[w]] = BLACK AND color[right[w]] = BLACK) THEN
96.              Color[w] ← RED
97.              x ← padre[x]
98.       ELSE IF (color[right[w]] = BLACK) THEN
99.              Color[left[w]] ← BLACK
100.             Color[w] ← RED
101.             RIGHT_ROTATE(RB,w)
102.             w ← right[padre[x]]
103.    Color[w] ← color[padre[x]
104.    Color[padre[x]] ← BLACK
105.    Color[right[w]] ← BLACK
106.    LEFT_ROTATE (RB, padre[x])
107.    x ← Root [RB]
108.    ELSE IF (x = right[padre[x]) THEN
109.           w ← left[padre[x]]
110.          IF (color[w] = RED) THEN
111.               Color[w] ← BLACK
112.               Color[padre[x]] ← RED
113.               RIGHT_ROTATE (RB, padre[x])
114.               w ←left[padre[x]]
115.          IF (color[left[w]] = BLACK AND color[right[w]] = BLACK) THEN
116.               Color[w] ← RED
```

```
117.            x ← padre[x]
118.        ELSE IF (color[left[w]] = BLACK)
119.                Color[right[w]] ← BLACK
120.                Color[w] ←RED
121.                LEFT_ROTATE (RB,w)
122.                w ←right[padre[x]]
123.        Color[w] ← color[padre[x]]
124.        Color[padre[x]] ← BLACK
125.        Color[left[w]] ←BLACK
126.        RIGHT_ROTATE (RB, padre[x])
127.        x ←Root [RB]
128.    ENDWHILE
129.    Color[x] ←BLACK

130.    RB_DELETE (RB, x)
131.    y ← x
132.    ycolore ← color[y]
133.    IF (left[x] = NIL) THEN
134.        z ← right[x]
135.        TRANSPLANT (RB, x, right[x])
136.    ELSE IF (right[z] = NIL)
137.         z ← left[x]
138.         TRANSPLANT (RB, x, left[x])
139.    ELSE
140.      y ← MINIMO (right[x])
141.      ycolore ← color[y]
142.      z ← right[y]
143.    IF (padre[y] = x) THEN
144.          Padre[x] ← y
145.    ENDIF
146.       ELSE
147.          TRANSPLANT (RB, y, right[y])
148.          Right[y] ← right[x]
149.          Padre[right[y]] ← y
150.          END ELSE
151.    TRANSPLANT (RB, x, y)
152.    Left[y] ← left[x]
153.    Padre[left[y]] ← y
154.    Color[y] ← color[x]
155.    IF (ycolore = BLACK)
156.        DELETE_FIXUP (RB,z)

157.UNIONE(RB1, RB2, A, B)
158.IF (A != NIL[RB1] AND B != NIL[RB2]) THEN
```

```
159.      INSERIMENTO (key[A])
160.      INSERIMENTO (key[B])
161.      UNIONE (RB1, RB2, left[A], left[B])
162.      UNIONE (RB1, RB2, right[A], right[B])
163.ELSE IF (A != NIL[RB1]) THEN
164.      INSERIMENTO (key[A])
165.      UNIONE (RB1, RB2, left[A], B)
166.      UNIONE (RB1, RB2, right[A], B)
167.ELSE IF (B != NIL[RB2]) THEN
168.      UNIONE (RB1, RB2, A, left[B])
169.      UNIONE (RB1, RB2, A, right[B])

170.RICERCA(RB, x ,k)
171.IF (x = NIL OR k = key[x]) THEN
172.    Return x
173.IF (k < key[x]) THEN
174.    Return RICERCA(left[x], k)
175.ELSE
176.    Return RICERCA(right[x], k)

177.INTERSEZIONE (RB1, RB2, A, B)
178.IF (A ! =NIL[RB1]) THEN
179.    IF (RICERCA (B, key[A]) != NIL[RB2]) THEN
180.            INSERIMENTO (key[A])
181.            INTERSEZIONE (RB1, RB2, left[A], B)
182.    ELSE
183.            INTERSEZIONE (RB1, RB2, left[A], B)
184.            INTERSEZIONE (RB1, RB2, right[A], B)

185.DIFFERENZA (RB, RB1, A, B)
186.IF (A != NIL) THEN
187.    K = RICERCA (RB, B, key[A])
188.    IF (k = NIL[RB]) THEN
189.            INSERIMENTO (D , key[A])
190              DIFFERENZA (RB, left[A], B, K)
