ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2017-2018
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ
ΜΑΘΗΜΑ: ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

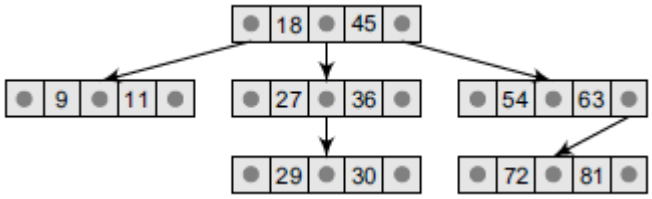
ΕΡΓΑΣΤΗΡΙΟ 8 – ΣΗΜΕΙΩΣΕΙΣ

Multi-way search Trees

Στα δυαδικά δένδρα αναζήτησης, κάθε κόμβος έχει το πολύ δύο παιδιά: το αριστερό και το δεξιό. Στα M-δένδρα, κάθε κόμβος έχει M-1 κλειδιά και M παιδιά. Το M καλείται ο βαθμός του δένδρου (degree of the tree). Στα δυαδικά δένδρα έχουμε M=2. Σχηματικά ένας κόμβος σε ένα M-δένδρο έχει ως εξής:

P_0	K_0	P_1	K_1	P_2	K_2	…	K_{M-2}	P_{M-1}

Όπου P_0, P_1, ... είναι οι δείκτες προς τα υπο-δένδρα και K_1, K_2, ... είναι τα κλειδιά του κόμβου. Τα κλειδιά αποθηκεύονται με τέτοιο τρόπο ώστε να ισχύει (αύξουσα σειρά): $K_i < K_{i+1}$, $0 <= i <= M-2$. Επίσης, δεν είναι υποχρεωτικό κάποιος κόμβος να περιλαμβάνει ακριβώς M-1 κλειδιά και M υπο-δένδρα. Το ακόλουθο σχήμα απεικονίζει ένα παράδειγμα ενός M-δένδρου.



Πρακτικά, το M μπορεί να είναι αρκετά μεγάλο. Οι κανόνες που ισχύουν για ένα M-δένδρο είναι οι ακόλουθοι:

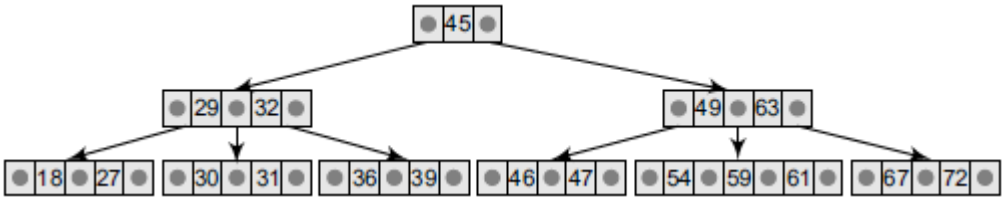
- Όλα τα κλειδιά που αποθηκεύονται στο P0 υπο-δένδρο είναι μικρότερα από το K0. Ομοίως, τα κλειδιά στο υπο-δένδρο που δείχνει ο P_1, είναι μικρότερα από το K_1, κοκ.
- Όλα τα κλειδιά που αποθηκεύονται στο υπο-δένδρο που δείχνει ο δείκτης P_1 είναι μεγαλύτερα από το K_0, όλα τα κλειδιά που βρίσκονται στο P_2 είναι μεγαλύτερα από το K_1, κοκ.

B Trees

Τα B-δένδρα είναι ειδική περίπτωση των M-δένδρων και υιοθετούνται για την προσπέλαση σε δίσκους. Τα B-δένδρα έχουν όλα τα χαρακτηριστικά των M-δένδρων και επιπλόεν:

- Κάθε κόμβος έχει **το πολύ m παιδιά** (το δένδρο είναι τάξης m).
- Κάθε κόμβος εκτός από τη ρίζα και τα φύλλα έχει **τουλάχιστον m/2 παιδιά**. Αυτό το χαρακτηριστικό κάνει τα B-δένδρα να χαρακτηρίζονται από μικρού μήκους μονοπάτια από τη ρίζα μέχρι και τα φύλλα ακόμα και αν το δένδρο αποθηκεύει πολλά δεδομένα.
- Η ρίζα έχει τουλάχιστον **δύο παιδιά** αν δεν είναι φύλλα.
- **Όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο**.

Δεν είναι απαραίτητο όλοι οι κόμβοι να έχουν τον ίδιο αριθμό παιδιών αλλά να έχουν τουλάχιστον m/2 παιδιά. Σχηματικά ένα B-δένδρο τάξης 4 έχει ως εξής:

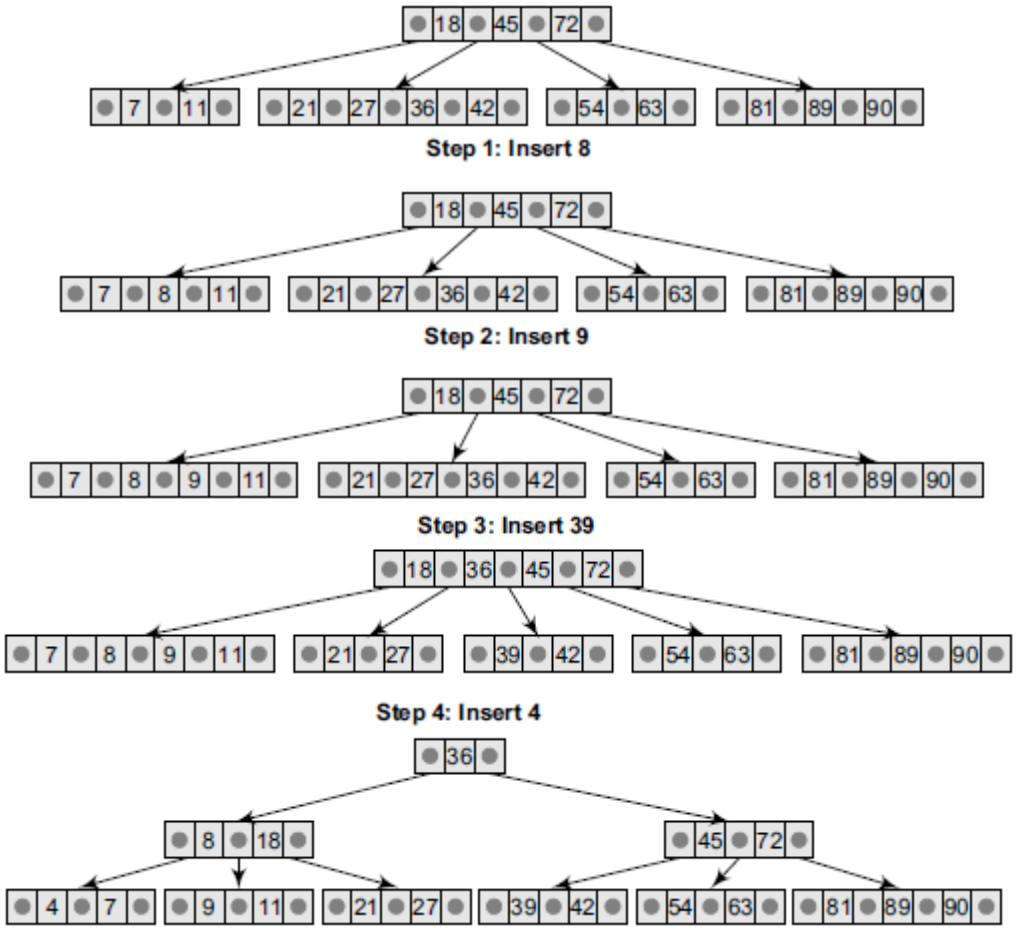


Αναζήτηση στοιχείου: Η αναζήτηση στοιχείου είναι παραπλήσια με την αναζήτηση σε ένα δυαδικό δένδρο. Στο δένδρο της παραπάνω εικόνας για την αναζήτηση του 59 θα πρέπει να ξεκινήσουμε από τη ρίζα. Κατευθυνόμαστε προς το δεξιό υπο-δένδρο αφού το 59 είναι μεγαλύτερο από τη ρίζα. Στη συνέχεια το 59 είναι μεγαλύτερο από το 49 και μικρότερο από το 63, συνεπώς κατευθυνόμαστε προς το δεξιό υπο-δένδρο

