ΔΠΜΣ Η/Α – Δομές Δεδομένων & Αλγόριθμοι

PROJECT 2

Red – Black Trees

ΚΑΡΑΓΙΑΝΝΗΣ ΓΕΩΡΓΙΟΣ
ΑΜ.: 2015508

P2.1 Ιδιότητες των Red – Black Trees

Τα red – black trees είναι ένα μοντέλο δυαδικών δένδρων αναζήτησης (binary search trees), όπου κάθε κόμβος τους, πέρα από τα πεδία *κλειδί*, *αριστερό_παιδί*, *δεξιό_παιδί* και *πατέρας*, περιέχει μία επιπλέον πληροφορία που τον περιγράφει· το *χρώμα*. Το χρώμα ενός κόμβου μπορεί να είναι είτε *μαύρο* είτε *κόκκινο*· εξ ου και το όνομα της δομής. Η τιμή χρώματος που θα έχουν οι κόμβοι ενός red – black tree, καθορίζεται με βάση τις ιδιότητες του δένδρου, οι οποίες θα πρέπει να ικανοποιούνται κάθε φορά:

1. Κάθε κόμβος είναι είτε *κόκκινος* είτε *μαύρος*
2. Ο ριζικός κόμβος είναι πάντα *μαύρος*
3. Κάθε καταληκτικός κόμβος (NIL) έχει *μαύρο χρώμα*
4. Ένας *κόκκινος* κόμβος έχει και τα δύο παιδιά του *μαύρα*
5. Όλες οι καθοδικές διαδρομές, από έναν κόμβο προς καταληκτικούς κόμβους, περιέχουν ισάριθμους *μαύρους* κόμβους

Οι παραπάνω ιδιότητες αποτελούν τους περιορισμούς που διέπουν το χρωματισμό των κόμβων σε ένα red – black tree. Η τήρησή τους εγγυάται ότι καμία διαδρομή, μεταξύ της ρίζας και ενός καταληκτικού κόμβου, δεν θα έχει μήκος μεγαλύτερο από το διπλάσιο, οποιασδήποτε άλλης διαδρομής. Η συνθήκη αυτή, κατά προσέγγιση, καθιστά τα red – black trees ισοσταθμισμένα.

Με βάση την τελευταία συνθήκη, προκύπτει ότι το ύψος οποιουδήποτε red – black tree με n εσωτερικούς κόμβους, δίνεται από τη σχέση $h = 2\log(n+1)$. Συνεπώς, ένα οποιοδήποτε red – black tree με n εσωτερικούς κόμβους, αποτελεί ένα δυαδικό δέντρο αναζήτησης με ύψος $O(\log n)$. Εύκολα λοιπόν, κανείς συμπεραίνει ότι σε δομές red – black trees, οι πράξεις δυναμικών συνόλων υλοποιούνται σε χρόνο $O(\log n)$.

P2.2 Περιστροφές (Rotations)

Πράξεις δυναμικών συνόλων, όπως οι *ΕΙΣΑΓΩΓΗ* και *ΔΙΑΓΡΑΦΗ*, μπορούν να τροποποιήσουν τη δομή ενός δέντρου, με αποτέλεσμα να χαθούν οι ιδιότητες των red – black trees. Για το λόγο αυτό, τα red – black trees διαθέτουν ειδικό μηχανισμό που αποκαθιστά τη δομή τους. Οι *περιστροφές* είναι μία τοπική πράξη (εκτελείται σε ένα δένδρο αναζήτησης), η οποία διατηρεί την ιδιότητα του δυαδικού δένδρου αναζήτησης (binary search tree). Τα red – black trees εκτελούν αριστερές και δεξιές περιστροφές μεταξύ ενός κόμβου και του δεξιού/αριστερού θυγατρικού του, αντίστοιχα. Όπως φαίνεται και στο παρακάτω

σχήμα, η εκτέλεση μίας δεξιάς περιστροφής καθιστά τον αριστερό θυγατρικό κόμβο x τη νέα ρίζα του υποδένδρου, μετατρέποντας τον πατρικό y σε *δεξιό_παιδί* του x. Η εκτέλεση μίας αριστερής περιστροφής είναι συμμετρική και επιφέρει τα αντίστροφα αποτελέσματα.

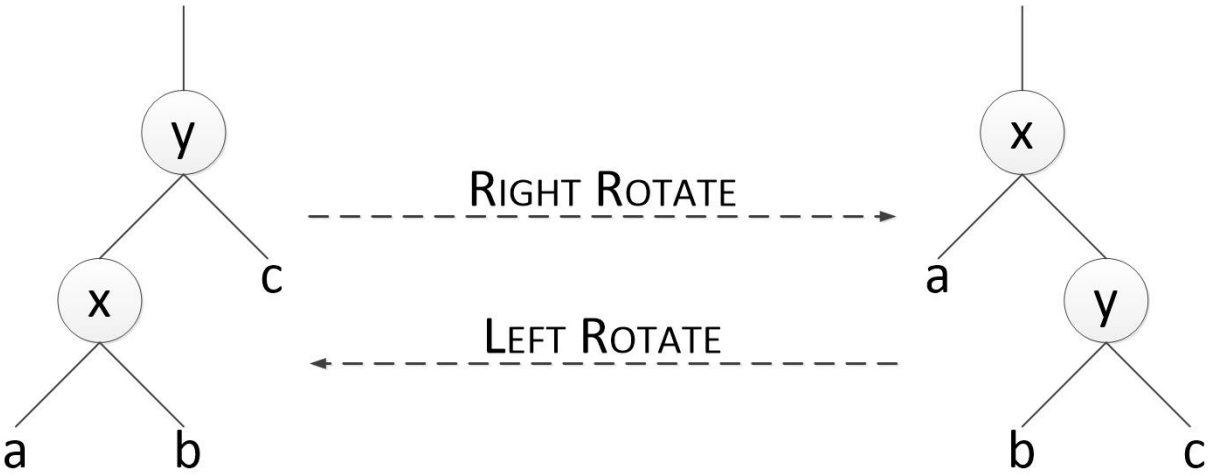

Figure 1 Οι πράξεις περιστροφής σε ένα δυαδικό δένδρο αναζήτησης

Βασική προϋπόθεση για την εκτέλεση των περιστροφών είναι ο πατρικός κόμβος της ρίζας (root) να είναι ο NIL, ενώ ο εκάστοτε θυγατρικός που συμμετέχει στην περιστροφή να μην είναι ο NIL. Παρακάτω παρουσιάζονται οι κώδικες της δεξιάς και της αριστερής περιστροφής.

```
1.  void right_rotation(){
2.
3.    // finding child participating in rotation
4.    z_aux = r_aux->left_child;
5.
6.    // converting z_aux's right sub-tree in r_aux's left sub-tree
7.    r_aux->left_child = z_aux->right_child;
8.    if (z_aux->right_child != NIL){
9.      z_aux->right_child->parent = r_aux;
10.   }
11.
12.   // connecting r_aux's parent with z_aux
13.   z_aux->parent = r_aux->parent;
14.   if (r_aux->parent == NIL){
15.     rbt_root = z_aux;
16.   }
17.   else{
18.     if (r_aux == r_aux->parent->right_child){
19.       r_aux->parent->right_child = z_aux;
20.     }
21.     else{
22.       r_aux->parent->left_child = z_aux;
23.     }
24.   }
25.
26.   // setting r_aux as z_aux's right child
27.   z_aux->right_child = r_aux;
28.   r_aux->parent = z_aux;
29. }
```