191.            DIFFERENZA (RB, right[A], B, K)
192.    ELSE
193.            DIFFERENZA (RB, left[A], B, K)
194.            DIFFERENZA (RB, right[A], B, K)

196.ELIMINA_DUPLICATI (y)
197.IF (y != NIL)
198.    x ← y
199.    Chiave ← key[x]
```

```
200.    i ←0
201.    WHILE (x != NIL OR chiave = key[x])
202.            IF (chiave = key[x]) THEN
203.                    i ← i+1
204.                    IF (key[left[x]] = chiave OR key[right[x]] = chiave) THEN
205.                            IF (key[right[x]] != chiave) THEN
206.                                    i ←i+1
207.            IF (chiave < key[x]) THEN
208.                    x ←left[x]
209.            ELSE
210.                    x ←right[x]
211.    END WHILE
212.IF (i = 2) THEN
213.    DELETE_RB(y)
214.ELSE IF (i>2)
215.    WHILE (i > 2)
216.            n ← key[y]
217.            DELETE_RB(Y)
218.            k ← RICERCA (Root [RB], n)
219.            DELETE_RB(k)
220.            i ←i-1
221.    END WHILE
222.END IF
223.ELIMINA_DUPLICATI (RB, left[y])
224.ELIMINA_DUPLICATI (RB, right[y])
```

Dalla riga 1 alla riga 28 viene illustrato il comportamento delle due funzioni **LEFT_ROTATE** e **RIGHT_ROTATE,** le suddette vengono richiamate all'interno del metodo INSERIMENTO e nel metodo DELETE_FIXUP per evitare lo sbilanciamento dell'albero, conservando una complessità di tipo lineare senza degradare nel caso peggiore, ovvero la trasformazione in una lista che porterebbe ad una complessità per le due funzioni (INSERIMENTO e RB_DELETE) lineare rispetto al numero dei nodi.
Di seguito una rappresentazione grafica del funzionamento dei due metodi:



Dalla riga 29 alla 31 viene illustrata la funzione **MINIMO** che tramite un "while" scorre l'albero fino alla foglia sinistra restituendo quindi il nodo con chiave minore rispetto a tutti gli altri.

Dalla riga 32 alla 76 viene illustrato il metodo **INSERIMENTO.**
Dalla riga 32 alla 48 visitiamo l'albero tramite un "while" alla ricerca della posizione corretta per il nuovo nodo da inserire.
Confrontiamo la chiave del nuovo nodo con la chiave del nodo corrente, se key[z] (nodo da aggiungere) < key[x] allora analizziamo il sottoalbero sinistro altrimenti analizziamo il destro infine coloriamo il nodo corrente x di ROSSO.
Dalla riga 39 alla 75 entriamo in un "while" nel caso in cui x (nodo corrente) è diverso dalla radice dell'albero e suo padre è di colore ROSSO.
Entriamo nella sezione di codice dalla riga 50 alla 61 solo **se il padre del nodo x è uguale al figlio destro del nodo corrispondente a suo "nonno".**
Al suo interno distinguiamo **tre diversi casi** di inserimento per far sì che l'albero non violi le proprietà precedentemente elencate.

CASO 1:
Entriamo nella sezione di codice dalla riga 52 alla 55 solo **se il colore di y ovvero il figlio destro del "nonno" di x è ROSSO,** in questo caso il colore del padre di x diventa NERO, il colore di y diventa NERO, il colore del "nonno" di x diventa ROSSO e al nodo x viene assegnato il nodo corrispondente a suo "nonno".

CASO 2:
Entriamo nella sezione di codice dalla riga 56 alla 58 solo **se x è uguale al figlio destro di suo padre,** in questo caso al nodo x viene associato il nodo corrispondente a suo padre e infine viene richiamata la funzione **LEFT_ROTATE(x)** precedentemente descritta.