του 49. Αυτό το υποδένδρο έχει τριες τιμές, 54, 59, 61. Προφανώς, σε αυτό το σημείο θα εντοπίσουμε επιτυχώς το αναζητούμενο στοιχείο. Αν θέλουμε να αναζητήσουμε το 9, τότε από τη ρίζα θα κατευθυνθούμε προς το αριστερό υπο-δένδρο και στη συνέχεια αφού το 9 είναι μικρότερο από το 29 θα κατευθυνθούμε προς το αριστερό υπο-δένδρο επίσης. Τέλος, μια και 9<18 και το 18 δεν έχει κάποιο αριστερό υπο-δένδρο, αποφασίζουμε πως το 9 δεν είναι αποθηκευμένο στο δένδρο μας.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

Εισαγωγή στοιχείου: Όλες οι εισαγωγές γίνονται στο επίπεδο των φύλλων. Αρχικά, αναζητούμε τον κόμβο στον οποίο θα μπει το νέο στοιχείο και στη συνέχεια εξετάζουμε αν το φύλλο είναι γεμάτο ή όχι. Αν δεν είναι γεμάτο, απλά εισάγουμε το στοιχείο στη σωστή σειρά των κλειδιών. Αν είναι γεμάτο, τότε εισάγουμε το στοιχείο στη σωστή σειρά και διασπάμε σε δύο κόμβους το συγκεκριμένο φύλλο. Το κλειδί διάμεσος μετακινείται στον πατρικό κόμβο και εργαζόμαστε όπως και στην εισαγωγή ενός στοιχείου σε ένα φύλλο. Ακολουθεί ένα παράδειγμα εισαγωγής καθώς και οι αντίστοιχοι αλγόριθμοι.

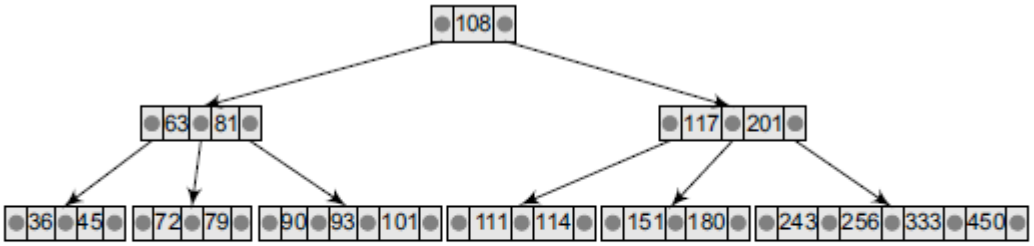


```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

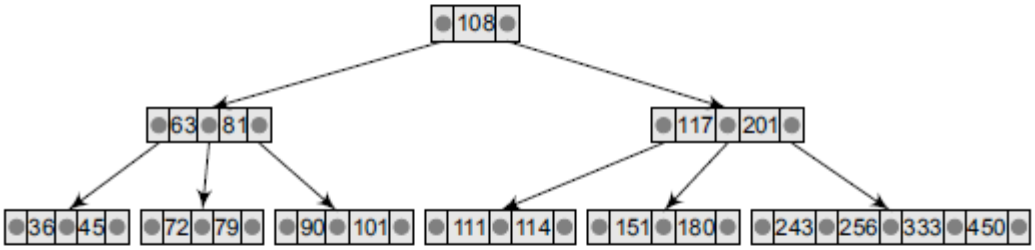
```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i − 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

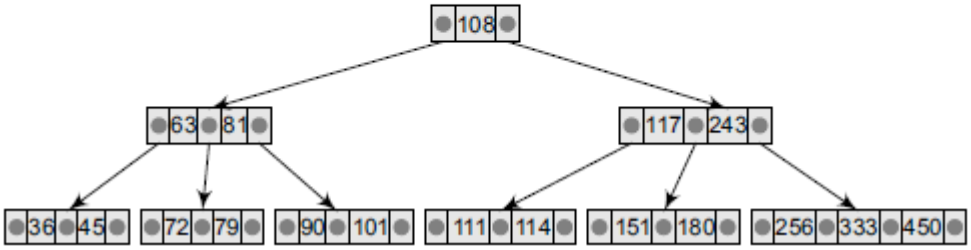
Διαγραφή στοιχείου: Υπάρχουν δύο περιπτώσεις διαγραφής: διαγραφή σε ένα κόμβο φύλλο και διαγραφή σε ένα εσωτερικό κόμβο. Γενικά, εντοπίζουμε το στοιχείο που πρόκειται να διαγραφεί. Αν ο κόμβος που περιέχει το στοιχείο προς διαγραφή περιλαμβάνει πλήθος στοιχείων πάνω από το όριο του m/2 τότε απλά διαγράφουμε το στοιχείο. Αν τα εναπομείναντα στοιχεία δεν είναι πάνω από το όριο του m/2 τότε παίρνουμε στοιχείο είτε από τον αριστερό ή το δεξιό κόμβο. Αν ο αριστερός κόμβος έχει στοιχεία πάνω από το όριο, τότε παίρνουμε το μεγαλύτερο κλειδί στον πατρικό κόμβο και κατεβάζουμε το ενδιάμεσο κλειδί του πατρικού κόμβου. Αν ο δεξιός κόμβος έχει στοιχεία πάνω από το όριο, τοποθετούμε το μικρότερο κλειδί στον πατρικό κόμβο και κατεβάζουμε από τον πατρικό το ενδιάμεσο κλειδί. Στην περίπτωση που και οι δύο κόμβοι-αδέρφια έχουν το μικρότερο δυνατό πλήθος στοιχείων, τότε συγχωνεύουμε σε νέο κόμβο στον οποίο τοποθετείται το ενδιάμεσο κλειδί του πατρικού κόμβου. Αν ο πατρικός κόμβος μείνει με λιγότερα στοιχεία από το όριο του δένδρου, τότε η προηγούμενη διαδικασία 'ανεβαίνει' προς τα πάνω και, έτσι, μειώνεται το ύψος του δένδρου. Ακολουθεί ένα παράδειγμα σχετικό με τη διαγραφή στοιχείων καθώς και ο αντίστοιχος αλγόριθμος.