Code 1 Η συνάρτηση *right_rotation()* για την εκτέλεση δεξιάς περιστροφής

```
1. void left_rotation(){
2.
3.    // finding child participating in rotation
4.    z_aux = r_aux->right_child;
5.
6.    // converting z_aux's left sub-tree in r_aux's right sub-tree
7.    r_aux->right_child = z_aux->left_child;
8.    if (z_aux->left_child != NIL){
9.      z_aux->left_child->parent = r_aux;
10.   }
11.
12.   // connecting r_aux's parent with z_aux
13.   z_aux->parent = r_aux->parent;
14.
15.   if (r_aux->parent == NIL){
16.     rbt_root = z_aux;
17.   }
18.   else {
19.     if (r_aux == r_aux->parent->left_child){
20.       r_aux->parent->left_child = z_aux;
21.     }
22.     else{
23.       r_aux->parent->right_child = z_aux;
24.     }
25.   }
26.
27.   // setting r_aux as z_aux's right child
28.   z_aux->left_child = r_aux;
29.   r_aux->parent = z_aux;
30. }
```

Code 2 Η συνάρτηση *left_rotation()* για την εκτέλεση αριστερής περιστροφής

Κατά την εκτέλεση των συναρτήσεων *right_rotation()* και *left_rotation()*, μεταβάλλονται μόνο οι δείκτες *parent*, *right_child* και *left_child* των κόμβων, που συμμετέχουν στην περιστροφή. Τα υπόλοιπα πεδία τους, καθώς κι οι υπόλοιποι κόμβοι, παραμένουν αμετάβλητα. Η εκτέλεση μίας περιστροφής ολοκληρώνεται σε χρόνο O(1).

Οι μηχανισμοί των περιστροφών επιτρέπουν στα red – black trees να διατηρούνται ισοσταθμισμένα καθ' όλη τη διάρκεια, και κατά συνέπεια, ο χρόνος εκτέλεσης των πράξεων *ΕΙΣΑΓΩΓΗ* και *ΔΙΑΓΡΑΦΗ*, να παραμένει O(log n).

P2.3 Εισαγωγή (Insert)

Καθώς τα red – black trees είναι δυαδικά δένδρα αναζήτησης, εφοδιασμένα με κάποιες επιπλέον ιδιότητες, η *ΕΙΣΑΓΩΓΗ* δεν μπορεί να διαφέρει αρκετά από την αντίστοιχη πράξη σε ένα κοινό δυαδικό δένδρο αναζήτησης. Οι βασικές διαφορές αφορούν τη διατήρηση των παραπάνω ιδιοτήτων. Αρχικά, ο νέος κόμβος που εισάγεται στη δομή χρωματίζεται *κόκκινος*, ενώ οι δείκτες *αριστερό_παιδί* και *δεξιό_παιδί* δείχνουν στον κόμβο NIL. Επιπλέον, όπως έχει ήδη αναφερθεί, οι ιδιότητες ενός red – black tree μπορεί να χαθούν μετά την εκτέλεση μίας πράξης δυναμικού συνόλου. Για την αποκατάσταση των ιδιοτήτων αυτών, μόλις ολοκληρωθεί η *ΕΙΣΑΓΩΓΗ*, καλείται η βοηθητική συνάρτηση insert_fix_up(), που αναλαμβάνει αυτή τη διαδικασία.

```
1.  void insert(){
2.
3.    int key;
4.    node_ptr aux = NIL, f_aux = rbt_root;
5.    /*
6.    f_aux : f(orward)_aux pointing one of aux's childs
7.    */
8.
9.    printf("\n Give new node's key\t:\t");
10.   scanf("%d", &key);
11.   getchar();
12.
13.   // construction of new struct node
14.   x_aux = (node_ptr)malloc(sizeof(struct node));
15.   x_aux->key = key;
16.   x_aux->color = red; // new node is colored red
17.   x_aux->parent = NIL;
18.   x_aux->left_child = NIL; // new node's left_child points NIL
19.   x_aux->right_child = NIL; // new node's right_child points NIL
20.
21.   // searching new node's position in red - black tree
22.   while (f_aux != NIL){
23.     aux = f_aux;
24.     if (x_aux->key < f_aux->key){
25.       f_aux = f_aux->left_child;
26.     }
27.     else if (x_aux->key > f_aux->key){
28.       f_aux = f_aux->right_child;
29.     }
30.   }
31.
32.   // fixing pointers between new node and its parent
33.   x_aux->parent = aux;
34.   if (aux == NIL){
35.     rbt_root = x_aux;
36.   }
37.   else{
38.     if (x_aux->key < aux->key){
39.       aux->left_child = x_aux;
40.     }
41.     else if (x_aux->key > aux->key){
42.       aux->right_child = x_aux;
43.     }
44.   }
45.
46.   // call insert_fix_up to restore red - black tree's properties
47.   insert_fix_up();
48. }
```

Code 3 Η συνάρτηση *insert()* για την *ΕΙΣΑΓΩΓΗ* ενός κόμβου σε δομή red – black tree

Εάν το δέντρο είναι κενό, τότε ο νεοεισαχθείς κόμβος καταλαμβάνει θέση ρίζας στο δένδρο και αυτομάτως παραβιάζει την ιδιότητα 2. Η τελευταία γραμμή κώδικα της *insert_fix_up()* αποκαθιστά αυτού του είδους παραβιάσεις, όποτε και αν εμφανιστούν. Σε αντίθετη περίπτωση, όπου η δομή δεν είναι κενή, ο νεοεισαχθείς *κόκκινος* κόμβος αποκτά δύο *μαύρα* παιδιά NIL και ένα πατέρα χρώματος είτε *μαύρου*, είτε *κόκκινου*. Εάν ο πατέρας του νέου κόμβου είναι μαύρος, τότε όλες οι ιδιότητες διατηρούνται, χωρίς η εισαγωγή του νέου κόμβου να δημιουργεί κάποια παραβίαση. Αντίθετα, αν ο πατέρας είναι και αυτός *κόκκινος,* τότε άμεσα παραβιάζεται η 4^{η} ιδιότητα των red – black trees, η οποία θέλει και τα δύο παιδιά ενός *κόκκινου* κόμβου να είναι *μαύρα*. Συνολικά, οι περιπτώσεις στις οποίες ένας *κόκκινος* κόμβος αποκτά ένα επίσης *κόκκινο* παιδί

είναι έξι. Ωστόσο, ανά δύο, οι περιπτώσεις αυτές είναι απολύτως συμμετρικές μεταξύ τους. Η συμμετρία βρίσκεται ανάμεσα στα δύο υποδένδρα του *«παππού»* του (αριστερό/δεξιό). Παρακάτω αναλύονται οι τρεις περιπτώσεις που προκύπτουν όταν ο νέος κόμβος τοποθετείται στο αριστερό υποδένδρο του *«παππού»* του.

- **Περίπτωση 1: Ο *«θείος»* του νέου κόμβου είναι *κόκκινος***

  Στη κατάσταση αυτή, τόσο ο *πατέρας* του νέου κόμβου, όσο και ο αδελφός αυτού (που είναι *«θείος»* του νέου κόμβου), είναι και οι δύο κόκκινοι. Μέχρι πριν την εισαγωγή, όλες οι ιδιότητες τηρούνταν. Δεδομένης της τήρησης λοιπόν, ο *«παππούς»* του νέου κόμβου (*new_node->parent->parent*) θα πρέπει να είναι *μαύρος*. Η τελευταία συνθήκη, μας προτρέπει να χρωματίσουμε τον *πατέρα* και το *«θείο» μαύρους*. Με αυτό τον τρόπο, το πρόβλημα του κοινού *κόκκινου* χρώματος επιλύεται μεταξύ του νέου κόμβου και του πατρικού του. Ωστόσο, η ιδιότητα 5, επιβάλλει να χρωματίσουμε τον *«παππού» κόκκινο*, ώστε να μη μεταβληθεί το πλήθος των *μαύρων* κόμβων που περιέχονται κατά μήκος μίας διαδρομής προς αυτούς. Πλέον, ο έλεγχος μεταφέρεται στον *«παππού»*, δύο επίπεδα προς τα πάνω, καθώς το *κόκκινο* χρώμα του, ενδεχομένως να παραβιάζει εκ νέου την ιδιότητα 4 μεταξύ αυτού και του πατρικού του κόμβου.