CASO 3:
Dalla riga 59 alla 61 coloriamo il padre di x a NERO, il "nonno" di x a ROSSO ed infine effettuiamo una **RIGHT_ROTATE (nonno[x]).**

Entriamo nella sezione di codice dalla riga 62 alla 75 solo **se il padre del nodo x è uguale al figlio destro del nodo corrispondente al "nonno".**
In questa sezione di codice distinguiamo **tre casi:**

CASO 1:
Entriamo nella sezione di codice dalla riga 64 alla 68 solo **se il colore di y ovvero il figlio sinistro del "nonno" di x è ROSSO,** in questo caso il colore del padre di x diventa NERO, il colore di y diventa NERO, il colore del "nonno" di x diventa ROSSO e al nodo x viene assegnato il nodo corrispondente a suo "nonno".

CASO 2:
Entriamo nella sezione di codice dalla riga 69 alla 71 solo **se x è uguale al figlio sinistro di suo padre,** in questo caso al nodo x viene associato il nodo corrispondente a suo padre e infine viene richiamata la funzione **RIGHT_ROTATE(x)** precedentemente descritta.

CASO 3:
Dalla riga 72 alla 74 coloriamo il padre di x a NERO, il "nonno" di x a ROSSO ed infine effettuiamo una **LEFT_ROTATE (nonno[x]).**

Nell'ultima riga ovvero la 75 coloriamo la radice dell'albero a NERO.

Dalla riga 77 alla 85 viene illustrata la funzione **TRANSPLANT** che viene utilizzata per sostituire un sottoalbero con un altro sottoalbero.

Dalla riga 130 alla 156 viene illustrata la funzione **RB_DELETE.**
Distinguiamo tre diversi casi per l'eliminazione del nodo dall'albero.

CASO 1:
Dalla riga 131 a 132 utilizziamo un nodo di appoggio y assegnandogli il nodo x da eliminare e ci salviamo il suo colore in una variabile ycolore.
Entriamo nella sezione di codice dalla riga 133 alla 135 solo **se il figlio sinistro del nodo x è uguale a NILL,** in questo caso assegniamo ad un nodo z il figlio destro di x e richiamiamo la funzione **TRANSPLANT (T, x, right[x])** precedentemente analizzata per "trapiantare" il sottoalbero del figlio destro di x in x stesso.

CASO 2:
Entriamo nella sezione di codice dalla riga 136 alla 138 solo **se il figlio destro del nodo x è uguale a NILL,** in questo caso assegniamo ad un nodo z il figlio sinistro di x e richiamiamo la funzione **TRANSPLANT (RB, x, left[x])** precedentemente analizzata.

CASO 3:
Dalla riga 139 alla 142 assegniamo ad y tramite la funzione **MINIMO** il nodo con chiave inferiore partendo dal figlio destro del nodo x.
Assegniamo alla variabile ycolore il colore di y e assegniamo al nodo z il figlio destro del nodo y.

Entriamo nella sezione di codice dalla riga 143 a 145 **se il padre di y è uguale al nodo z** assegnando al padre di x il nodo y.
In caso contrario accediamo alla sezione di codice dalla riga 147 alla 150 richiamando la funzione **TRANSPLANT (RB, y, right[y]).**
Assegniamo inoltre al figlio destro di y il figlio destro del nodo x e al padre del figlio destro di y il nodo y stesso.

CASO 4:
Dalla riga 151 alla 155 richiamiamo la **TRANSPLANT (RB, x, y)** e assegniamo al figlio sinistro di y il figlio destro di x.
Assegniamo inoltre al padre del figlio sinistro del nodo y il nodo y stesso e settiamo il colore di y al colore di x.

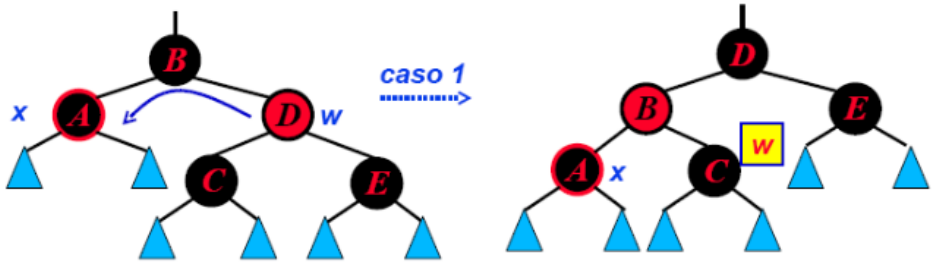
Entriamo nella sezione di codice dalla riga 155 a 156 solo se ycolore è NERO richiamando la funzione **DELETE_FIXUP** che andremo ora ad analizzare nel dettaglio.