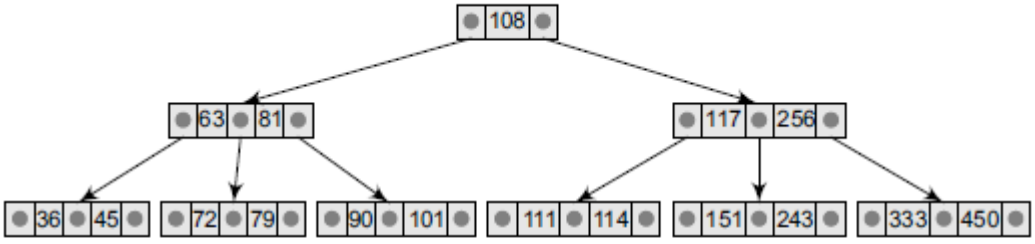
Step 1: Delete 93



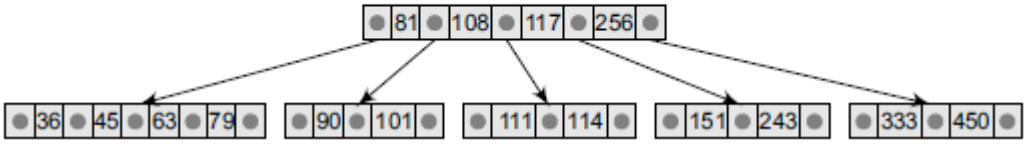
Step 2: Delete 201



Step 3: Delete 180



Step 4: Delete 72



```
B-Tree-Delete-Key(x, k)
    if not leaf[x] then
      y ← Preceding-Child(x)
      z ← Successor-Child(x)
      if n[y] > t − 1 then
            k′ ← Find-Predecessor-Key(k, x)
            Move-Key(k′, y, x)
            Move-Key(k, x, z)
            B-Tree-Delete-Key(k, z)
      else if n[z] > t − 1 then
            k′ ← Find-Successor-Key(k, x)
            Move-Key(k′, z, x)
            Move-Key(k, x, y)
            B-Tree-Delete-Key(k, y)
      else
            Move-Key(k, x, y)
            Merge-Nodes(y, z)
            B-Tree-Delete-Key(k,  y)

else (leaf node)
  y ← Preceding-Child(x)
  z ← Successor-Child(x)
  w ← root(x)
  v ← RootKey(x)
      if n[x] > t − 1 then Remove-Key(k, x)
      else if n[y] > t − 1 then
            k′ ← Find-Predecessor-Key(w, v)
            Move-Key(k′, y, w)
            k′ ← Find-Successor-Key(w, v)
            Move-Key(k′, w, x)
            B-Tree-Delete-Key(k, x)
      else if n[w] > t − 1 then
            k′ ← Find-Successor-Key(w, v)
            Move-Key(k′, z, w)
            k′ ← Find-Predecessor-Key(w, v)
            Move-Key(k′, w, x)
            B-Tree-Delete-Key(k, x)
```

else
  $s \leftarrow$ FIND-SIBLING(w)
  $w' \leftarrow \text{root}(w)$
    if $n[w'] = t - 1$ **then**
      MERGE-NODES(w', w)
      MERGE-NODES(w, s)
      B-TREE-DELETE-KEY(k, x)
    else
      MOVE-KEY(v, w, x)
      B-TREE-DELETE-KEY(k, x)

PRECEDING-CHILD(x) Returns the left child of key x.

MOVE-KEY(k, n_1, n_2) Moves key k from node n_1 to node n_2.

MERGE-NODES(n_1, n_2) Merges the keys of nodes n_1 and n_2 into a new node.

FIND-PREDECESSOR-KEY(n, k) Returns the key preceding key k in the child of node n.

REMOVE-KEY(k, n) Deletes key k from node n. n must be a leaf node.

Code 1

```
#include<stdlib.h>
#include<stdio.h>
#define M 5

struct node{
   int n; /* n < M No. of keys in node will always less than order of B tree */
   int keys[M-1]; /*array of keys*/
   struct node *p[M];   /* (n+1 pointers will be in use) */
}*root=NULL;

enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };

void insert(int key);
void display(struct node *root,int);
void DelNode(int x);
void search(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);

int main()
{
   int key;
   int choice;
   printf("Creation of B tree for node %d\n",M);
   while(1)
```

```
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Search\n");
        printf("4.Display\n");
        printf("5.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Enter the key : ");
                scanf("%d",&key);
                insert(key);
                break;
            case 2:
                printf("Enter the key : ");
                scanf("%d",&key);
                DelNode(key);
                break;
            case 3:
                printf("Enter the key : ");
                scanf("%d",&key);
                search(key);
                break;
            case 4:
                printf("Btree is :\n");
                display(root,0);
                break;
            case 5:
                exit(1);
            default:
                printf("Wrong choice\n");
                break;
        }/*End of switch*/
    }/*End of while*/
    return 0;
}/*End of main()*/

void insert(int key)
{
    struct node *newnode;
    int upKey;
    enum KeyStatus value;
    value = ins(root, key, &upKey, &newnode);
    if (value == Duplicate)
        printf("Key already available\n");
```

```
    if (value == InsertIt)
    {
        struct node *uproot = root;
        root=(struct node*) malloc(sizeof(struct node));
        root->n = 1;
        root->keys[0] = upKey;
        root->p[0] = uproot;
        root->p[1] = newnode;
    }/*End of if */
}/*End of insert()*/

enum KeyStatus ins(struct node *ptr, int key, int *upKey,struct node **newnode)
{
    struct node *newPtr, *lastPtr;
    int pos, i, n,splitPos;
    int newKey, lastKey;
    enum KeyStatus value;
    if (ptr == NULL)
    {
        *newnode = NULL;
        *upKey = key;
        return InsertIt;
    }
    n = ptr->n;
    pos = searchPos(key, ptr->keys, n);
    if (pos < n && key == ptr->keys[pos])
        return Duplicate;
    value = ins(ptr->p[pos], key, &newKey, &newPtr);
    if (value != InsertIt)
        return value;
    /*If keys in node is less than M-1 where M is order of B tree*/
    if (n < M - 1)
    {
        pos = searchPos(newKey, ptr->keys, n);
        /*Shifting the key and pointer right for inserting the new key*/
        for (i=n; i>pos; i--)
        {
            ptr->keys[i] = ptr->keys[i-1];
            ptr->p[i+1] = ptr->p[i];
        }
        /*Key is inserted at exact location*/
        ptr->keys[pos] = newKey;
        ptr->p[pos+1] = newPtr;
        ++ptr->n; /*incrementing the number of keys in node*/
        return Success;
    }/*End of if */
    /*If keys in nodes are maximum and position of node to be inserted is last*/
    if (pos == M - 1)
```

```
    {
        lastKey = newKey;
        lastPtr = newPtr;
    }
    else /*If keys in node are maximum and position of node to be inserted is not last*/
    {
        lastKey = ptr->keys[M-2];
        lastPtr = ptr->p[M-1];
        for (i=M-2; i>pos; i--)
        {
            ptr->keys[i] = ptr->keys[i-1];
            ptr->p[i+1] = ptr->p[i];
        }
        ptr->keys[pos] = newKey;
        ptr->p[pos+1] = newPtr;
    }
    splitPos = (M - 1)/2;
      (*upKey) = ptr->keys[splitPos];

    (*newnode)=(struct node*) malloc(sizeof(struct node));/*Right node after split*/
    ptr->n = splitPos; /*No. of keys for left splitted node*/
    (*newnode)->n = M-1-splitPos;/*No. of keys for right splitted node*/
    for (i=0; i < (*newnode)->n; i++)
    {
        (*newnode)->p[i] = ptr->p[i + splitPos + 1];
        if(i < (*newnode)->n - 1)
            (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
        else
            (*newnode)->keys[i] = lastKey;
    }
    (*newnode)->p[(*newnode)->n] = lastPtr;
    return InsertIt;
}/*End of ins()*/

void display(struct node *ptr, int blanks)
{
    if (ptr)
    {
        int i;
        for(i=1;i<=blanks;i++)
            printf(" ");
        for (i=0; i < ptr->n; i++)
            printf("%d ",ptr->keys[i]);
        printf("\n");
        for (i=0; i <= ptr->n; i++)
            display(ptr->p[i], blanks+10);
    }/*End of if*/
}/*End of display()*/
```

```
void search(int key)
{
   int pos, i, n;
   struct node *ptr = root;
   printf("Search path:\n");
   while (ptr)
   {
      n = ptr->n;
      for (i=0; i < ptr->n; i++)
         printf(" %d",ptr->keys[i]);
      printf("\n");
      pos = searchPos(key, ptr->keys, n);
      if (pos < n && key == ptr->keys[pos])
      {
         printf("Key %d found in position %d of last dispalyed node\n",key,i);
         return;
      }
      ptr = ptr->p[pos];
   }
   printf("Key %d is not available\n",key);
}/*End of search()*/

int searchPos(int key, int *key_arr, int n)
{
   int pos=0;
   while (pos < n && key > key_arr[pos])
      pos++;
   return pos;
}/*End of searchPos()*/

void DelNode(int key)
{
   struct node *uproot;
   enum KeyStatus value;
   value = del(root,key);
   switch (value)
   {
   case SearchFailure:
      printf("Key %d is not available\n",key);
      break;
   case LessKeys:
      uproot = root;
      root = root->p[0];
      free(uproot);
      break;
   }/*End of switch*/
}/*End of delnode()*/
```