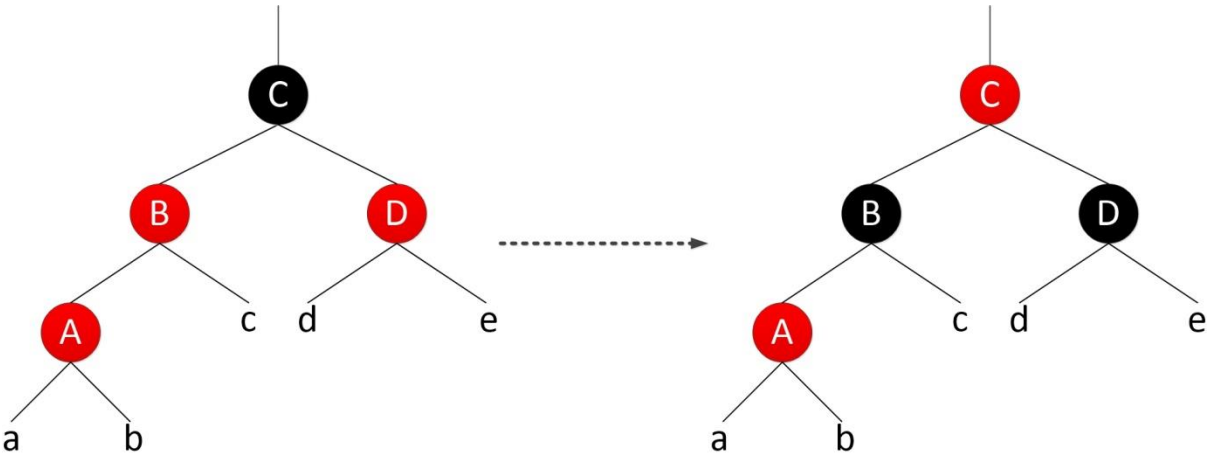

Figure 2 Η περίπτωση 1 της *insert_fix_up()*. Η εισαγωγή του νέου κόμβου A παραβιάζει την ιδιότητα 4. Η δράση της *insert_fix_up()* μεταβάλλει το *χρώμα* των κόμβων B, C και D. Πλέον, η ιδιότητα 4 μπορεί να παραβιάζεται μόνο από τον κόμβο C και τον πατρικό του.

- **Περίπτωση 2: Ο *«θείος»* του νέου κόμβου είναι *μαύρος* και ο νέος κόμβος είναι *δεξί_παιδί***
- **Περίπτωση 3: Ο *«θείος»* του νέου κόμβου είναι *μαύρος* και ο νέος κόμβος είναι *αριστερό_παιδί***

  Στις περιπτώσεις 2 και 3, το πρόβλημα του κοινού *κόκκινου* χρώματος μεταξύ του νέου κόμβου και του πατρικού του εμφανίζεται και πάλι. Αντίθετα από την περίπτωση 1 όμως, αυτή τη φορά ο *«θείος»* είναι *μαύρος*. Η διαφορά μεταξύ των δύο αυτών περιπτώσεων εντοπίζεται στη σχέση μεταξύ του νέου κόμβου και του *πατέρα* του. Στην περίπτωση 2, ο νέος κόμβος είναι *δεξιό_παιδί* του πατρικού του. Ωστόσο, με την εκτέλεση μίας αριστερής

περιστροφής (*left_rotate()*), ο νέος κόμβος γίνεται πατρικός του *πατέρα* του και αντίστροφα, ο πατρικός γίνεται *αριστερό_παιδί* του νέου κόμβου, όπως φαίνεται και στο παρακάτω σχήμα. Επειδή και οι δύο κόμβοι είναι *κόκκινοι*, καμία από τις υπόλοιπες ιδιότητες δε μεταβάλλεται, η ιδιότητα 4 εξακολουθεί να παραβιάζεται, ενώ η νέα τοπολογία περιγράφεται από την περίπτωση 3.

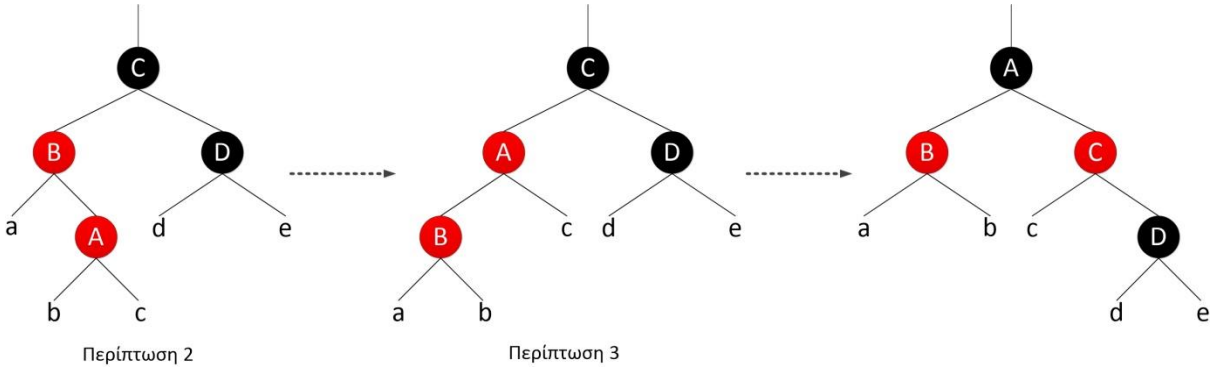

Figure 3 Οι περιπτώσεις 2 και 3 της *insert_fix_up()* όταν ο νέος κόμβος τοποθετείται στο αριστερό υποδένδρο του *«παππού»*. Η εισαγωγή του νέου κόμβου (A και B αντίστοιχα) παραβιάζει την ιδιότητα 4. Με την εκτέλεση μίας αριστερή περιστροφής, η περίπτωση 2 ανάγεται στην περίπτωση 3. Στη περίπτωση 3, η δράση της *insert_fix_up()* μεταβάλλει το *χρώμα* των κόμβων A και C και εκτελεί μία δεξιά περιστροφή.

Στην περίπτωση 3, ανεξάρτητα από τη προέλευση του κόμβου B στη θέση αυτή (είτε μέσω της περίπτωσης 2, είτε απευθείας από εισαγωγή), ο *«θείος»* του παραμένει *μαύρος*. Επιπλέον, ο *«παππούς»* παραμένει επίσης *μαύρος*, αφού ούτε η περίπτωση 2, ούτε η εισαγωγή νέου κόμβου τον επηρεάζουν. Σε αυτή τη φάση της αποκατάστασης, το πρόβλημα του κοινού *κόκκινου* χρώματος επιλύεται, αρχικά, με τον αναχρωματισμό του *πατέρα* και του *«παππού»* του B. Για να μη διαταραχθεί η ιδιότητα 5, η διαδικασία ολοκληρώνεται με την εφαρμογή μίας δεξιάς περιστροφής (*right_rotate()*) στον πατρικό κόμβο του B. Πλέον, στο δένδρο δεν μπορεί να υπάρχει κανένα ζεύγος διαδοχικών *κόκκινων* κόμβων και η εκτέλεση του βρόγχου `while` τερματίζει.