Dalla riga 86 alla riga 129 analizziamo la funzione **DELETE_FIXUP** richiamata nella RB_DELETE solo nel caso in cui il colore di y (ovvero il colore del nodo x da eliminare) è NERO.
Questa condizione potrebbe creare delle violazioni alle proprietà dell'albero RB precedentemente descritte.
Se la DELETE_FIXUP viene richiamata **quindi y era NERO dopo la RB_DELETE y potrebbe essere la nuova radice ed essere ROSSA violando la proprietà 7.**
Rimuovendo y, z e il padre di z potrebbero esserci due nodi ROSSI di seguito violando la proprietà 8.
Se spostiamo y(NERO) qualunque cammino avrà un nero mancante violando la proprietà 9, Quindi bisogna aggiungere un NERO "extra" che sostituisce y.
Analizzando lo pseudocodice abbiamo quindi che:

Entriamo nella sezione di codice dalla riga 87 a 155 solo se x è diverso dalla radice dell'albero e il suo colore è NERO in caso affermativo cicliamo tramite il costrutto "while" finché una delle due condizioni non diventi falsa.
Dalla riga 88 a 89 se x è figlio sinistro di suo padre assegniamo al nodo w il figlio destro di suo padre e da qui distinguiamo **quattro casi:**

CASO 1:
Entriamo nella sezione di codice dalla riga 90 alla 94 **se il colore di w è rosso.**
Se w è rosso deve avere entrambi figli neri per non violare la proprietà 8.
Scambiamo quindi i colori di w e padre[x] per poi effettuare una rotazione sinistra di padre[x] e infine settare w al figlio destro del padre di x.

Di seguito un esempio grafico.



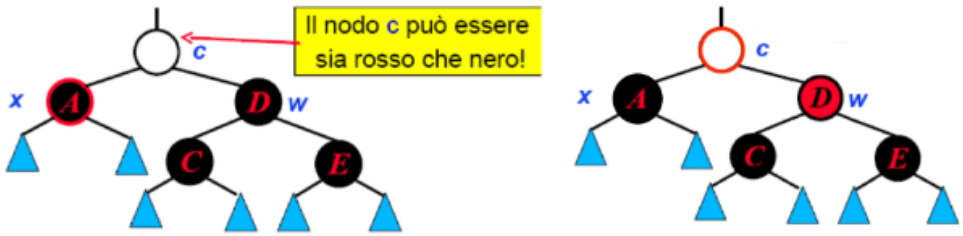
CASO 2:

Entriamo nella sezione di codice dalla riga 95 a 97 solo se il colore del figlio sinistro di w è NERO e il colore del figlio destro di w è NERO.

In questo caso il fratello di w di x è nero ed entrambi i suoi figli sono neri quindi togliamo un nero sia da w che da x e il rosso di w lo aggiungiamo al padre di x.

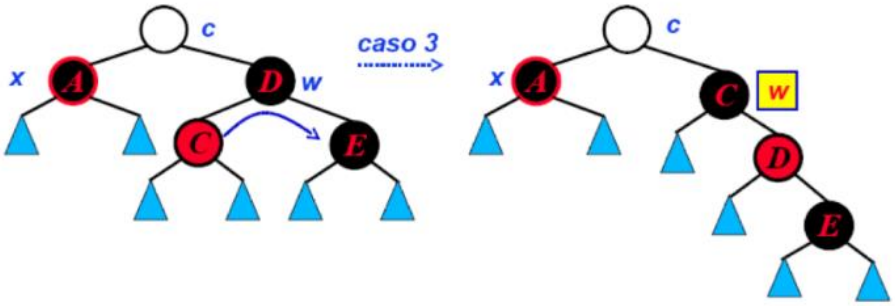
Se x è ROSSO-NERO il ciclo non viene ripetuto ed il nero “extra” viene rimosso.