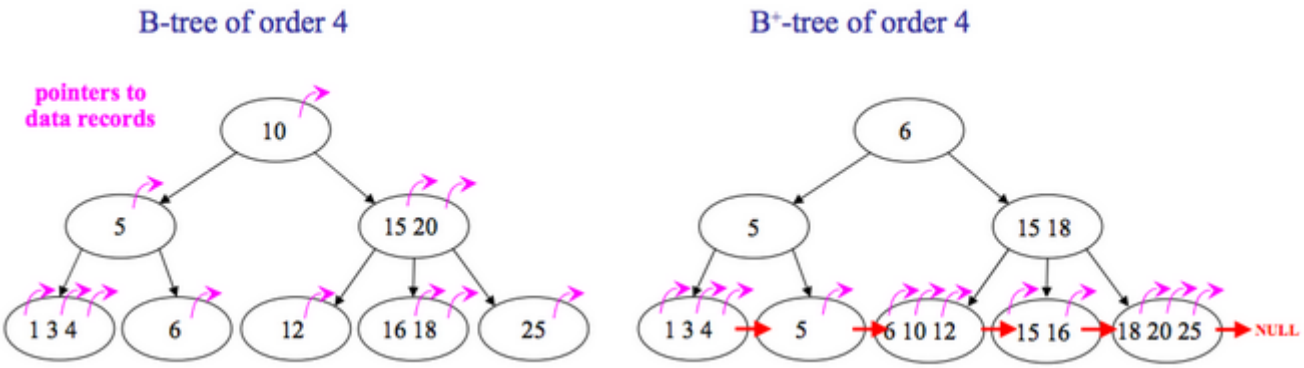
```
enum KeyStatus del(struct node *ptr, int key)
{
   int pos, i, pivot, n ,min;
   int *key_arr;
   enum KeyStatus value;
   struct node **p,*lptr,*rptr;

   if (ptr == NULL)
      return SearchFailure;
   /*Assigns values of node*/
   n=ptr->n;
   key_arr = ptr->keys;
   p = ptr->p;
   min = (M - 1)/2;/*Minimum number of keys*/

   pos = searchPos(key, key_arr, n);
   if (p[0] == NULL)
   {
      if (pos == n || key < key_arr[pos])
         return SearchFailure;
      /*Shift keys and pointers left*/
      for (i=pos+1; i < n; i++)
      {
         key_arr[i-1] = key_arr[i];
         p[i] = p[i+1];
      }
      return  --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;
   }/*End of if */

   if (pos < n && key == key_arr[pos])
   {
      struct node *qp = p[pos], *qp1;
      int nkey;
      while(1)
      {
         nkey = qp->n;
         qp1 = qp->p[nkey];
         if (qp1 == NULL)
            break;
         qp = qp1;
      }/*End of while*/
      key_arr[pos] = qp->keys[nkey-1];
      qp->keys[nkey - 1] = key;
   }/*End of if */
   value = del(p[pos], key);
   if (value != LessKeys)
      return value;
```

```
if (pos > 0 && p[pos-1]->n > min)
{
   pivot = pos - 1; /*pivot for left and right node*/
   lptr = p[pivot];
   rptr = p[pos];
   /*Assigns values for right node*/
   rptr->p[rptr->n + 1] = rptr->p[rptr->n];
   for (i=rptr->n; i>0; i--)
   {
      rptr->keys[i] = rptr->keys[i-1];
      rptr->p[i] = rptr->p[i-1];
   }
   rptr->n++;
   rptr->keys[0] = key_arr[pivot];
   rptr->p[0] = lptr->p[lptr->n];
   key_arr[pivot] = lptr->keys[--lptr->n];
   return Success;
}/*End of if */
if (pos<n && p[pos+1]->n > min)
{
   pivot = pos; /*pivot for left and right node*/
   lptr = p[pivot];
   rptr = p[pivot+1];
   /*Assigns values for left node*/
   lptr->keys[lptr->n] = key_arr[pivot];
   lptr->p[lptr->n + 1] = rptr->p[0];
   key_arr[pivot] = rptr->keys[0];
   lptr->n++;
   rptr->n--;
   for (i=0; i < rptr->n; i++)
   {
      rptr->keys[i] = rptr->keys[i+1];
      rptr->p[i] = rptr->p[i+1];
   }/*End of for*/
   rptr->p[rptr->n] = rptr->p[rptr->n + 1];
   return Success;
}/*End of if */

if(pos == n)
   pivot = pos-1;
else
   pivot = pos;

lptr = p[pivot];
rptr = p[pivot+1];
/*merge right node with left node*/
lptr->keys[lptr->n] = key_arr[pivot];
```

```
    lptr->p[lptr->n + 1] = rptr->p[0];
    for (i=0; i < rptr->n; i++)
    {
       lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
       lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
    }
    lptr->n = lptr->n + rptr->n +1;
    free(rptr); /*Remove right node*/
    for (i=pos+1; i < n; i++)
    {
       key_arr[i-1] = key_arr[i];
       p[i] = p[i+1];
    }
    return  --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}/*End of del()*/
```

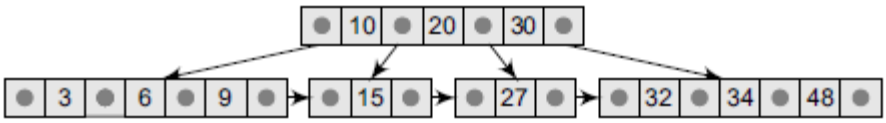
B+ Trees

Πρόκειται για ειδική περίπτωση των B-δένδρων. Ενώ ένα B-δένδρο μπορεί να αποθηκεύσει κλειδιά και εγγραφές σε εσωτερικούς κόμβους, τα B+ δένδρα αποθηκεύουν όλες τις εγγραφές στα φύλλα. Στους εσωτερικούς κόμβους αποθηκεύονται μόνο τα κλειδιά. Συχνά, τα φύλλα συνδεόνται μεταξύ τους μέσω μιας λίστας.

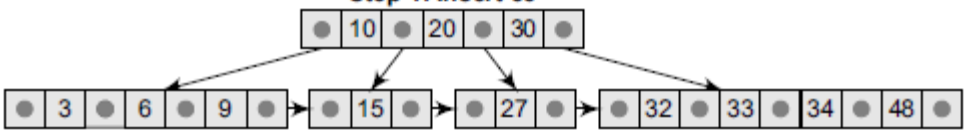


Το κύριο πλεονέκτημα των B+ δένδρων είναι ότι για τους εσωτερικούς κόμβους μπορούμε αποθηκεύσουμε περισσότερα κλειδιά στη μνήμη αφού δεν απαιτείται η αποθήκευση δεικτών προς τα δεδομένα (εγγραφές). Οι δείκτες προς τα δεδομένα αποθηκεύονται μόνο στα φύλλα ενός B+ δένδρου. Με αυτό τον τρόπο διευκολύνεται η αναζήτηση στοιχείων αφού θα γίνει μόνο στα φύλλα του δένδρου.

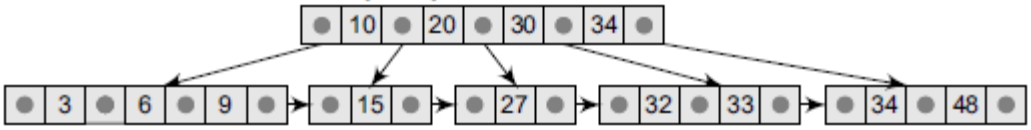
Εισαγωγή στοιχείου. Το νέο στοιχείο απλά προστίθεται σε κάποιο από τα φύλλα του δένδρου. Αν όμως, το φύλλο στο οποίο πρόκειται να γίνει η εισαγωγή είναι πλήρες από στοιχεία, τότε ο κόμβος διασπάται σε δύο. Αυτό προκαλεί, την εισαγωγή ενός νέου κλειδιού στον πατρικό κόμβο. Επιπρόσθετα, η προσθήκη κλειδιού στον πατρικό κόμβο μπορεί να προκαλέσει διάσπαση, κοκ.



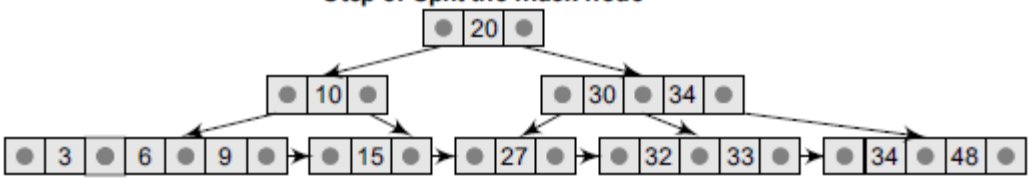
Step 1: Insert 33



Step 2: Split the leaf node



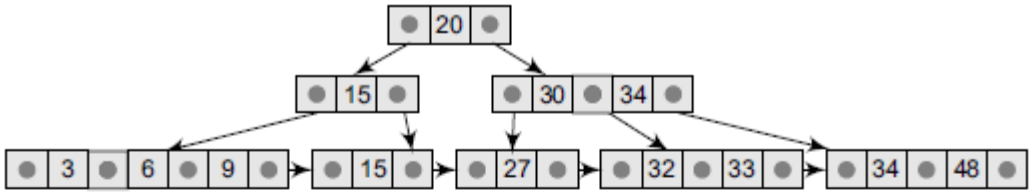
Step 3: Split the index node



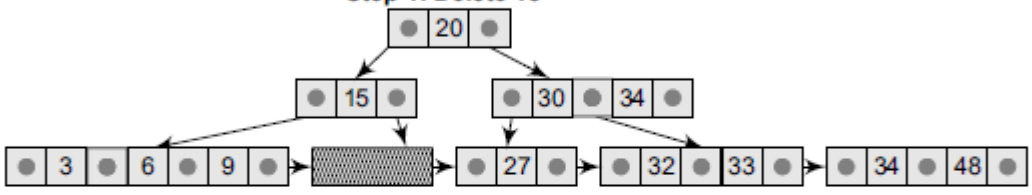
Ο αλγόριθμος εισαγωγής έχει ως εξής:

Step 1: Insert the new node as the leaf node. Step 2: If the leaf node overflows, split the node and copy the middle element to next index node. Step 3: If the index node overflows, split that node and move the middle element to next index page.

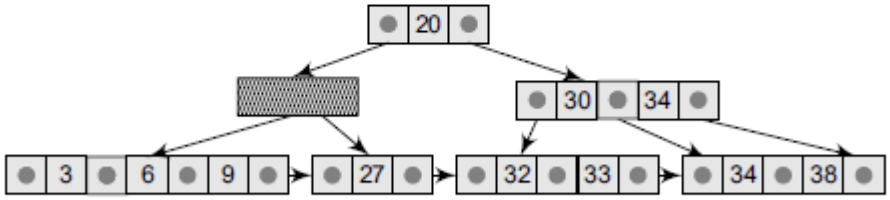
Διαγραφή στοιχείου. Αν η διαγραφή του στοιχείου αφήνει τον κόμβο άδειο, τότε οι γειτονικοί κόμβοι εξετάζονται ώστε να συγχωνευτούν με τον κόμβο που δεν είναι γεμάτος. Επίσης, το αντίστοιχο κλειδί θα πρέπει να διαγραφεί από τον πατρικό κόμβο που μπορεί να καταλήξει να είναι άδειος. Σε αυτή την περίπτωση, μπορεί να προκληθεί ένα 'κύμα' συγχωνεύσεων προς τη ρίζα του δένδρου.



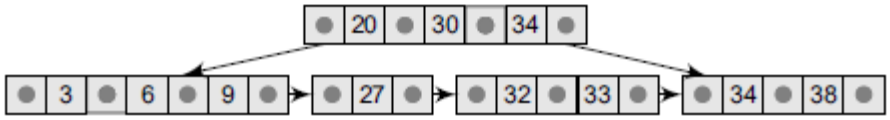
Step 1: Delete 15



Step 2: Leaf node underflows so merge with left sibling and remove key 15



Step 3: Now index node underflows, so merge with sibling and delete the node



Ο αλγόριθμος διαγραφής έχει ως εξής:

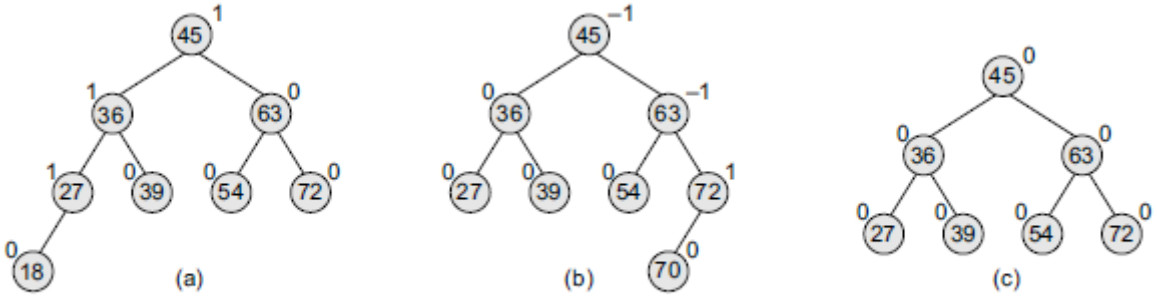
Step 1:	Delete the key and data from the leaves.
Step 2:	If the leaf node underflows, merge that node with the sibling and delete the key in between them.
Step 3:	If the index node underflows, merge that node with the sibling and move down the key in between them.

Code 2

http://www.amittai.com/prose/bpt.c

AVL Trees

Τα AVL δένδρα είναι **ισορροπημένα (balanced) δυαδικά δένδρα αναζήτησης** στα οποία τα δύο υπο-δένδρα (της ρίζας) έχουν διαφορά ύψους το πολύ 1. Το κύριο πλεονέκτημα τους είναι ο χρόνος εκτέλεσης κάποιας ενέργειας έχει πολυπλοκότητα O(logn) λόγω του γεγονότος ότι το ύψος του δένδρου είναι O(logn). Στα δένδρα αυτά αποθηκεύεται επιπρόσθετα ο **παράγοντας εξισορρόπησης (balance factor)** που ισούται με τη διαφορά του ύψους των δύο υπο-δένδρων (αριστερό - δεξιό). Στη ακόλουθη εικόνα βλέπουμε ένα παράδειγμα ενός AVL δένδρου όπου αναγράφονται οι παράγοντες εξισορρόπησης για κάθε κόμβο. Θα πρέπει να σημειωθεί ότι οι αποδεκτές τιμές για τον παράγοντα εξισορρόπησης είναι -1, 0 ή 1.

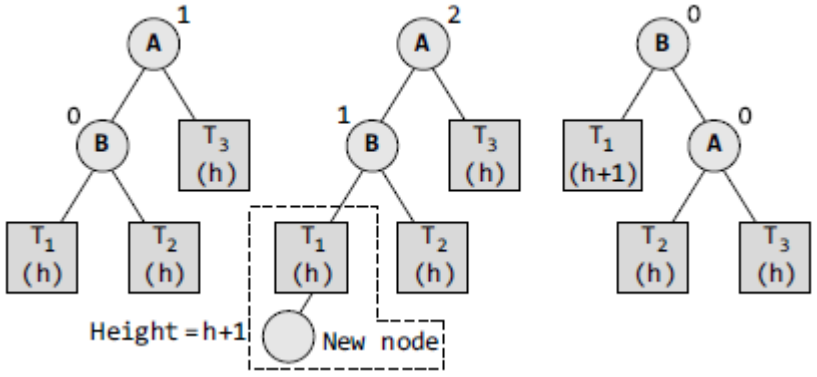


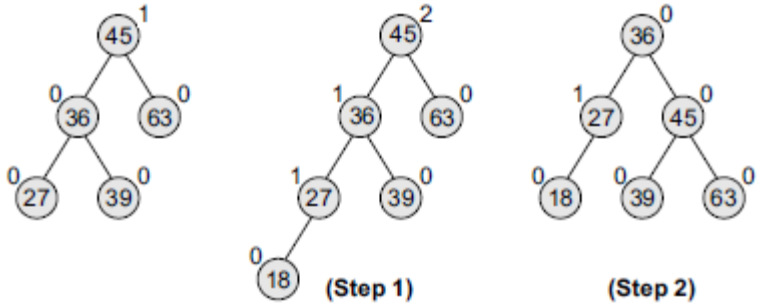
(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Αναζήτηση στοιχείου. Υλοποιείται ακριβώς με το ίδιο τρόπο όπως σε ένα δυαδικό δένδρο αναζήτησης.

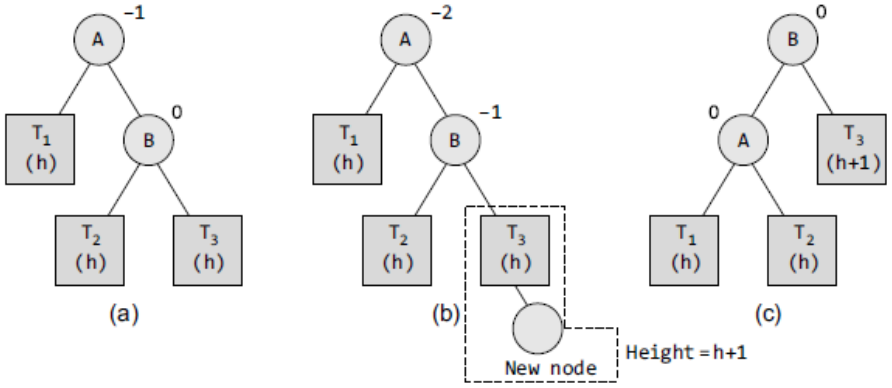
Εισαγωγή στοιχείου. Η εισαγωγή γίνεται επίσης όπως και στα δυαδικά δένδρα αναζήτησης. Ο νέος κόμβος έχει παράγοντα εξισορρόπησης ίσο με 0 αφού πρόκειται για φύλλο του δένδρου. Όμως, οι παράγοντες εξισορρόπησης των κόμβων που βρίσκονται στο μονοπάτι από τη ρίζα μέχρι το νέο κόμβο θα πρέπει να ενημερωθούν. Σε αυτή την περίπτωση, για να ισχύουν οι κανόνες του AVL δένδρου μπορεί να χρειαστεί η λεγόμενη **περιστροφή (rotation)**. Οι κατηγορίες περιστροφής είναι οι ακόλουθες:

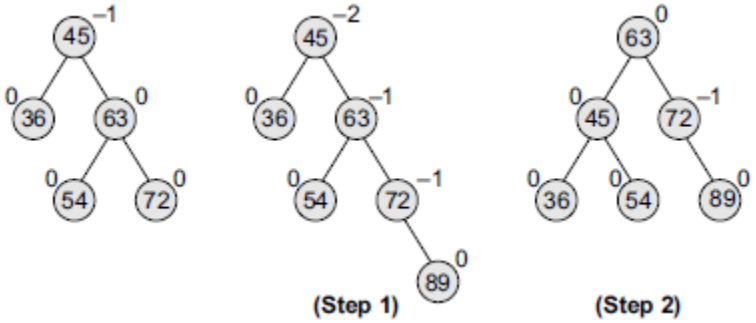
- **Περιστροφή LL**. Ο νέος κόμβος εισάγεται στο αριστερό υπο-δένδρο του αριστερού υπο-δένδρου του κρίσμου κόμβου. **Κρίσιμος είναι ο κόμβος που βρίσκεται πιο κοντά στον νεοεισερχόμενο κόμβο και έχει παράγοντα εξισορρόπησης διαφορετικό από τα -1, 0, 1**. Στην ακόλουθη εικόνα, ο κρίσιμος κόμβος είναι ο 72.



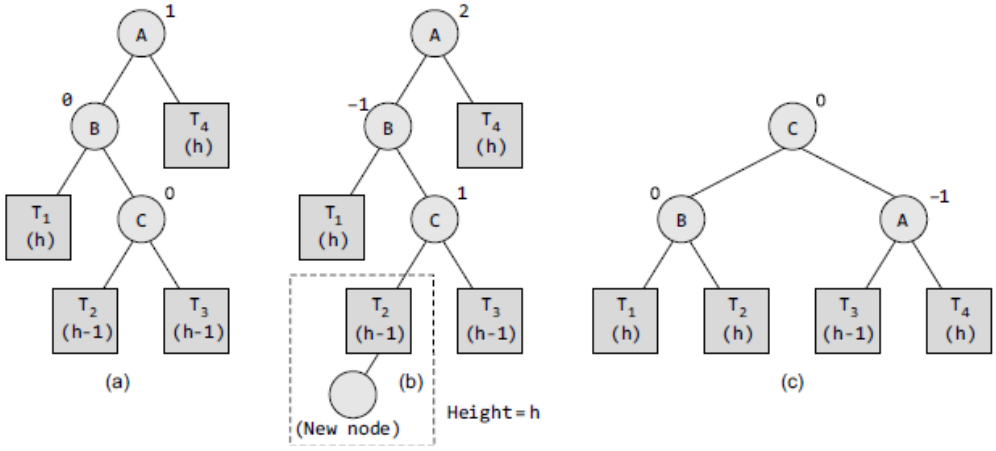


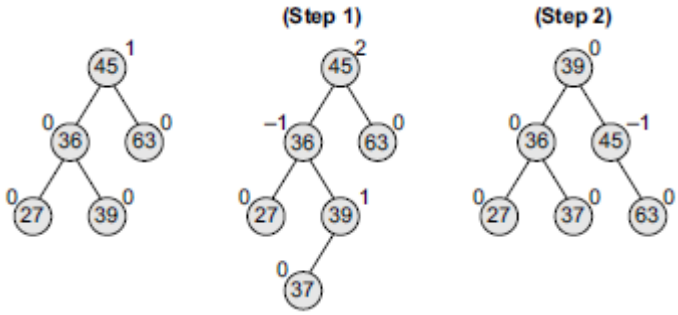
- **Περιστροφή RR**. Ο νέος κόμβος εισάγεται στο δεξιό υπο-δένδρο του δεξιού υπο-δένδρου του κρίσμου κόμβου.



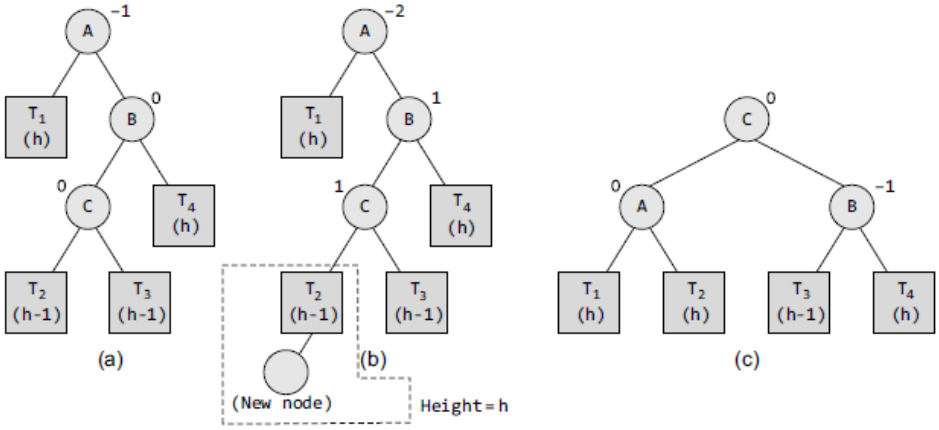


- **Περιστροφή LR**. Ο νέος κόμβος εισάγεται στο δεξιό υπο-δένδρο του αριστερού υπο-δένδρου του κρίσμου κόμβου.

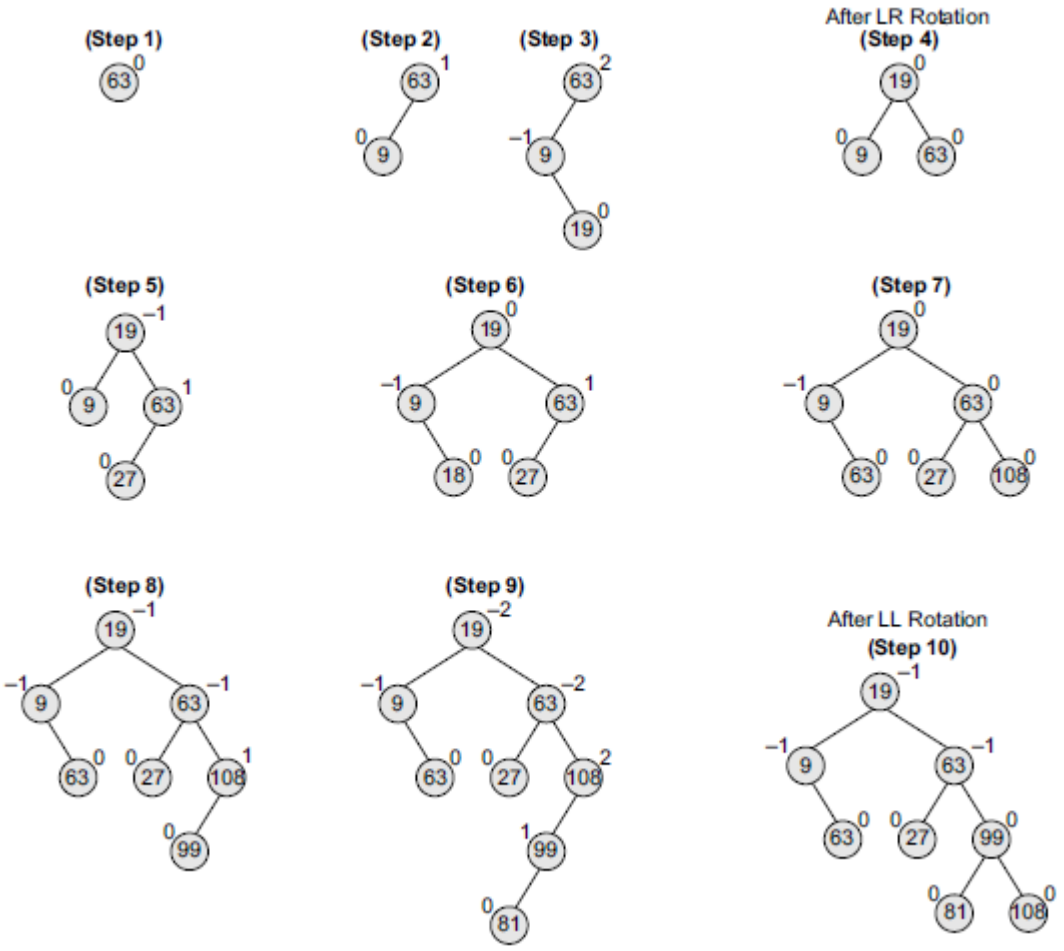




- **Περιστροφή RL**. Ο νέος κόμβος εισάγεται στο αριστερό υπο-δένδρο του δεξιού υπο-δένδρου του κρίσμου κόμβου.

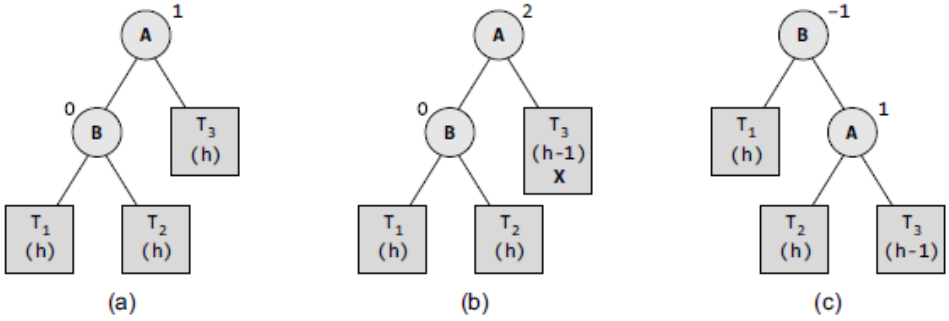


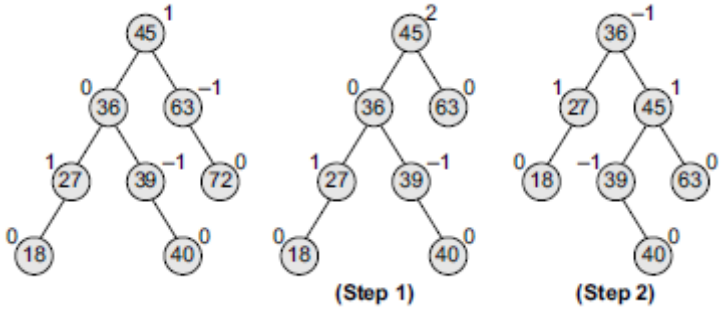
Ακολουθεί ένα παράδειγμα εισαγωγής των αριθμών: **63, 9, 19, 27, 18, 108, 99, 81**



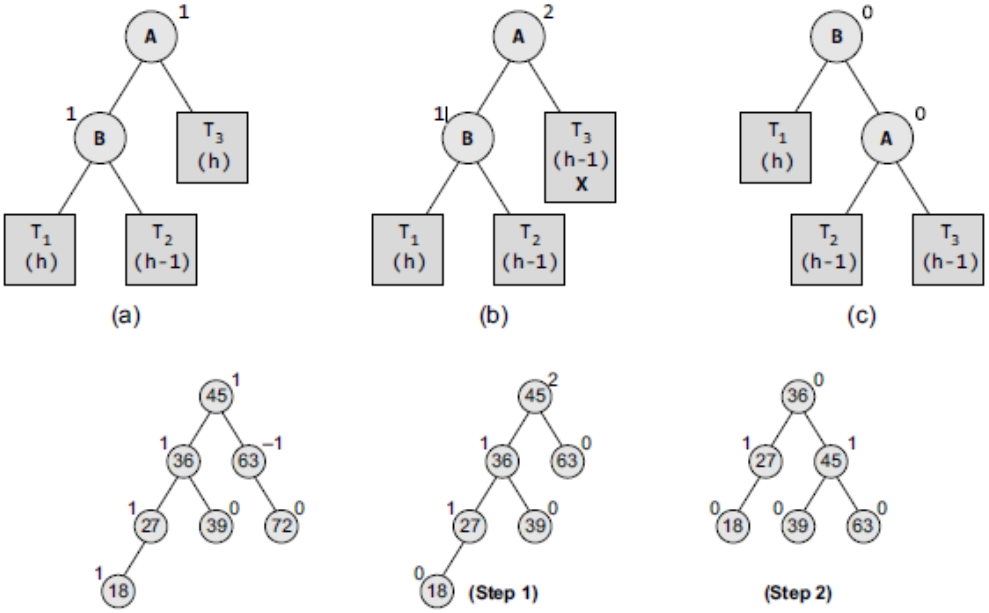
Διαγραφή στοιχείου. Επίσης, η διαγραφή είναι όμοια με τη διαγραφή στοιχείου σε ένα δυαδικό δένδρο. Το ζήτημα είναι ότι η διαγραφή μπορεί να διαταράξει τη μορφή του δένδρου, συνεπώς, θα πρέπει να εκτελέσουμε περιστροφές. Όμοια με την εισαγωγή αναζητούμε τον κρίσιμο κόμβο. Ο τύπος της περιστροφής εξαρτάται από το αν ο κόμβος διαγράφεται από το αριστερό ή το δεξιό υπο-δένδρο. Γενικά, έχουμε την L και την R περιστροφή. Η κάθε μια έχει τις ακόλουθες παραλλαγές: R0, R-1, R1, L0, L-1, L1.

- **Περιστροφή R0**. Εφαρμόζεται όταν ο κόμβος που βρίσκεται στο αριστερό υπο-δένδρο του κρίσιμου κόμβου έχει παράγοντα εξισορρόπησης ίσο με 0. Ο κόμβος που θα διαγραφεί βρίσκεται στο δεξιό υπο-δένδρο του κρίσιμου κόμβου.