Η συνάρτηση *insert()* απαιτεί χρόνο O(log n) μέχρι να ενσωματώσει το νέο κόμβο στη δομή. Η αποκατάσταση των ιδιοτήτων που παραβιάζονται γίνεται από την *insert_fix_up()* και διαρκεί για όσο χρόνο εκτελείται η `while`. Κατά την αποκατάσταση, δεν εκτελούνται ποτέ περισσότερες από δύο περιστροφές καθώς, όταν εκτελεστούν οι περιπτώσεις 2 ή/και 3, ο βρόγχος τερματίζεται. Ωστόσο, η επανάληψη του βρόγχου οφείλεται στην περίπτωση 1. Κατά την εκτέλεσή της, το πρόβλημα του κοινού *κόκκινου* χρώματος επιλύεται και ο έλεγχος μεταφέρεται δύο επίπεδα προς τα πάνω, προκαλώντας το βρόγχο να εκτελεστεί O(log n) φορές. Τελικά ο συνολικός χρόνος που απαιτείται για την εκτέλεση της *insert()* είναι O(log n), αποτέλεσμα αναμενόμενο, αφού όπως έχουμε ήδη αναφέρει τα red – black trees είναι δυαδικά δένδρα αναζήτησης με ύψος O(log n).

```
1.  void insert_fix_up(){
2.
3.    // compliance of 4th property
4.    while (x_aux->parent->color == red){
5.      if (x_aux->parent == x_aux->parent->parent->left_child){
6.        y_aux = x_aux->parent->parent->right_child;
7.        if (y_aux->color == red){
8.          x_aux->parent->color = black;                 // case 1
9.          y_aux->color = black;                         // case 1
10.         x_aux->parent->parent->color = red;           // case 1
11.         x_aux = x_aux->parent->parent;                // case 1
12.       }
13.       else{
14.
15.         if (x_aux == x_aux->parent->right_child){
16.           x_aux = x_aux->parent;                      // case 2
17.           r_aux = x_aux;                              // case 2
18.           left_rotation();                            // case 2
19.         }
20.         x_aux->parent->color = black;                 // case 3
21.         x_aux->parent->parent->color = red;           // case 3
22.         r_aux = x_aux->parent->parent;                // case 3
23.         right_rotation();                             // case 3
24.       }
25.     }
26.     else if (x_aux->parent == x_aux->parent->parent->right_child){
27.       y_aux = x_aux->parent->parent->left_child;
28.       if (y_aux->color == red){
29.         x_aux->parent->color = black;                 // case 1
30.         y_aux->color = black;                         // case 1
31.         x_aux->parent->parent->color = red;           // case 1
32.         x_aux = x_aux->parent->parent;                // case 1
33.       }
34.       else{
35.         if (x_aux == x_aux->parent->left_child){
36.           x_aux = x_aux->parent;                      // case 2
37.           r_aux = x_aux;                              // case 2
38.           right_rotation();                           // case 2
39.         }
40.         x_aux->parent->color = black;                 // case 3
41.         x_aux->parent->parent->color = red;           // case 3
42.         r_aux = x_aux->parent->parent;                // case 3
43.         left_rotation();                              // case 3
44.       }
45.     }
46.   }
47.
48.   // compliance of 2nd property
49.   rbt_root->color = black;
50. }
```

Code 4 Η συνάρτηση *insert_fix_up()* για την αποκατάσταση των ιδιοτήτων 2, 4 και 5 μετά την *ΕΙΣΑΓΩΓΗ* ενός κόμβου σε δομή red – black tree

P2.4 Διαγραφή (Delete)

Όπως η *ΕΙΣΑΓΩΓΗ*, έτσι και η *ΔΙΑΓΡΑΦΗ* δεν διαφέρει αρκετά από την αντίστοιχη πράξη σε ένα κοινό δυαδικό δέντρο. Οι όποιες διαφορές οφείλονται και πάλι στις ιδιότητες που πρέπει να τηρούνται. Αρχικά, εντοπίζεται ο διαγραφόμενος κόμβος και ανάλογα με τους απογόνους του, αποφασίζεται ποιος κόμβος θα τον

διαδεχθεί. Εάν ο διαγραφόμενος κόμβος έχει *κόκκινο χρώμα*, οι ιδιότητες των red – black trees εξακολουθούν να ισχύουν.

```
1.  void delete(){
2.
3.    int key;
4.    status y_original_color;
5.
6.    printf("\n Give me the the node's key\t:\t");
7.    scanf("%d", &key);
8.    getchar();
9.
10.   // search node to delete
11.   find_node(key);
12.   if (z_aux == NIL)
13.     printf("\n Node with key = %d not found", key);
14.   else
15.     printf("\n %d = %d. Found node to delete", key, z_aux->key);
16.
17.   // search descendant to replace deleted node
18.   if ((z_aux->left_child == NIL) || (z_aux->right_child == NIL)){
19.     y_aux = z_aux;
20.   }
21.   else{
22.     find_predecessor();
23.   }
24.
25.   // fix pointers of deleted node's immediate relatives and descendant
26.   if (y_aux->left_child != NIL){
27.     x_aux = y_aux->left_child;
28.   }
29.   else{
30.     x_aux = y_aux->right_child;
31.   }
32.   x_aux->parent = y_aux->parent;
33.
34.   if (y_aux->parent == NIL){
35.     rbt_root = x_aux;
36.   }
37.   else{
38.     if (y_aux == y_aux->parent->left_child){
39.       y_aux->parent->left_child = x_aux;
40.     }
41.     else{
42.       y_aux->parent->right_child = x_aux;
43.     }
44.   }
45.
46.   // copy descendant properties to deleted node's position
47.   if (y_aux != z_aux){
48.     z_aux->key = y_aux->key;
49.   }
50.
51.   // delete proper node
52.   y_original_color = y_aux->color;
53.   free(y_aux);
54.
55.   // call delete_fix_up to restore red - black tree's properties, if needed
56.   if (y_original_color == black){
57.     delete_fix_up();
58.   }
59. }
```

Code 5 Η συνάρτηση *delete()* για τη *ΔΙΑΓΡΑΦΗ* ενός κόμβου από δομή red – black tree

Αντίθετα, αν ο διαγραφόμενος κόμβος είναι *μαύρος,* τότε όλοι οι πρόγονοι του κόμβου που διαγράφηκε παραβιάζουν την ιδιότητα 5 καθώς, όλες οι διαδρομές που τον περιλάμβαναν πριν διαγραφεί, τώρα θα έχουν ένα *μαύρο* κόμβο λιγότερο. Επιπλέον, υπάρχει πιθανότητα να έχουν παραβιασθεί οι ιδιότητες 2, αν διαγραφεί ο ριζικός κόμβος και τη θέση του πάρει ένας *κόκκινος* κόμβος, και η 4, εάν το θυγατρικός και ο πατρικός του διαγραφέντα κόμβου είναι και οι δύο *κόκκινοι*. Η αποκατάσταση των ιδιοτήτων των red – black trees γίνεται με την κλήση της συνάρτησης *delete_fix_up()* πριν την ολοκλήρωση της διαγραφής.

Για την αποκατάσταση της ιδιότητας 5 χρησιμοποιούμε το εξής τέχνασμα. Αρχικά, ορίζουμε ένα δείκτη x_aux ο οποίος δείχνει σε ένα από τα παιδιά του κόμβου που διαγράφουμε και θεωρούμε ότι μετά τη διαγραφή, ο δείκτης αυτός αποκτά «μαύρο χαρακτήρα», εξαιτίας του κόμβου που διεγράφη. Ο «μαύρος χαρακτήρας» του δείκτη μεταβιβάζεται στον κόμβο που δείχνει κάθε φορά και συνεισφέρει κατά μία μονάδα επιπλέον στο πλήθος των μαύρων κόμβων των διαδρομών που τον περιέχουν. Με αυτό τον τρόπο, ουσιαστικά αποκαθίσταται η ιδιότητα 5, ωστόσο η παραβίαση μεταφέρεται στην ιδιότητα 1. Το *χρώμα* του κόμβου που δείχνει ο x_aux, πλέον δεν είναι ούτε *μαύρο* ούτε *κόκκινο*· επηρεάζεται από το «μαύρο χαρακτήρα» του δείκτη και ο κόμβος γίνεται είτε κοκκινόμαυρος, είτε διπλά μαύρος.