Se x è NERO-NERO il ciclo viene ripetuto. Di seguito un esempio grafico

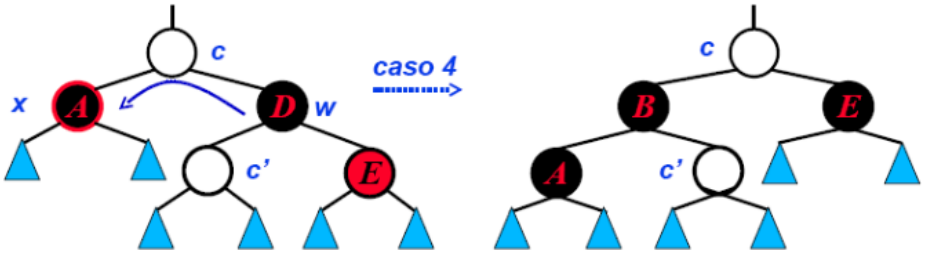


CASO 3:

Entriamo nella sezione di codice dalla riga 98 a 102 solo se il colore del figlio destro di w è NERO e il colore del figlio sinistro di w è rosso.
In questo caso settiamo il colore del figlio sinistro di w a NERO, il colore di w a ROSSO ed eseguiamo una RIGHT_ROTATE (RB, w).
Settiamo infine il nodo w al figlio destro del padre di x.
Di seguito un esempio grafico.



CASO 4:
Dalla riga 102 alla 107 ricadiamo nell'ultimo caso in cui il colore del figlio destro di w è ROSSO.
Settiamo quindi il colore di w al colore del padre di x, il colore del figlio destro di w a NERO ed effettuiamo una LEFT_ROTATE (RB, padre[x]) per eliminare il nero "extra".
Alla riga 107 effettuiamo l'assegnazione che fa terminare il ciclo ovvero x ← radice [RB]



Dalla riga 108 alla 128 abbiamo i quattro casi simmetrici in cui x non è figlio sinistro di suo padre ma figlio destro.
All'ultima riga ovvero la 129 abbiamo la colorazione del nodo x a NERO all'esterno del costrutto "while".

Dalla riga 157 a 169 viene illustrato il metodo **UNIONE (RB, RB1, A, B).**
La funzione viene sviluppata tramite la ricorsione visitando due alberi che in questo caso rappresentano due insiemi di numeri interi.
Entriamo nella sezione di codice dalla riga 158 a 162 solo se i nodi passati sono diversi dai nodi NIL dei rispettivi alberi di appartenenza.

In caso affermativo invochiamo il metodo **INSERIMENTO** a cui passiamo come parametro la prima volta la chiave del nodo A mentre la seconda passiamo come parametro la chiave del nodo B.
Successivamente richiamiamo il metodo **UNIONE** una volta sulla parte sinistra dell'albero e poi sulla parte destra.
Entriamo invece nella sezione di codice dalla riga 163 a 166 quando uno dei due nodi, ad esempio, B è uguale a NIL[RB1].
In questo caso invochiamo il metodo **INSERIMENTO** solo sulla chiave del nodo che risulta ancora diverso da NIL.
Successivamente invochiamo il metodo **UNIONE** sulla parte sinistra dell'albero in cui il nodo è ancora diverso da NIL in questo caso left[A] e poi sulla parte destra quindi right[A].
Entriamo nell'ultima parte di codice ovvero dalla riga 166 a 169 quando A è uguale a NIL[RB] e quindi in questo caso invochiamo **INSERIMENTO** solo sul key[B].
Successivamente richiamiamo il metodo **UNIONE** passandogli questa volta prima left[B] e poi right[B].

Dalla riga 177 a 184 viene illustrato il metodo **INTERSEZIONE (RB, RB1, A, B).**
Entriamo nella sezione di codice dalla riga 178 a 181 solo se il nodo in questo caso A è diverso da NIL[RB], in caso affermativo controlliamo tramite il costrutto "if" se la ricerca della chiave di A nell'albero RB1 da esito positivo per poter confermare l'inserimento della chiave di A nell'insieme risultato, dato che per la definizione di intersezione sappiamo che **RB ∩ RB1 è un insieme (albero) tale che {x ∈ RB e x ∈ RB1).**
Successivamente richiamiamo **INTERSEZIONE** passandogli come parametro il figlio sinistro del nodo A.