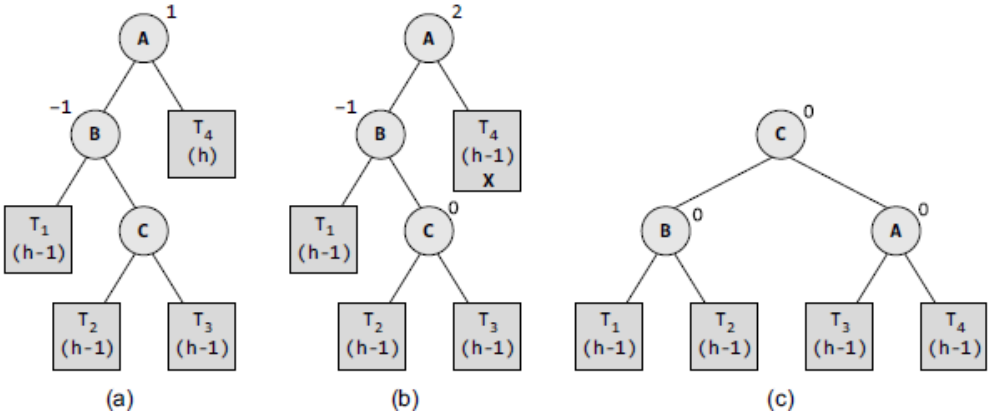


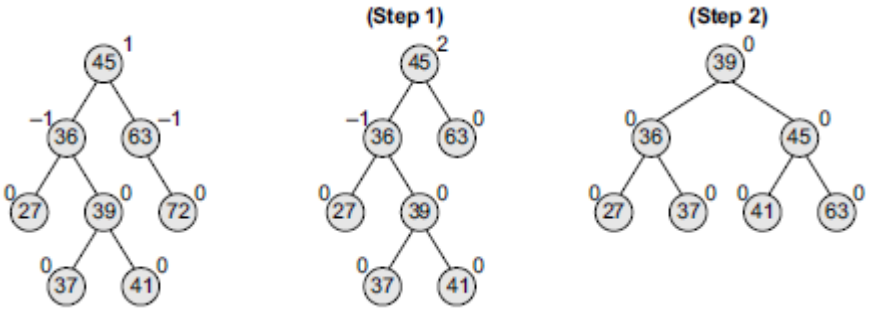


- **Περιστροφή R1**. Εφαρμόζεται όταν ο κόμβος που βρίσκεται στο αριστερό υπο-δένδρο του κρίσιμου κόμβου έχει παράγοντα εξισορρόπησης ίσο με 1. Ο κόμβος που θα διαγραφεί βρίσκεται στο δεξιό υπο-δένδρο του κρίσιμου κόμβου.



- **Περιστροφή R-1**. Εφαρμόζεται όταν ο κόμβος που βρίσκεται στο αριστερό υπο-δένδρο του κρίσιμου κόμβου έχει παράγοντα εξισορρόπησης ίσο με -1. Ο κόμβος που θα διαγραφεί βρίσκεται στο δεξιό υπο-δένδρο του κρίσιμου κόμβου.





Οι **L0, L-1, & L1** είναι συμμετρικές περιπτώσεις των προηγούμενων περιστροφών.

Code 3

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

typedef struct node
{
   int data;
   struct node *left,*right;
   int ht;
}node;

node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
int height( node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);

int main()
{
   node *root=NULL;
   int x,n,i,op;

   do
   {
      printf("\n1)Create:");
      printf("\n2)Insert:");
      printf("\n3)Delete:");
      printf("\n4)Print:");
      printf("\n5)Quit:");
      printf("\n\nEnter Your Choice:");
```

```
      scanf("%d",&op);

      switch(op)
      {
         case 1: printf("\nEnter no. of elements:");
               scanf("%d",&n);
               printf("\nEnter tree data:");
               root=NULL;
               for(i=0;i<n;i++)
               {
                  scanf("%d",&x);
                  root=insert(root,x);
               }
               break;

         case 2: printf("\nEnter a data:");
               scanf("%d",&x);
               root=insert(root,x);
               break;

         case 3: printf("\nEnter a data:");
               scanf("%d",&x);
               root=Delete(root,x);
               break;

         case 4: printf("\nPreorder sequence:\n");
               preorder(root);
               printf("\n\nInorder sequence:\n");
               inorder(root);
               printf("\n");
               break;
      }
   }while(op!=5);

   return 0;
}

node * insert(node *T,int x)
{
   if(T==NULL)
   {
      T=(node*)malloc(sizeof(node));
      T->data=x;
      T->left=NULL;
      T->right=NULL;
   }
   else
      if(x > T->data)       // insert in right subtree
```

```
    {
      T->right=insert(T->right,x);
      if(BF(T)==-2)
        if(x>T->right->data)
          T=RR(T);
        else
          T=RL(T);
    }
    else
      if(x<T->data)
      {
        T->left=insert(T->left,x);
        if(BF(T)==2)
          if(x < T->left->data)
            T=LL(T);
          else
            T=LR(T);
      }

    T->ht=height(T);

    return(T);
}

node * Delete(node *T,int x)
{
  node *p;

  if(T==NULL)
  {
    return NULL;
  }
  else
    if(x > T->data)       // insert in right subtree
    {
      T->right=Delete(T->right,x);
      if(BF(T)==2)
        if(BF(T->left)>=0)
          T=LL(T);
        else
          T=LR(T);
    }
    else
      if(x<T->data)
      {
        T->left=Delete(T->left,x);
        if(BF(T)==-2)    //Rebalance during windup
          if(BF(T->right)<=0)
```

```
                T=RR(T);
            else
                T=RL(T);
        }
        else
        {
          //data to be deleted is found
          if(T->right!=NULL)
          {   //delete its inorder succesor
            p=T->right;

            while(p->left!= NULL)
              p=p->left;

            T->data=p->data;
            T->right=Delete(T->right,p->data);

            if(BF(T)==2)//Rebalance during windup
              if(BF(T->left)>=0)
                T=LL(T);
              else
                T=LR(T);\
          }
          else
            return(T->left);
        }
  T->ht=height(T);
  return(T);
}

int height(node *T)
{
  int lh,rh;
  if(T==NULL)
    return(0);

  if(T->left==NULL)
    lh=0;
  else
    lh=1+T->left->ht;

  if(T->right==NULL)
    rh=0;
  else
    rh=1+T->right->ht;

  if(lh>rh)
    return(lh);
```

```
   return(rh);
}

node * rotateright(node *x)
{
   node *y;
   y=x->left;
   x->left=y->right;
   y->right=x;
   x->ht=height(x);
   y->ht=height(y);
   return(y);
}

node * rotateleft(node *x)
{
   node *y;
   y=x->right;
   x->right=y->left;
   y->left=x;
   x->ht=height(x);
   y->ht=height(y);

   return(y);
}

node * RR(node *T)
{
   T=rotateleft(T);
   return(T);
}

node * LL(node *T)
{
   T=rotateright(T);
   return(T);
}

node * LR(node *T)
{
   T->left=rotateleft(T->left);
   T=rotateright(T);

   return(T);
}

node * RL(node *T)
```

```
{
   T->right=rotateright(T->right);
   T=rotateleft(T);
   return(T);
}

int BF(node *T)
{
   int lh,rh;
   if(T==NULL)
      return(0);

   if(T->left==NULL)
      lh=0;
   else
      lh=1+T->left->ht;

   if(T->right==NULL)
      rh=0;
   else
      rh=1+T->right->ht;

   return(lh-rh);
}

void preorder(node *T)
{
   if(T!=NULL)
   {
      printf("%d(Bf=%d)",T->data,BF(T));
      preorder(T->left);
      preorder(T->right);
   }
}

void inorder(node *T)
{
   if(T!=NULL)
   {
      inorder(T->left);
      printf("%d(Bf=%d)",T->data,BF(T));
      inorder(T->right);
   }
}
```