Κατά την εκτέλεση της *delete_fix_up()* γίνεται αποκατάσταση των ιδιοτήτων 1, 2 και 4. Η εκτέλεση του βρόγχου `while` μετατοπίζει τον πρόσθετο «μαύρο χαρακτήρα» προς τα επάνω μέχρι:

1. ο δείκτης να δείχνει σε έναν κοκκινόμαυρο κόμβο, όπου απλά τον χρωματίζει *μαύρο*
2. ο δείκτης να δείχνει στο ριζικό κόμβο, όπου απλώς αφαιρεί τον επιπλέον «μαύρο χαρακτήρα»
3. να μπορούν να εκτελεστούν κατάλληλες περιστροφές και αναχρωματισμοί

Στο εσωτερικό του βρόγχου εξετάζεται ο κόμβος που δείχνει ο x_aux σε σχέση με τον *«αδελφό»* του και τα *παιδιά* του τελευταίου. Οι περιπτώσεις είναι και πάλι συμμετρικές, ανάλογα με το αν ο x_aux δείχνει σε αριστερό ή δεξιό παιδί. Παρακάτω αναλύονται οι διαφορετικές περιπτώσεις που προκύπτουν όταν ο εξεταζόμενος κόμβος είναι *αριστερό_παιδί*.

- **Περίπτωση 1: Ο *«αδελφός»* του εξεταζόμενου κόμβου είναι *κόκκινος***

  Το γεγονός ότι ο αδελφικός κόμβος είναι *κόκκινος* συνεπάγεται ότι και τα δύο *παιδιά* του πρέπει να είναι *μαύρα*. Χωρίς να παραβιάσουμε καμία ιδιότητα, μπορούμε αρχικά να αλλάξουμε το *χρώμα* του αδελφικού και του πατρικού κόμβου. Στη συνέχεια, εκτελώντας μία περιστροφή, ο νέος αδελφικός κόμβος θα είναι ένα από τα παιδιά του προηγούμενου αδελφού, η περίπτωση 1 ανάγεται σε μία από τις περιπτώσεις 2, 3 ή 4.

  Οι περιπτώσεις 2, 3 και 4 προκύπτουν όταν ο «*αδελφός»* του εξεταζόμενου κόμβου είναι *μαύρος*. Η διαφορές τους εντοπίζονται στο *χρώμα* των θυγατρικών του *«αδελφού»*.

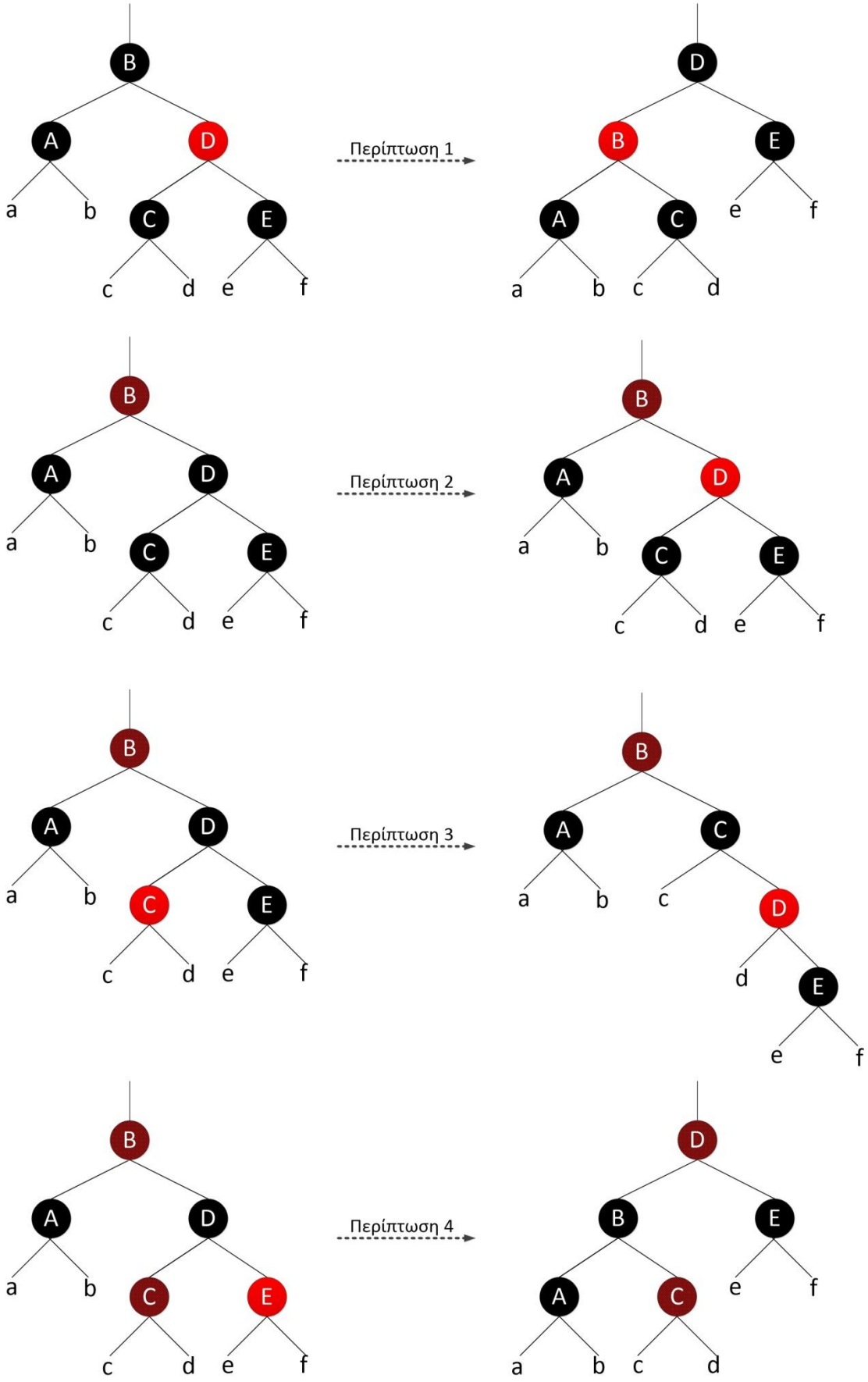

Figure 4 Οι περιπτώσεις της *delete_fix_up()* όταν ο x_aux δείχνει σε αριστερό παιδί. Εδώ, πριν την εκτέλεση της κάθε περίπτωσης, ο x_aux δείχνει τον κόμβο A

- **Περίπτωση 2: Ο *«αδελφός»* του εξεταζόμενου κόμβου είναι *μαύρος*, και οι δύο θυγατρικοί κόμβοι του *«αδελφού»* είναι επίσης *μαύροι***

  Στην περίπτωση αυτή χρωματίζουμε τον αδελφικό κόμβο *κόκκινο* και μεταφέρουμε το μελανό χαρακτήρα του δείκτη ένα επίπεδο προς τα πάνω, στον πατρικό κόμβο. Ουσιαστικά, αφαιρούμε και από τους δύο κόμβους ένα μελανό χαρακτήρα, αφήνοντας στον εξεταζόμενο κόμβο το *χρώμα* που είχε πριν πάρει τον επιπλέον «μαύρο χαρακτήρα». Ο πατρικός κόμβος τώρα θα είναι είτε κοκκινόμαυρος, είτε διπλά μαύρος και ο βρόγχος εκτελείται, εξετάζοντας πλέον τον πατρικό κόμβο. Εάν η μετάβαση του ελέγχου έχει γίνει από την περίπτωση 1, τότε ο εξεταζόμενος κόμβος, πριν μελετηθεί στην περίπτωση 2, χρωματίζεται *κόκκινος*. Στην περίπτωση αυτή, η συνθήκη ελέγχου του βρόγχου δεν ικανοποιείται και ο βρόγχος τερματίζει. Πριν τον τερματισμό της διαδικασίας, ο εξεταζόμενος κόμβος στον οποίο δείχνει ο x_aux χρωματίζεται *μαύρος*, ικανοποιώντας την ιδιότητα 4.

- **Περίπτωση 3: Ο *«αδελφός»* του εξεταζόμενου κόμβου είναι *μαύρος*, το *αριστερό_παιδί* του *«αδελφού»* είναι *κόκκινο,* ενώ το *δεξιό_παιδί του* είναι *κόκκινο***

  Στην κατάσταση αυτή, αρχικά αλλάζουμε το χρώμα των κόμβων *«αδελφός»* και *αριστερό_παιδί* (*του «αδελφού»*) και στη συνέχεια εκτελούμε μία δεξιά περιστροφή στον *«αδελφό»*. Καμία επιπλέον από τις ιδιότητες δεν παραβιάζεται κατά τη διάρκεια της παραπάνω διαδικασίας, ενώ ταυτόχρονα, ο *αριστερό_παιδί* του *«αδελφού»* γίνεται ο νέος *«αδελφός»* του εξεταζόμενου κόμβου και είναι μαύρος. Ο πρώην *«αδελφός»*, και πρώην *πατέρας* του νέου *«αδελφού»*, πλέον έχει γίνει *αριστερό_παιδί* του νέου *«αδελφού»* και είναι *κόκκινος*. Η νέα διάταξη παραπέμπει στην περίπτωση 4.

- **Περίπτωση 4: Ο *«αδελφός»* του εξεταζόμενου κόμβου είναι *μαύρος* και το *δεξιό_παιδί του* είναι *κόκκινο***

  Στην παρούσα φάση, αφού αναχρωματίσουμε τον *πατέρα* του εξεταζόμενου κόμβου, τον *«αδελφό»* του, αλλά και το *δεξιό_παιδί* του, εκτελείται μία αριστερή περιστροφή στον *πατέρα*. Η διαδικασία δε μεταβάλλει καμία από τις ισχύουσες ιδιότητες της δομής, ενώ ταυτόχρονα εξαλείφει τον πρόσθετο *μαύρο χαρακτήρα* από τον εξεταζόμενο κόμβο, μεταφέροντάς τον στη ρίζα του red – black tree, ώστε να ικανοποιηθεί η ιδιότητα 2. Καθώς η συνθήκη ελέγχου δεν ικανοποιείται πλέον, η εκτέλεση η εκτέλεση του βρόγχου τερματίζεται.

Αξίζει να τονισθεί ότι κατά την εκτέλεση οποιασδήποτε περίπτωσης, αποκαθίσταται μόνο η ιδιότητα 1. Η αποκατάσταση των ιδιοτήτων 2 και 4 προοικονομούνται στις περιπτώσεις 4 και 2, αντίστοιχα, και ολοκληρώνονται στην τελευταία γραμμή κώδικα της *insert_fix_up()*.

```
1.  void delete_fix_up(){
2.    while ((x_aux != rbt_root) && (x_aux->color == black)){ // compliance of 1st property
3.      if (x_aux == x_aux->parent->left_child){
4.        z_aux = x_aux->parent->right_child;
5.        if (z_aux->color == red){
6.          z_aux->color = black;                   // case 1
7.          x_aux->parent->color = red;             // case 1
8.          r_aux = x_aux->parent;                  // case 1
9.          left_rotation();                        // case 1
10.         z_aux = x_aux->parent->right_child;     // case 1
11.       }
12.       if ((z_aux->left_child->color == black) && (z_aux->right_child->color == black)){
13.         z_aux->color = red;                     // case 2
14.         x_aux = x_aux->parent;                  // case 2
15.       }
16.       else{
17.         if (z_aux->right_child->color == black){
18.           z_aux->left_child->color = black;     // case 3
19.           z_aux->color = red;                   // case 3
20.           r_aux = z_aux;                        // case 3
21.           right_rotation();                     // case 3
22.           z_aux = x_aux->parent->right_child;   // case 3
23.         }
24.         z_aux->color = x_aux->parent->color;    // case 4
25.         x_aux->parent->color = black;           // case 4
26.         z_aux->right_child->color = black;      // case 4
27.         r_aux = x_aux->parent;                  // case 4
28.         left_rotation();                        // case 4
29.         x_aux = rbt_root;                       // case 4
30.       }
31.     }
32.     else{// if (x_aux == x_aux->parent->right_child){
33.     z_aux = x_aux->parent->left_child;
34.       if (z_aux->color == red){
35.         z_aux->color = black;                   // case 1
36.         x_aux->parent->color = red;             // case 1
37.         r_aux = x_aux->parent;                  // case 1
38.         right_rotation();                       // case 1
39.         z_aux = x_aux->parent->left_child;      // case 1
40.       }
41.       if ((z_aux->left_child->color == black) && (z_aux->right_child->color == black)){
42.         z_aux->color = red;                     // case 2
43.         x_aux = x_aux->parent;                  // case 2
44.       }
45.       else{
46.         if (z_aux->left_child->color == black){
47.           z_aux->right_child->color = black;    // case 3
48.           z_aux->color = red;                   // case 3
49.           r_aux = z_aux;                        // case 3
50.           left_rotation();                      // case 3
51.           z_aux = x_aux->parent->left_child;    // case 3
52.         }
53.         z_aux->color = x_aux->parent->color;    // case 4
54.         x_aux->parent->color = black;           // case 4
55.         z_aux->left_child->color = black;       // case 4
56.         r_aux = x_aux->parent;                  // case 4
57.         right_rotation();                       // case 4
58.         x_aux = rbt_root;                       // case 4
59.       }
60.     }
61.   }
62.   x_aux->color = black; // compliance of properties 2 & 4
63. }
```

Code 6 Η συνάρτηση *delete_fix_up()* για την αποκατάσταση των ιδιοτήτων 1, 2 και 4 μετά τη *ΔΙΑΓΡΑΦΗ* ενός κόμβου από δομή red – black tree

Αφού όπως έχουμε πει, τα red – black trees είναι δυαδικά δέντρα αναζήτησης, αναμένουμε ο χρόνος εκτέλεσης της *ΔΙΑΓΡΑΦΗΣ* να είναι O(log n), επίσης. Οι συναρτήσεις *find_node()* και *find_predecessor(),* που καλούνται στις γραμμές 11 και 22 αντίστοιχα, της *delete()*, εκτελούν και οι δύο από μία πράξη *ΑΝΑΖΗΤΗΣΗ*. Ήδη όμως γνωρίζουμε ότι μία τέτοια διαδικασία, για την ολοκλήρωσή της σε δυαδικό δένδρο αναζήτησης, απαιτεί χρόνο O(log n). Όλες οι υπόλοιπες εντολές της *delete()* απαιτούν χρόνο O(1), εκτός από τη κλήση της *insert_fix_up()*. Εντός της συνάρτησης αποκατάστασης, εκτελείτε κάμποσες φορές ο βρόγχος `while` και ένας αναχρωματισμός χρόνου O(1). Καθώς ο βρόγχος περιλαμβάνει μία σειρά από αναχρωματισμούς, μετατοπίσεις και περιστροφές, το ερώτημα είναι πόσες φορές εκτελείται ο ίδιος ο βρόγχος. Η περίπτωση 4 εκτελείται μία μόνο φορά και ο βρόγχος τερματίζει. Ομοίως, η περίπτωση 3 εκτελείται επίσης μία φορά, αφού μετασχηματίζεται στην περίπτωση 4. Η περίπτωση 1, όπως είπαμε μετασχηματίζεται στις περιπτώσεις 2, 3 και 4. Εάν από την 1, ο έλεγχος μεταφερθεί στις περιπτώσεις 3 ή/και 4, τότε ισχύουν τα παραπάνω. Όταν όμως, μεταφερθεί στην περίπτωση 2, ο χρόνος εκτέλεσής της εξαρτάται από αυτή. Όπως είδαμε παραπάνω, στη περίπτωση 2 αναχρωματίζεται ο *«αδελφός»* ο έλεγχος μεταφέρεται στον πατρικό του εξεταζόμενου κόμβου. Αυτό πρακτικά σημαίνει ότι μπορεί να ανέβει O(log n) φορές προς τα πάνω, και άρα ο βρόγχος να εκτελεστεί και αυτός O(log n) φορές. Τελικά, και η *delete()* εκτελείται σε χρόνο O(log n), όπως είχαμε προβλέψει.

P2.5 Ο συνολικός κώδικας

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef int bool;
#define true 1
#define false 0

typedef enum {
  red,
  black
} status;

typedef struct node *node_ptr;
struct node{
  int key;
  status color;
  node_ptr parent, left_child, right_child;
};

//global variables' declaration
node_ptr rbt_root, r_aux, x_aux, y_aux, z_aux, NIL;

//functions' declaration
void right_rotation();
void left_rotation();
void initialization();
void insert_fix_up();
void insert();
void find_predecessor();
void delete_fix_up();
void find_node(int key);
void delete();
```

```
void tree_print(node_ptr *nd);
void print();

int main(void){

  char c1='a';

  initialization();
  rbt_root = NIL;

  while ((c1 != 'q') && (c1 != 'Q')){
    printf("\n Your options are :\n (i)nsert\n (d)elete\n (p)rint\n (q)uit");
    if ((c1 != 'q') || (c1 != 'Q')){
      printf("\n Give new choice: ");
            c1='a';
            fflush(stdin);
            c1=getchar();
            getchar();

      if ((c1 == 'q') || (c1 == 'Q')){
              printf("\n Quit \n");
          }
      if ((c1 == 'i') || (c1 == 'I')){
        printf("\n INSERT \n");
                insert();
          }
          if ((c1 == 'd') || (c1 == 'D')){
            printf("\n DELETE \n");
                de1ete();
          }
            if ((c1 == 'p') || (c1 == 'P')){
        printf("\n PRINT \n");
                print();
            }
        }
    }

  return 0;
}



void initialization(){

  NIL = (node_ptr)malloc(sizeof(struct node));
  NIL->key = (int)INFINITY;//0;
  NIL->color = black;
  NIL->parent = NULL;
  NIL->left_child = NULL;
  NIL->right_child = NULL;
}


void right_rotation(){

  // finding child participating in rotation
  z_aux = r_aux->left_child;

  // converting z_aux's right sub-tree in r_aux's left sub-tree
  r_aux->left_child = z_aux->right_child;
  if (z_aux->right_child != NIL){
    z_aux->right_child->parent = r_aux;
  }

  // connecting r_aux's parent with z_aux
  z_aux->parent = r_aux->parent;
  if (r_aux->parent == NIL){
    rbt_root = z_aux;
```

```
  }
  else{
    if (r_aux == r_aux->parent->right_child){
      r_aux->parent->right_child = z_aux;
    }
    else{
      r_aux->parent->left_child = z_aux;
    }
  }

  // setting r_aux as z_aux's right child
  z_aux->right_child = r_aux;
  r_aux->parent = z_aux;
}


void left_rotation(){

  // finding child participating in rotation
  z_aux = r_aux->right_child;

  // converting z_aux's left sub-tree in r_aux's right sub-tree
  r_aux->right_child = z_aux->left_child;
  if (z_aux->left_child != NIL){
    z_aux->left_child->parent = r_aux;
  }

  // connecting r_aux's parent with z_aux
  z_aux->parent = r_aux->parent;
  if (r_aux->parent == NIL){
    rbt_root = z_aux;
  }
  else{
    if (r_aux == r_aux->parent->left_child){
      r_aux->parent->left_child = z_aux;
    }
    else{
      r_aux->parent->right_child = z_aux;
    }
  }

  // setting r_aux as z_aux's right child
  z_aux->left_child = r_aux;
  r_aux->parent = z_aux;
}


void insert_fix_up(){

  // compliance of 4th property
  while (x_aux->parent->color == red){
    printf("\n Node and parent are both red. ");
    if (x_aux->parent == x_aux->parent->parent->left_child){
      y_aux = x_aux->parent->parent->right_child;
      if (y_aux->color == red){
        printf("Uncle of node is red -- push blackness down from grandparent");
        x_aux->parent->color = black;          // case 1
        y_aux->color = black;                  // case 1
        x_aux->parent->parent->color = red;    // case 1
        x_aux = x_aux->parent->parent;         // case 1
      }
      else{

        if (x_aux == x_aux->parent->right_child){
          printf("Node is right child, parent is left child -- rotate");
          x_aux = x_aux->parent;               // case 2
          r_aux = x_aux;                       // case 2
          printf("\n Single rotate left");     // case 2
```

```
          left_rotation();                    // case 2
          printf("\n Node and parent are both red. ");
        }
        printf("Node is left child, parent is left child\n Can fix extra redness
with a single rotation");
        x_aux->parent->color = black;         // case 3
        x_aux->parent->parent->color = red;   // case 3
        r_aux = x_aux->parent->parent;        // case 3
        printf("\n Single rotate right");     // case 3
        right_rotation();                     // case 3
      }
    }
    else if (x_aux->parent == x_aux->parent->parent->right_child){
      y_aux = x_aux->parent->parent->left_child;
      if (y_aux->color == red){
        printf("Uncle of node is red -- push blackness down from grandparent");
        x_aux->parent->color = black;         // case 1
        y_aux->color = black;                 // case 1
        x_aux->parent->parent->color = red;   // case 1
        x_aux = x_aux->parent->parent;        // case 1
      }
      else{

        if (x_aux == x_aux->parent->left_child){
          printf("Node is left child, parent is right child -- rotate");
          x_aux = x_aux->parent;              // case 2
          r_aux = x_aux;                      // case 2
          printf("\n Single rotate right");   // case 2
          right_rotation();                   // case 2
          printf("\n Node and parent are both red. ");
        }
        printf("Node is right child, parent is right child\n Can fix extra redness
with a single rotation");
        x_aux->parent->color = black;         // case 3
        x_aux->parent->parent->color = red;   // case 3
        r_aux = x_aux->parent->parent;        // case 3
        printf("\n Single rotate left");      // case 3
        left_rotation();                      // case 3
      }
    }
  }
  // compliance of 2nd property
  if (rbt_root->color == red)
    printf("\n Root of the tree is red. Color it black");
  rbt_root->color = black;
  printf("\n");
}


void insert(){

  int key;
  node_ptr aux = NIL, f_aux = rbt_root;
  /*
    f_aux : f(orward)_aux pointing one of aux's childs
  */

  printf("\n Give new node's key\t:\t");
  scanf("%d", &key);
  getchar();

  // construction of new struct node
  x_aux = (node_ptr)malloc(sizeof(struct node));
  x_aux->key = key;
  x_aux->color = red;      // new node is colored red
  x_aux->parent = NIL;
  x_aux->left_child = NIL; // new node's left_child points NIL
  x_aux->right_child = NIL;// new node's right_child points NIL
```

```
  // searching new node's position in red - black tree
  printf("\n Inserting %d", x_aux->key);
  while (f_aux != NIL){
    aux = f_aux;
    if (x_aux->key < f_aux->key){
      printf("\n %d < %d. Looking at left subtree", x_aux->key, f_aux->key);
      f_aux = f_aux->left_child;
    }
    else{// if (x_aux->key >= f_aux->key){
      printf("\n %d >= %d. Looking at right subtree", x_aux->key, f_aux->key);
      f_aux = f_aux->right_child;
    }
  }
  printf("\n Found null tree (or guard NIL), inserting element");

  // fixing pointers between new node and its parent
  x_aux->parent = aux;
  if (aux == NIL){
    rbt_root = x_aux;
  }
  else{
    if (x_aux->key < aux->key){
      aux->left_child = x_aux;
    }
    else{// if (x_aux->key >= aux->key){
      aux->right_child = x_aux;
    }
  }

  // call insert_fix_up to restore red - black tree's properties
  insert_fix_up();
}


void find_predecessor(){
  //searches for predecessor

  if (z_aux->left_child != NIL){
    y_aux = z_aux->left_child;
    while(y_aux->right_child != NIL){
      y_aux = y_aux->right_child;
    }
  }
  else{
    y_aux = z_aux->parent;
    while ((y_aux != NIL) && (z_aux = y_aux->left_child)){
      z_aux = y_aux;
      y_aux = y_aux->parent;
    }
  }
}


void delete_fix_up(){
  // compliance of 1st property
  while ((x_aux != rbt_root) && (x_aux->color == black)){
    if (x_aux == x_aux->parent->left_child){
      z_aux = x_aux->parent->right_child;
      if (z_aux->color == red){
        printf("\n Double black node has black sibling. Rotate tree to make sibling
black...");
        z_aux->color = black;                  // case 1
        x_aux->parent->color = red;            // case 1
        r_aux = x_aux->parent;                 // case 1
        printf("\n Single rotate left");
        left_rotation();                       // case 1
        z_aux = x_aux->parent->right_child;    // case 1
```

```
      }
      if ((z_aux->left_child->color == black) && (z_aux->right_child->color ==
black)){
        printf("\n Double black node has black sibling and 2 black nephews. Push up
black level");
        z_aux->color = red;                    // case 2
        x_aux = x_aux->parent;                 // case 2
      }
      else{

        if (z_aux->right_child->color == black){
          printf("\n Double black node has black siblings, but double black node is
a left child,");
          printf("\n and the right nephew is black. Rotate tree to make opposite
nephew red...");
          z_aux->left_child->color = black;    // case 3
          z_aux->color = red;                  // case 3
          r_aux = z_aux;                       // case 3
          printf("\n Single rotate right");
          right_rotation();                    // case 3
          z_aux = x_aux->parent->right_child;  // case 3
        }
        printf("\n Double black node has black sibling, is a left child, and its
right nephew is red");
        printf("\n One rotation can fix double blackness.");
        z_aux->color = x_aux->parent->color;   // case 4
        x_aux->parent->color = black;          // case 4
        z_aux->right_child->color = black;     // case 4
        r_aux = x_aux->parent;                 // case 4
        printf("\n Single rotate left");
        left_rotation();                       // case 4
        x_aux = rbt_root;                      // case 4
      }
    }
    else{// if (x_aux == x_aux->parent->right_child){
      z_aux = x_aux->parent->left_child;
      if (z_aux->color == red){
        printf("\n Double black node has black sibling. Rotate tree to make sibling
black...");
        z_aux->color = black;                  // case 1
        x_aux->parent->color = red;            // case 1
        r_aux = x_aux->parent;                 // case 1
        printf("\n Single rotate right");
        right_rotation();                      // case 1
        z_aux = x_aux->parent->left_child;     // case 1
      }
      if ((z_aux->left_child->color == black) && (z_aux->right_child->color ==
black)){
        printf("\n Double black node has black sibling and 2 black nephews. Push up
black level");
        z_aux->color = red;                    // case 2
        x_aux = x_aux->parent;                 // case 2
      }
      else{

        if (z_aux->left_child->color == black){
          printf("\n Double black node has black siblings, but double black node is
a right child,");
          printf("\n and the left nephew is black. Rotate tree to make opposite
nephew red...");
          z_aux->right_child->color = black;   // case 3
          z_aux->color = red;                  // case 3
          r_aux = z_aux;                       // case 3
          printf("\n Single rotate left");
          left_rotation();                     // case 3
          z_aux = x_aux->parent->left_child;   // case 3
        }
```

```
        printf("\n Double black node has black sibling, is a right child, and its
left nephew is red");
        printf("\n One rotation can fix double blackness.");
        z_aux->color = x_aux->parent->color;   // case 4
        x_aux->parent->color = black;          // case 4
        z_aux->left_child->color = black;      // case 4
        r_aux = x_aux->parent;                 // case 4
        printf("\n Single rotate right");
        right_rotation();                      // case 4
        x_aux = rbt_root;                      // case 4
      }
    }
  }
  // compliance of properties 2 & 4
  x_aux->color = black;
  printf("\n");
}


void find_node(int key){

  z_aux = rbt_root;

  while ((z_aux != NIL) && (z_aux->key != key)){

    if (key < z_aux->key){
      printf("\n %d < %d. Looking at left subtree", key, z_aux->key);
      z_aux = z_aux->left_child;
    }
    else{
      printf("\n %d > %d. Looking at left subtree", key, z_aux->key);
      z_aux = z_aux->right_child;
    }
  }
}


void delete(){

  int key;
  status y_original_color;

  printf("\n Give me the the node's key\t:\t");
  scanf("%d", &key);
    getchar();

  // search node to delete
    printf("\n Deleting %d", key);
    find_node(key);
    if (z_aux == NIL)
    printf("\n Node with key = %d not found", key);
  else
    printf("\n %d = %d. Found node to delete", key, z_aux->key);

  // search descendant to replace deleted node
  if ((z_aux->left_child == NIL) || (z_aux->right_child == NIL)){
    y_aux = z_aux;
  }
  else{
    printf("\n Node to delete has two children.\n Find largest node in left
subtree.");
    find_predecessor();
  }

  // fix pointers of deleted node's immediate relatives and descendant
  if (y_aux->left_child != NIL){
    x_aux = y_aux->left_child;
  }
```

```
  else{
    x_aux = y_aux->right_child;
  }

  x_aux->parent = y_aux->parent;

  if (y_aux->parent == NIL){
    rbt_root = x_aux;
  }
  else{
    if (y_aux == y_aux->parent->left_child){
      printf("\n Set parent of deleted node to left child of deleted node");
      y_aux->parent->left_child = x_aux;
    }
    else{
      if ((y_aux->left_child == NIL)&&(y_aux == z_aux))
        printf("is a leaf. Delete it");
      else if (y_aux == z_aux)
        printf("has no left child.\n Set parent of deleted node to right child of
deleted node");
      y_aux->parent->right_child = x_aux;
    }
  }

  // copy descendant properties to deleted node's position
  if (y_aux != z_aux){
    printf("\n Copy largest value of left subtree into node to delete");
    z_aux->key = y_aux->key;
  }

  // delete proper node
  y_original_color = y_aux->color;
  printf("\n Remove node");
  free(y_aux);

  // call delete_fix_up to restore red - black tree's properties, if needed
  if (y_original_color == black){
    delete_fix_up();
  }

}

void tree_print(node_ptr *nd){

  node_ptr aux = (*nd);

  if (aux != NIL){
    if (aux->left_child != NIL){
      tree_print(&(aux->left_child));
    }

    printf("%d\t", aux->key);

    if (aux->color == red)
      printf("red\t");
    else if (aux->color == black)
      printf("black\t");

    if (aux->parent == NIL)
      printf("NIL\t");
    else
      printf("%d\t", aux->parent->key);

    if (aux->left_child == NIL)
      printf("NIL\t");
    else
      printf("%d\t", aux->left_child->key);
```

```
    if (aux->right_child == NIL)
      printf("    NIL\t");
    else
      printf("    %d\t", aux->right_child->key);

    if (aux->parent == NIL)
      printf("\t<--(root)\n ", aux->key);
    else
      printf("\n ");

    if (aux->right_child != NIL){
      tree_print(&(aux->right_child));
    }
  }
}


void print(){

  printf("\n KEY\tCOLOR\tPARENT\tLEFT_CHILD  RIGHT_CHILD\n");
  printf("-------------------------------------------------\n ");
  tree_print(&rbt_root);
}
```