Entriamo invece nella sezione di codice da 181 a 184 quando A diventa uguale a NIL[RB], in questo caso richiamiamo **INTERSEZIONE** prima sul figlio sinistro di A e poi sul figlio destro.

Dalla riga 185 a 192 viene illustrato il metodo **DIFFERENZA (RB1, RB, A, B, K).**
Entriamo nella sezione di codice dalla riga 186 a 190 quando il nodo A è diverso dal nodo NIL, in questo caso salviamo nel nodo K il nodo risultato della ricerca sull'albero RB passandogli come parametro la chiave del nodo A.
Nel caso in cui la ricerca da esito negativo ovvero K è uguale a NIL[RB] allora inseriamo in un albero **D** la key[A] e richiamiamo la funzione **DIFFERENZA** prima passandogli il figlio sinistro di A e successivamente il destro.
Entriamo invece nella sezione di codice dalla riga 190 a 192 quando K è diverso da NIL, in questo caso effettuiamo la chiamata alla funzione **DIFFERENZA** prima passandogli il figlio sinistro di A e successivamente il destro. L'albero risultato **D** sarà l'insieme formato dagli elementi che appartengono a RB e non appartengono a RB1 dato che per definizione **RB – RB1 è un insieme tale che {x ∈ RB e x ∉ RB1}.**

Dalla riga 196 a 224 viene illustrato il metodo **ELIMINA_DUPLICATI (y).**
Dalla riga 197 a 200 se y è diverso da NIL assegniamo ad un nodo di appoggio x il nodo y, assegniamo alla variabile chiave la key[x] ed inizializziamo il contatore i ← 0.
Entriamo nella sezione di codice dalla riga 201 a 211 ciclando tramite un costrutto "while" Finché x è diverso da NIL oppure chiave è uguale a key[x].

Se la chiave è uguale a key[x] contiamo le occorrenze tramite il contatore i.
Continuiamo a ciclare dando al nodo x il valore del suo figlio sinistro se la chiave è minore di key[x] altrimenti gli assegniamo il valore del figlio destro fino all'uscita dal "while".
Dalla riga 212 a 213 controlliamo se i nodi con chiave ripetuta sono due tramite il costrutto "if" eliminandone soltanto uno tramite la funzione DELETE_RB(y) per evitare così la ripetizione del nodo nei risultati delle operazioni.
Nel caso in cui i nodi ripetuti sono più di due effettuiamo un "while" fin tanto che il contatore è maggiore di due.
All'interno del costrutto creiamo una variabile di appoggio n per il salvataggio di key[y] e richiamiamo la DELETE_RB(y).
Successivamente sempre all'interno del costrutto "while" creiamo un nodo di appoggio k in cui salviamo il nodo risultante dalla ricerca del nodo con chiave uguale ad n (key[y]).
Il nodo k verrà sicuramente trovato perché abbiamo un numero di occorrenze maggiore di due quindi procediamo con l'eliminazione di k tramite la funzione DELETE_RB(k).
Usciti dal while (righe 215-22) richiamiamo **ELIMINA_DUPLICATI** prima passandogli il figlio sinistro di y e successivamente quello destro.

ANALISI COMPLESSITA':

La struttura dati utilizzata dal programma è un **ALBERO BINARIO DI RICERCA SELF-BALANCING.**
Come analizzato in precedenza la struttura utilizza dei metodi come LEFT_ROTATE e RIGHT_ROTATE per evitare la degradazione al caso peggiore in cui l'albero si sbilanci diventando una lista.
Tutto questo fa sì che le operazioni di ricerca, eliminazione ed inserimento abbiano una complessità del tipo **O(log(n))** con n uguale al numero di elementi dell'albero.
Nel metodo **UNIONE** utilizziamo un algoritmo ricorsivo che si ripete fino a quando abbiamo ancora elementi nei due alberi su cui effettuare l'operazione di unione.
Dato che la complessità dell'algoritmo INSERIMENTO è O(log(n)) effettuiamo una sommatoria per il contatore **i** che va da 1 ad **n** (numero nodi) di **2log(n)** ovvero:

$\sum_{(i = 1 \text{ to } n)} 2\log(i) \rightarrow$ **O(n(log(n)) utilizzando Stirling. CASO PEGGIORE**

Nel metodo **INTERSEZIONE** utilizziamo ancora una volta un algoritmo ricorsivo che si ripete fino a quando l'albero non ha più elementi, effettuando quindi nel caso peggiore **n** (numero nodi) volte la ricerca**(O(log(n))** e l'inserimento**(O(log(n)).**

$\sum_{(i = 1 \text{ to } n)} \log(i) + \sum_{(i = 1 \text{ to } n)} \log(i) \rightarrow 2(n(\log(n))$ utilizzando Stirling**.**

→ **O(n(log(n)) CASO PEGGIORE.**

Il metodo **DIFFERENZA** viene sviluppato anch'esso come algoritmo ricorsivo che si ripete nel caso peggiore **n** (numero nodi) volte richiamando **2n** volte la funzione **RICERCA (O(log(n)))** e **n** volte la funzione **DELETE_RB(O(log(n))).**

$$\sum_{(i = 1 \text{ to } n)} 2\log(i) \quad + \quad \sum_{(i = 1 \text{ to } n)} \log(i) \quad \rightarrow \textbf{O(n(log(n)) CASO PEGGIORE.}$$

Il metodo **ELIMINA_DUPLICATI** è quello asintoticamente più "costoso", è sviluppato come un algoritmo che nel caso peggiore visita tutti i nodi dell'albero una volta tramite la ricorsione e successivamente utilizzando un costrutto "while" per contare il numero di duplicati.
Utilizziamo in seguito un secondo costrutto "while" che ripete **i** (numero duplicati) volte la **ricerca(O(log(n)))** e **l'eliminazione(O(log(n)))** del nodo.
Quindi il caso peggiore sarebbe quando **i (numero duplicati) = n (numero nodi)** portando quindi la complessità asintotica a **O (n^3 log(n))**

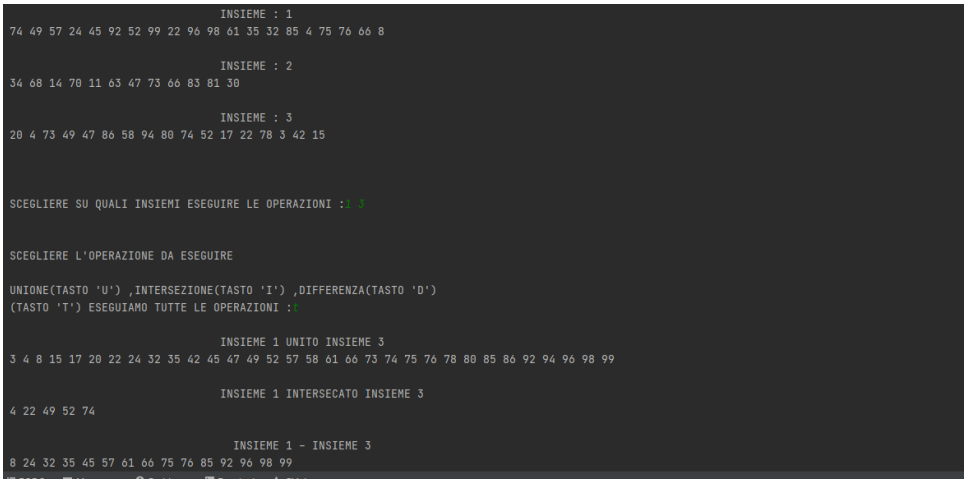
TEST E RISULTATI:

Invochiamo la funzione **LETTURA_FILE** per inserire passo per passo gli interi letti come chiavi dei nodi dell'albero su cui operare.
Dopo aver invocato la funzione il programma mostra a video gli insiemi su cui operare facendo scegliere all'utente tramite la tastiera l'operazione da eseguire.
Di seguito le possibili opzioni:

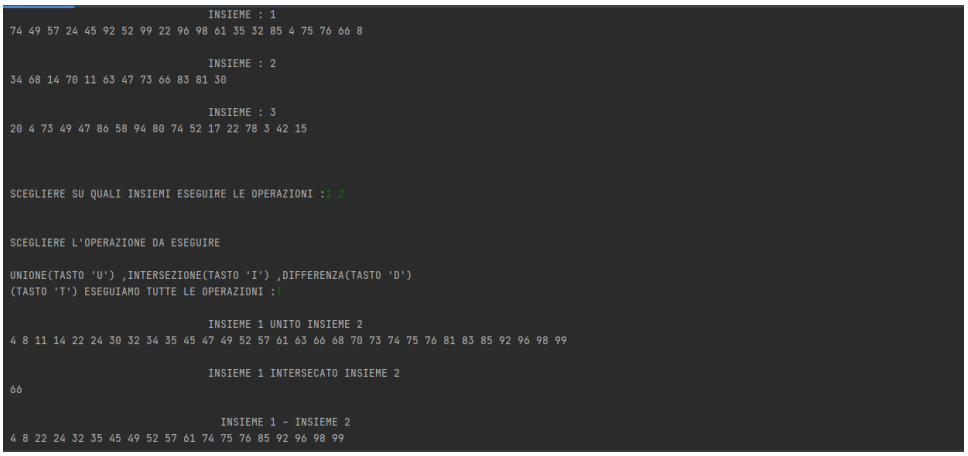
- U/u corrisponde alla funzione **UNIONE.**
- I/i corrisponde alla funzione **INTERSEZIONE.**
- D/d corrisponde alla funzione **DIFFERENZA.**
- T/t corrisponde all'esecuzione di tutte le operazioni **UNIONE, INTERSEZIONE E DIFFERENZA.**

Di seguito gli esempi sviluppati dal programma.

TEST 1:


INSIEME : 1
74 49 57 24 45 92 52 99 22 96 98 61 35 32 85 4 75 76 66 8
INSIEME : 2
34 68 14 70 11 63 47 73 66 83 81 30
INSIEME : 3
20 4 73 49 47 86 58 94 80 74 52 17 22 78 3 42 15
SCEGLIERE SU QUALI INSIEMI ESEGUIRE LE OPERAZIONI :1 3
SCEGLIERE L'OPERAZIONE DA ESEGUIRE
UNIONE(TASTO 'U') ,INTERSEZIONE(TASTO 'I') ,DIFFERENZA(TASTO 'D')
(TASTO 'T') ESEGUIAMO TUTTE LE OPERAZIONI :t
INSIEME 1 UNITO INSIEME 3
3 4 8 15 17 20 22 24 32 35 42 45 47 49 52 57 58 61 66 73 74 75 76 78 80 85 86 92 94 96 98 99
INSIEME 1 INTERSECATO INSIEME 3
4 22 49 52 74
INSIEME 1 - INSIEME 3
8 24 32 35 45 57 61 66 75 76 85 92 96 98 99

TEST 2:


INSIEME : 1
74 49 57 24 45 92 52 99 22 96 98 61 35 32 85 4 75 76 66 8
INSIEME : 2
34 68 14 70 11 63 47 73 66 83 81 30
INSIEME : 3
20 4 73 49 47 86 58 94 80 74 52 17 22 78 3 42 15
SCEGLIERE SU QUALI INSIEMI ESEGUIRE LE OPERAZIONI :1 2
SCEGLIERE L'OPERAZIONE DA ESEGUIRE
UNIONE(TASTO 'U') ,INTERSEZIONE(TASTO 'I') ,DIFFERENZA(TASTO 'D')
(TASTO 'T') ESEGUIAMO TUTTE LE OPERAZIONI :t
INSIEME 1 UNITO INSIEME 2
4 8 11 14 22 24 30 32 34 35 45 47 49 52 57 61 63 66 68 70 73 74 75 76 81 83 85 92 96 98 99
INSIEME 1 INTERSECATO INSIEME 2
66
INSIEME 1 - INSIEME 2
4 8 22 24 32 35 45 49 52 57 61 74 75 76 85 92 96 98 99

CLASS DIAGRAM: