

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΕΡΓΑΣΙΑ ΣΤΙΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΣΩΚΡΑΤΗΣ ΑΘΑΝΑΣΙΑΔΗΣ

ΑΕΜ 2547

ΙΟΥΝΙΟΣ 2020

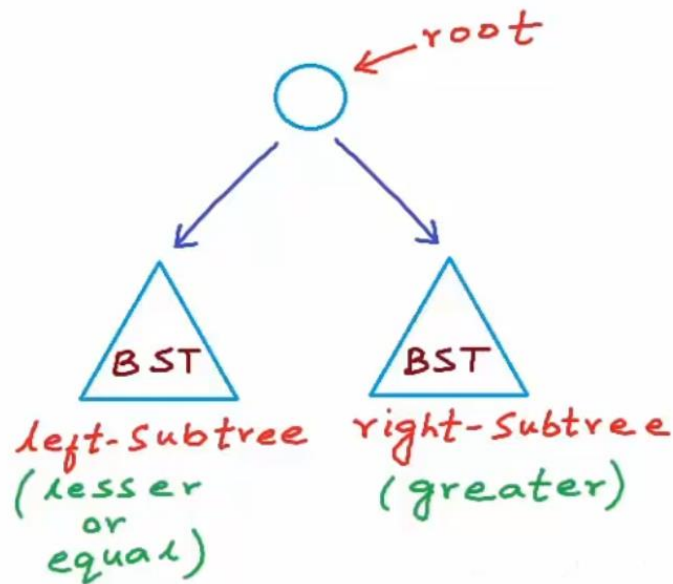
Contents

Binary Search Tree BST	3
Συνάρτηση Insert	5
Συνάρτηση Search.....	6
Συνάρτηση Delete	7
Συναρτήσεις printPostorder, printInorder, printPreorder	14
Λοιπές συναρτήσεις που χρησιμοποιήθηκαν για να διευκολυνθεί η ανάπτυξη του Project.....	15
Σχετικά με τα templates.....	15
AVL TREE	16
Κλάση AVLNode	18
Συνάρτηση height	18
Συνάρτηση GetNewNode.....	18
Συναρτήσεις rightRotate και leftRotate	19
Συνάρτηση balance	21
Συνάρτηση Insert και σενάρια Rotate μετά το Insert.....	21
Συνάρτηση Search.....	25
Συνάρτηση Delete	25
Συναρτήσεις printInorder, printPostorder, printPreorder	27
Λοιπές συναρτήσεις που χρησιμοποιήθηκαν για να διευκολυνθεί η ανάπτυξη του Project.....	28
Hash Table.....	29
Κλάση HashNode	29
Κλάση Hash Table	30
Hash Function	31
Συνάρτηση insertNode	31
Συνάρτηση Search.....	32
Συνάρτηση rehashing.....	33
Βοηθητικές συναρτήσεις	33
Main	34
Πειραματικά αποτελέσματα και σύγκριση των δομών.....	37

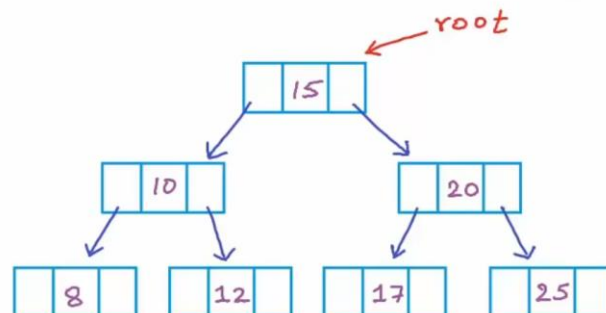
Binary Search Tree BST

Είναι μία δομή αποθήκευσης δεδομένων (δέντρικη δομή) στο οποίο χρησιμοποιούνται Δυναδικοί Κόμβοι, σε κάθε Κόμβο όλοι οι Κόμβοι που βρίσκονται στο αριστερό του υποδέντρο είναι μικρότεροι από αυτόν, ενώ όλοι οι κόμβοι που βρίσκονται στο δεξί υποδέντρο είναι μεγαλύτεροι (ή ίσοι στην συγκεκριμένη υλοποίηση) από αυτόν.

Ένα BST μπορεί να αναπαρασταθεί από την αναδρομική δομή που φαίνεται στο σχήμα:



Σε αυτήν την εργασία μπορούμε να φανταστούμε το BST ως μια διπλά Double Linked List σαν αυτή του σχήματος που ακολουθεί:



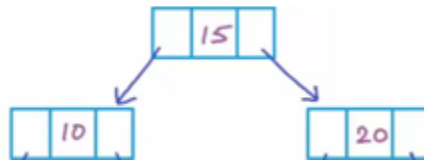
Αν κάθε Node του σχήματος είναι ένα Object, τότε θα πρέπει να το σχεδιάσουμε ως εξής:

```
class BstNode
{
private:
    Type data;
    int appearances;
    BstNode<Type>* left;
    BstNode<Type>* right;
```

Ως «data» ορίζουμε το πεδίο που αποθηκεύει το String μας (στο συγκεκριμένο παράδειγμα δουλεύουμε με String, στα γενικά πλαίσια της εργασίας όμως το BST και το AVL Tree έχουν σχεδιαστεί έτσι ώστε να δέχονται δεδομένα τύπου Int Float και String επειδή ο προγραμματιστής ήθελε να εξασκηθεί για το εργαστήριο που έδινε, η δομή Hash Table δέχεται μόνο δεδομένα τύπου String).

«Appearances» είναι ένα πεδίο όπου αποθηκεύουμε το πόσες φορές συναντάμε μια λέξη στο αρχείο που μετατρέπουμε σε BST.

Ως «left» και «right» ορίζουμε τους δύο pointers ακριβώς όπως φαίνεται στο σχήμα:



Οι pointers δείχνουν επίσης σε ένα αντικείμενο τύπου BstNode. Όπως και στην Double Linked List έχουμε 2 Nodes απλά στην BST δομή αντί να δημιουργούν μια γραμμική δομή την αντιμετωπίζουμε σαν δενδρική δομή.

Στην εργασία όλα τα Nodes θέλουμε να δημιουργούνται δυναμικά στο heap. Για τον λόγο αυτό χρησιμοποιήθηκε ο τελεστής new (θα μπορούσαμε να χρησιμοποιήσουμε και την συνάρτηση malloc που γνωρίζουμε παραδοσιακά από την C). Όμως όλα τα αντικείμενα που δημιουργούνται στο heap δεν μπορούν να έχουν όνομα (identifier) για αυτό τον λόγο χρησιμοποιούνται pointers (ο τελεστής new χρησιμοποιεί Pointer) και επομένως η συνάρτηση που ακολουθεί δουλεύει τέλεια και μπορούμε να την θεωρήσουμε ως Constructor (δεν είναι όμως) ενός Node. Έτσι δημιουργούμε μια κλάση την GetNewNode.

```

template <class Type>
BstNode<Type>* BstNode<Type>::GetNewNode (Type da
{
    //create a Node
    //return address of this new Node
    BstNode<Type>* newNode = new BstNode(); //-
    newNode->appearances = 1;
    newNode->data=data;
    newNode->left = newNode->right = NULL; //ini
    return newNode;
}

```

Επομένως αφού κατανοήσαμε την γενική ιδέα, καταλαβαίνουμε ότι για να έχουμε πρόσβαση στο BST το μόνο που χρειαζόμαστε και κρατάμε είναι η διεύθυνση του root Node (αφού από αυτόν μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε Node πηγαίνοντας «δεξιά» και «αριστερά» καθώς διασχίζουμε το δέντρο μας).

```

int main()
{
    BstNode<string>* root = NULL;
}

```

Το ότι το θέτουμε NULL σημαίνει ότι αρχικά το Δέντρο μας είναι άδειο.

Συνάρτηση Insert

```

//Insert Data in BST
template <class Type>
BstNode<Type>* BstNode<Type>::Insert (BstNode* root, Type data)
{
    if (root==NULL)
    {
        root = GetNewNode(data); //if tree is empty we create a
    }
    else if (data == (root->data)) //its the same string appeari
    {
        Type BstNode::data
        root->appearances++;
    }
    else if (data<(root->data) )
    {
        root->left = Insert (root->left,data); //we add the Node
    }
    else
    {
        root->right = Insert (root->right,data);
    }
    return root;
}

```

Είναι η μέθοδος που χρησιμοποιούμε όταν θέλουμε να προσθέσουμε έναν νέο κόμβο στο δέντρο μας. Ως ορίσματα δέχεται έναν Pointer ο οποίος είναι ο root Node και το data το οποίο θέλουμε να εισάγουμε στο δέντρο.

Αρχικά αν το δέντρο μας είναι κενό, τότε απλά φτιάχνουμε ένα BstNode και το βάζουμε να είναι το νέο μας root Node.

Αν το data που πάμε να εισάγουμε υπάρχει ήδη στο δέντρο τότε απλά αυξάνουμε κατά ένα το appearance του που σημαίνει ότι το έχουμε 2 φορές (λογική που ακολουθήθηκε αυθαίρετα αφού μας το επέτρεψε η εκφώνηση της εργασίας)

Αλλιώς αν το Tree δεν είναι άδειο και αν εισάγουμε ένα τελείως καινούργιο data το συγκρίνουμε με το root Node μας, αν είναι μικρότερο τότε πάμε να το εισάγουμε αναδρομικά στο αριστερό υποδέντρο αν είναι μεγαλύτερο η ίσο τότε πάμε να το εισάγουμε αναδρομικά στο δεξί υποδέντρο. Η αναδρομή χρησιμοποιείται γιατί όλες τις δομές δέντρων τις επεξεργαζόμαστε αναδρομικά για προφανείς λόγους.

Στο τέλος κάθε αναδρομής επιστρέφεται το root node.

Συνάρτηση Search

```
int BstNode<Type>::Search(BstNode<Type>* root, Type data) //
{
    if(root == NULL)
    {
        cout<<"Not Found"<<endl;
        return false;
    }
    else if(root->data == data)
    {
        cout<<root->data<<" "<<"Found: ";
        return root->appearances;
    }
    //else we search on the left subtree or in the right sub
    else if(data<root->data)
    {
        return Search(root->left, data);
    }
    else
    {
        return Search(root->right, data);
    }
}
```

Στη συνάρτηση αυτή αναζητούμε το data μέσα στην δομή BST. Αρχικά ελέγχουμε αν το δέντρο μας είναι άδειο, αν είναι άδειο προφανώς η Search θα επιστρέψει false γιατί δεν θα βρούμε το στοιχείο που αναζητούμε μας μέσα στην δομή.

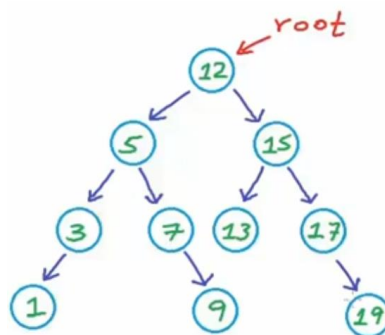
Αν η δομή μας δεν είναι κενή τότε συγκρίνουμε το data που αναζητούμε με το data που υπάρχει στον root κόμβο αν αυτό είναι ίδιο τότε σημαίνει ότι βρήκαμε το στοιχείο που αναζητούσαμε και επιστρέφουμε το πόσες φορές το βρήκαμε ενώ τυπώνουμε και το μήνυμα "Found".

Αν δεν είναι ίσο, τότε αν είναι μικρότερο θα ψάξουμε αναδρομικά στο αριστερό υποδέντρο θέτοντας ως root το αριστερό παιδί, ενώ αν είναι μεγαλύτερο ή ίσο θα ψάξουμε αναδρομικά στο δεξί υποδέντρο της δομής.

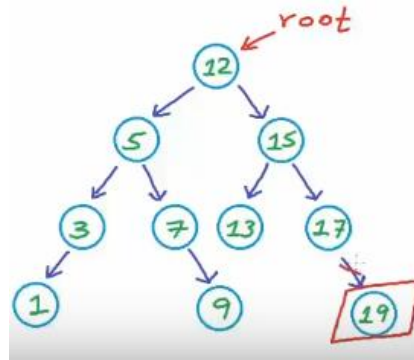
Συνάρτηση Delete

Όπως και στις περισσότερες Δομές Δεδομένων η συνάρτηση Delete έχει τις ιδιαιτερότητες της. Αυτό συμβαίνει επειδή όταν διαγράφουμε έναν Κόμβο από το BST τότε θέλουμε η δομή μας να παραμείνει BST δηλαδή για κάθε κόμβο όλες οι τιμές στο αριστερό υποδέντρο του κόμβου να είναι μικρότερες από αυτόν και όλες οι τιμές στο δεξί υποδέντρο να είναι μεγαλύτερες ή ίσες από αυτόν και αυτό να ισχύει για όλους τους κόμβους η πιο σωστά να συνεχίσει να ισχύει και μετά την διαγραφή κάποιου Κόμβου. Για να ισχύσει αυτό μετά την διαγραφή θα πρέπει να κάνουμε κάποιες τακτοποιήσεις για να διευθετήσουμε ότι η δομή μας παραμένει BST.

Αρχικά αν έχουμε το εξής δέντρο:

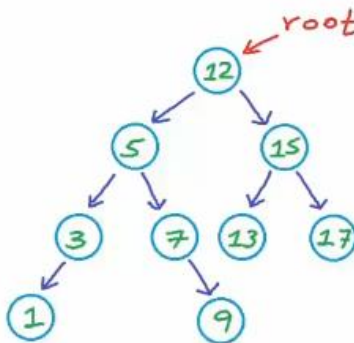


Και θέλουμε να διαγράψουμε από αυτό τον Κόμβο με την τιμή 19

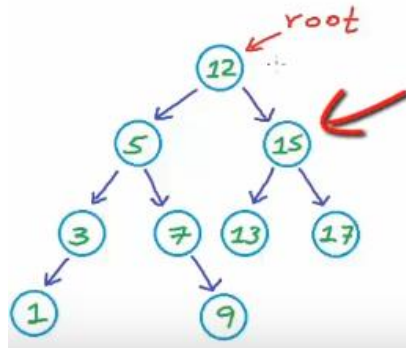


Τότε αρκεί να διαγράψουμε τον Pointer που οδηγεί σε αυτόν από τον γονέα του όπως φαίνεται στο σχήμα (τον θέτουμε σε NULL) και ένα πολύ σημαντικό κομμάτι είναι να ελευθερώσουμε την μνήμη που είχαμε δεσμεύσει για τον κόμβο 19.

Φυσικά αυτό το σενάριο είναι το εύκολο σενάριο όταν διαγράφουμε έναν κόμβο από μια δομή BST, και αυτό επειδή ο κόμβος με την τιμή 19 είναι κόμβος φύλλο και όταν διαγράφουμε έναν Κόμβο που είναι Leaf Node σε μία BST δομή, τότε είμαστε βέβαιοι ότι η δομή μας θα συνεχίσει να είναι BST αφού αυτός ο κόμβος δεν έχει ούτε δεξί ούτε αριστερό υποδέντρο και δεν τίθεται κανένα θέμα.



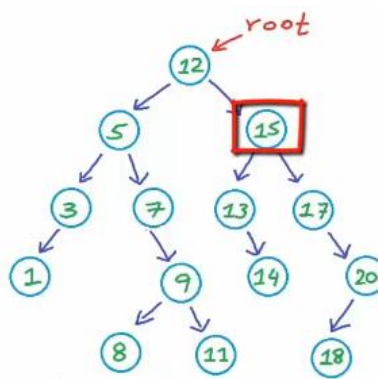
Τα πράγματα όμως είναι περισσότερο πολύπλοκα όταν θέλουμε να διαγράψουμε έναν κόμβο που δεν είναι Leaf Node. Για παράδειγμα αν θέλουμε να διαγράψουμε τον κόμβο που έχει τιμή 15



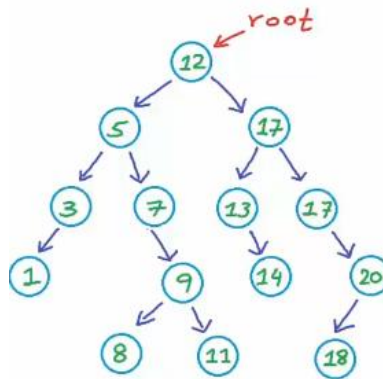
Δεν μπορούμε να διαγράψουμε απλά τον Pointer που οδηγεί σε αυτόν από τον γονέα του (κόμβος 12) γιατί έτσι διαγράφουμε όλο το δεξί υποδέντρο του κόμβου γονέα (στο συγκεκριμένο παράδειγμα). Θέλουμε να βεβαιωθούμε ότι όλοι οι κόμβοι του δεξιού υποδέντρου θα παραμείνουν διαθέσιμοι και θα είναι μέρος του BST και μετά την διαγραφή του Κόμβου με τιμή 15.

Σε όλες τις περιπτώσεις που ο κόμβος που διαγράφουμε δεν είναι Leaf Node τότε θα έχει 1 ή το πολύ 2 παιδιά. Αν έχει μόνο ένα παιδί τότε απλά συνδέουμε τον γονέα του κατευθείαν με το παιδί και φυσικά απελευθερώνουμε την μνήμη που δέσμευε ο κόμβος που διαγράψαμε. Για παράδειγμα αν θέλαμε στο σχήμα μας να διαγράψουμε τον κόμβο με την τιμή 3 τότε απλά θα συνδέαμε τον γονέα του (τον κόμβο με την τιμή 5) απευθείας στο μοναδικό παιδί του κόμβου που διαγράφουμε, δηλαδή στον κόμβο με την τιμή 1. Η δομή μας συνεχίζει να είναι BST και μετά την διαγραφή ενός κόμβου που έχει μόνο ένα παιδί όπως ακριβώς περιγράψαμε.

Στην περίπτωση όμως που ο κόμβος που διαγράφουμε έχει 2 παιδιά τότε τα πράγματα γίνονται ακόμη πιο σύνθετα. Για παράδειγμα αν θέλουμε να διαγράψουμε τον κόμβο με τιμή 15.



Αυτό που κάνουμε είναι ότι δεν διαγράφουμε ακριβώς τον κόμβο, άλλα ψάχνουμε τον κόμβο με την ελάχιστη τιμή στο δεξί υποδέντρο (που στην περίπτωση μας είναι το 17, και βάζουμε αυτήν την τιμή στον κόμβο που προηγουμένως ήταν η τιμή 15.

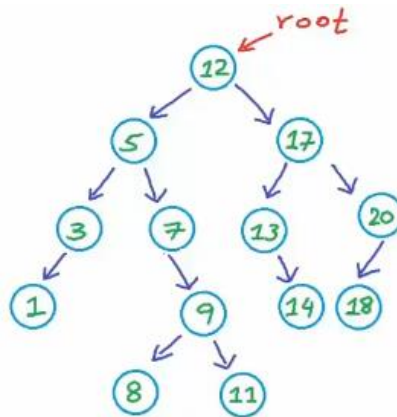


Ψάχνουμε το ελάχιστο του δεξιού υποδέντρου για τον λόγο ότι

1. Αυτό θα είναι σίγουρα μεγαλύτερο (η ίσο) από αυτό που θέλαμε να διαγράψουμε, επομένως αφού το τοποθετήσουμε στο σημείο που είχαμε τον κόμβο που θέλαμε να διαγράψουμε ότι είναι στο δεξί υποδέντρο του θα είναι μεγαλύτερο η ίσο από αυτό (αφού επιλέξαμε το ελάχιστο από αυτά) και
2. Αφού σίγουρα όλοι οι κόμβοι στο αριστερό του υποδέντρο είναι μικρότεροι από τον κόμβο που θέλαμε να διαγράψουμε (το 15) τότε θα είναι σίγουρα μικρότεροι από κάποιον κόμβο που είναι μεγαλύτερος (η ίσος) από αυτόν (το 17 είναι μεγαλύτερο από όλους τους κόμβους του αριστερού υποδέντρου).

Στο συγκεκριμένο παράδειγμα ήταν πολύ ιδανικό που βρέθηκε το 17 ως τον υποψήφιο κόμβο για να αντικαταστήσει τον 15 είχε μόνο ένα παιδί. Και αναλύθηκε προηγουμένως πως διαγράφουμε έναν κόμβο που έχει μόνο ένα παιδί, και ακολουθούμε αυτήν την διαδικασία για να διαγράψουμε το σημείο που είχαμε προηγουμένως τον κόμβο 17.

Το αποτέλεσμα είναι το εξής.



Μια εναλλακτική είναι αντί να επιλέξουμε το ελάχιστο του δεξιού υποδέντρου είναι να επιλέξουμε το μέγιστο του αριστερού υποδέντρου. Η εναλλακτική δεν αναλύεται άλλα μπορεί κανείς να φανταστεί πως προσεγγίζεται.

Είναι φανερό ότι με τον τρόπο που αναλύσαμε την διαγραφή ενός κόμβου είμαστε αναγκασμένοι να χρησιμοποιήσουμε για μια ακόμα φορά αναδρομή. Γιατί; Γιατί για να διαγράψουμε τον κόμβο 15 είπαμε ότι θα τον αντικαταστήσουμε με τον κόμβο που έχει την ελάχιστη τιμή στο δεξί υποδέντρο και αν ο κόμβος αυτός έχει επίσης 2 παιδιά θα ακολουθήσουμε την ίδια διαδικασία μέχρι να καταλήξουμε σε Leaf Node η μέχρι να καταλήξουμε σε κόμβο που έχει μόνο ένα παιδί.

```

BstNode<Type>* BstNode<Type>::Delete(BstNode* root, Type data)
{
    if(root == NULL)
        return root;
    else if(data<(root->data) )
        root->left = Delete(root->left, data);
    else if (data>(root->data))
        root->right = Delete(root->right, data);

    else
    {
        //it has more than one appearances we remove 1 appearance not the wh
        if(root->appearances > 1)
        {
            root->appearances--;
        }
        //Node has no child
        else if(root->left == NULL && root->right == NULL)
        {
            delete root;
            root = NULL;
        }
        //Node has 1 child
        else if(root->left == NULL)
        {
            BstNode *temp = root;
            root = root->right;
            delete temp;
        }
        else if(root->right == NULL)
        {
            BstNode *temp = root;
            root = root->left;
            delete temp;
        }
        //Node has 2 children
        else
        {
            BstNode *temp = findMin(root->right);
            root->data = temp->data;
            root->right = Delete(root->right, temp->data);
        }
    }
    return root;
}

```

Η συνάρτηση μας δέχεται το root Κόμβο που είναι και αυτός που έχουμε «αποθηκεύσει» και έχουμε πρόσβαση από την main. Μετά με τον ίδιο τρόπο που υλοποιήθηκε η Search ψάχνουμε να βρούμε τον κόμβο που θέλουμε να διαγράψουμε. Για προφανείς λόγους δεν χρησιμοποιήθηκε η συνάρτηση Search αφού αυτή έχει προγραμματιστεί να επιστρέφει τις εμφανίσεις του κόμβου που ψάχνουμε (δηλαδή το πόσες φορές υπάρχει αυτή η λέξη στην Δομή μας) η να επιστρέφει μια false τιμή σε περίπτωση που ο κόμβος δεν υπάρχει.

Στη συνέχεια στην else της γραμμής 122 ελέγχουμε αν οι εμφανίσεις της τιμής που θέλουμε να διαγράψουμε είναι περισσότερες από μία. Αν ισχύει αυτό τότε μειώνουμε τον δείκτη appearances κατά 1. Αν είναι 1 τότε ψάχνουμε να βρούμε σε ποια περίπτωση είμαστε από τις προηγούμενες.

1. Ο κόμβος δεν έχει παιδιά . Σημαίνει ότι ο κόμβος που θέλουμε να διαγράψουμε είναι Leaf Node και τότε απλά κάνουμε delete τον κόμβο που διαγράφουμε τον οποίο έχουμε εντοπίσει αναδρομικά και άρα είναι ο root , κάνοντας delete τον κόμβο με την εντολή delete root αποδεσμεύουμε την μνήμη που είχε δεσμευτεί για τον κόμβο αυτόν. Ο κόμβος root έχει ακόμα την διεύθυνση που είχε πριν όμως (ο pointer) για αυτόν τον θέτουμε ως NULL.
2. Η περίπτωση 2 είναι ο κόμβος να έχει 1 παιδί τότε αυτό που κάνουμε είναι ότι φτιάχνουμε έναν pointer για να αποθηκεύσουμε την διεύθυνση του κόμβου που θέλουμε να διαγράψουμε. Και μεταφέρουμε στον root τον κόμβο που είχαμε δεξιά/αριστερά του root. (που είναι και το μοναδικό του παιδί) ως τελευταίο βήμα φυσικά αποδεσμεύουμε την μνήμη που δέσμευε ο κόμβος που θέλαμε να διαγράψουμε, και αυτός είναι και ο λόγος που φτιάξαμε τον temp pointer. Υπάρχουν 2 else-if εδώ ανάλογα με το αν το μοναδικό παιδί του κόμβου που διαγράφουμε είναι αριστερό η δεξί παιδί.
3. Είναι η περίπτωση που ο κόμβος που θέλουμε να διαγράψουμε έχει 2 παιδιά. Σε αυτήν την περίπτωση ψάχνουμε τον ελάχιστο κόμβο του δεξιού υποδέντρου με την βοηθητική συνάρτηση findMin όπου ως όρισμα παίρνει το δεξί παιδί του κόμβου που διαγράφουμε.

```
template <class Type>
BstNode<Type>* BstNode<Type>::findMin(BstNode* root)
{
    while(root->left != NULL)
        root = root->left;
    return root;
}
```

Η συνάρτηση findMin πηγαίνει και βρίσκει το τέρμα αριστερά φύλλο του του δεξιού υποδέντρου γιατί θα είναι και το ελάχιστο του σύμφωνα με τον κανόνα με τον οποίο τοποθετούμε στοιχεία στο BST.

Στη συνέχεια αφού αποθηκεύσουμε τον κόμβο αυτόν στον temp pointer, αντικαθιστούμε τον κόμβο που θέλαμε να διαγράψουμε με τον κόμβο που βρήκαμε από την συνάρτηση findMin. Και διαγράφουμε αυτόν τον κόμβο

που βρήκαμε από την συνάρτηση findMin που είναι στο δεξί υποδέντρο του κόμβου root.

Συναρτήσεις printPostorder, printInorder, printPreorder

```
//Postorder
template <class Type>
void BstNode<Type>::printPostorder(BstNode* root)
{
    if (root == NULL)
        return;
    printPostorder(root->left);
    printPostorder(root->right);
    cout << root->data << " ";
}

//Inorder
template <class Type>
void BstNode<Type>::printInorder(BstNode* root)
{
    if (root == NULL)
        return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

//Preorder
template <class Type>
void BstNode<Type>::printPreorder(BstNode* root)
{
    if (root == NULL)
        return;
    cout << root->data << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}
```

Οι συναρτήσεις δουλεύουν αναδρομικά.

1. Η postorder πρώτα κάνει traverse στο αριστερό υποδέντρο μετά στο δεξί και στο τέλος επισκέπτεται την ρίζα. Βάση αυτής της λογικής είναι η συνάρτηση που θα καλούσαμε για να διαγράψουμε όλο το υποδέντρο για να επισκεφθούμε τους κόμβους με σειρά που μπορούμε να τους διαγράψουμε κιόλας.
2. Η inorder επισκέπτεται το αριστερό υποδέντρο μετά την ρίζα και μετά το δεξί υποδέντρο, και εδώ ακολουθούμε τον αναδρομικό κανόνα για να επισκεφθούμε όλους τους κόμβους.
3. Preorder. Επισκεπτόμαστε πρώτα τον κόμβο, μετά κάνουμε traverse στο αριστερό υποδέντρο και μετά στο δεξί. Αυτή η μέθοδος διάτρεξης

χρησιμοποιείται σε περίπτωση που θέλουμε να κάνουμε ένα αντίγραφο του δέντρου.

Αν προσέξουμε τις συναρτήσεις που γράψαμε σε c++ η μόνη διαφορά που έχουν είναι σε ποιο σημείο βάζουμε την εντολή `cout << root->data << " ";` .

Λοιπές συναρτήσεις που χρησιμοποιήθηκαν για να διευκολυνθεί η ανάπτυξη του Project

Υπάρχουν ακόμα 3 συναρτήσεις που χρησιμοποιήθηκαν για να διευκολυνθεί η ανάπτυξη του Project. Θέλω να σημειώσω ότι αυτές οι συναρτήσεις δεν υλοποιήθηκαν από εμένα αλλά τις πήρα έτοιμες κάνοντας κάποιες αλλαγές. Οι συναρτήσεις «print2DUtil» και «print2D» χρησιμοποιήθηκαν για να οπτικοποιηθεί το BST και να δω ότι το αναπτύσσω σωστά (είναι αποδοτικό το print μόνο με μικρά δέντρα) και η συνάρτηση «getSize» χρησιμοποιήθηκε για να συγκρίνω το BST με το AVL Tree που αναπτύσσεται αργότερα, για να δω αν είναι σωστό.

Άφησα τις συναρτήσεις και τις στέλνω μαζί με το Project για λόγους πληρότητας. Στον κώδικα υπάρχει σε σχόλιο η πηγή των συναρτήσεων.

Σχετικά με τα templates

Ο έξτρα κώδικας που υπάρχει για τα templates στο header αρχείο , και πιο συγκεκριμένα αναφερόμαστε σε αυτές τις γραμμές κώδικα

```
template class BstNode<int>;  
template class BstNode<float>;  
template class BstNode<string>;  
#endif // BSTNODE_H_INCLUDED
```

Εξηγούνται από την απάντηση 1 στον σύνδεσμο:

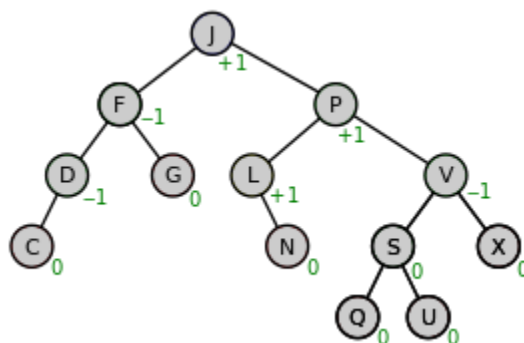
<https://stackoverflow.com/questions/8752837/undefined-reference-to-template-class-constructor?fbclid=IwAR3WTft2nwU3A68aPT4XJgKRGzi6FnR--Kob-iPYHW1hvy0VWVG3KsbMv23w>

AVL TREE

Το AVL Tree έχει πάρει το όνομα του από τους Adelson-Velsky και Landis. Είναι ένα ισοσταθμισμένο BST δέντρο. Σύμφωνα με την οπτικοποίηση του AVL Tree που υπάρχει στον σύνδεσμο:

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html> το Insert μοιάζει πολύ με αυτό του BST μόνο που υπάρχει μια έξτρα διαδικασία. Πιο συγκεκριμένα αφού βάλουμε έναν νέο κόμβο ελέγχουμε τα βάρη του κάθε κόμβου.

Ως βάρη ορίζουμε για κάθε κόμβο την διαφορά που έχουν στα ύψη τους το αριστερό με το δεξί υποδέντρο του. Για παράδειγμα στην εικόνα που ακολουθεί δίπλα σε κάθε κόμβο φαίνεται με πράσινο χρώμα αυτή η μετρική που μόλις περιγράψαμε. Αν σε όλους τους κόμβους οι τιμές αυτές είναι 1, -1 ή 0 τότε λέμε ότι το BST μας είναι ένα AVL Tree αφού είναι ισοσταθμισμένο. Επομένως σε κάθε εισαγωγή νέου κόμβου θα πρέπει να πραγματοποιούνται οι κατάλληλοι έλεγχοι για να παραμένει ισοσταθμισμένο το δέντρο μας.



Πιο συγκεκριμένα μετά από κάθε εισαγωγή στο δέντρο (οι εισαγωγές γίνονται είπαμε ακριβώς με τον ίδιο τρόπο που κάνουμε εισαγωγή στο BST για αυτό και δεν θα αναλυθούν από την αρχή, γίνεται έλεγχος και αν το δέντρο χρειάζεται να ισοσταθμιστεί αυτό επιτυγχάνεται με τις ανάλογες περιστροφές.

Έστω ότι φτιάχνουμε ένα AVL δέντρο. Αρχικά προσθέτουμε τον κόμβο 10



Στη συνέχεια προσθέτουμε τον κόμβο 20



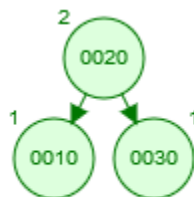
Τώρα αν πάμε να προσθέσουμε τον κόμβο 30, θα καταλήγουμε σε ένα όχι AVL Tree αφού το δέντρο μας θα σταματήσει να είναι ισοσταθμισμένο όπως φαίνεται παρακάτω.



Και αφού ανανεωθούν τα ύψηθα έχουμε το παρακάτω:



Ο κόμβος με την τιμή 10 θα έχει balance factor (αναλύεται παρακάτω τι είναι) 2 → αυτό σημαίνει ότι το δέντρο μας χρειάζεται ισοστάθμιση. Επομένως θα γίνει η ανάλογη περιστροφή για να ισοσταθμιστεί το δέντρο, και θα καταλήξουμε σε κάτι της μορφής:



Όπου έχουν ανανεωθεί τα βάρη σε τιμές που είναι αποδεκτές. Τώρα στην ρίζα του δέντρου έχουμε τον κόμβο με τιμή 20 και όχι αυτόν με την τιμή 10 που είχαμε όταν ξεκινήσαμε.

Κλάση AVLNode

Όπως κάναμε και προηγουμένως θα έχουμε μια κλάση που θα αναπαριστά τον κάθε κόμβο του δέντρου. Αυτή θα έχει όπως και πριν τα πεδία «data», «appearances», τους pointers για τα παιδιά «left» και «right» ενώ τώρα θα έχουμε επιπλέον ένα πεδίο «heightt» που θα έχει το βάρος που αναλύσαμε προηγουμένως.

```
class AVLNode
{
private:
    Type data;
    int appearances;
    AVLNode<Type>* left;
    AVLNode<Type>* right;
    int heightt;
```

Συνάρτηση height

Εκτός από το πεδίο height έχουμε και την συνάρτηση height η οποία επιστρέφει το μέγιστο από τα δύο ύψη του κάθε υποδέντρου για έναν κόμβο.

```
template <class Type>
int AVLNode<Type>::height (AVLNode<Type>* Node)
{
    if (Node == NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int rightHeight = height(Node->left);
        int leftHeight = height(Node->right);

        /* use the larger one */
        if (leftHeight > rightHeight)
            return(leftHeight+1);
        else
            return(rightHeight+1);
    }
}
```

Συνάρτηση GetNewNode

Ακριβώς όπως και στο BST έχουμε την συνάρτηση «GetNewNode» που είναι κάτι «σαν» constructor για ένα Node. Αναλύσαμε προηγουμένως γιατί. Όπως και πριν

```

template <class Type>
AVLNode<Type>* AVLNode<Type>::GetNewNode(Type data)
{
    AVLNode<Type>* newNode = new AVLNode(); //- (AVLNo
    newNode->appearances = 1;
    newNode->data=data;
    newNode->left = newNode->right = NULL; //initializ
    newNode->heightt=1;
    return newNode;
}

```

φτιάχνει ένα καινούργιο Node δεσμεύοντας την κατάλληλη μνήμη βάζει τις κατάλληλες τιμές στα πεδία του , με την διαφορά από την κλάση BST ότι εδώ βάζουμε και τιμή στο πεδίο heightt=1, και τέλος επιστρέφουμε τον pointer.

Συναρτήσεις rightRotate και leftRotate

```

template <class Type>
AVLNode<Type>* AVLNode<Type>::rightRotate(AVLNode<Type>* y)
{
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;
    //-----Y-----X-----
    //ROTATION T1 and T3 stay the same
    x->right = y;
    y->left = T2;
    //T1--T2-----T2--T3-----
    //change heights
    x->heightt = max(height(x->left),height(x->right))+1;
    y->heightt = max(height(y->left),height(y->right))+1;

    return x;
}

template <class Type>
AVLNode<Type>* AVLNode<Type>::leftRotate(AVLNode<Type>* x)
{
    //T1 and T3 don't change root
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;
    //----/-\-----/-\-----
    y->left = x;
    x->right = T2;
    //-----T2--T3-----T1--T2-----
    x->heightt = max(height(x->left),height(x->right))+1;
    y->heightt = max(height(y->left),height(y->right))+1;

    return y;
}

```

Η πρώτη συνάρτηση δέχεται έναν κόμβο AVL τον y. Αυτό που κάνει είναι να κάνει μια δεξιά περιστροφή στον κόμβο y και στο αριστερό παιδί του. Κάτι τέτοιο θα μπορούσε να οπτικοποιηθεί ως εξής:

```

//-----
//-----Y-----X-----
//-----/-\-----/-\-----
//---X---T3---RIGHT---T1---Y---
//---/-\---ROTATION---/-\---
//---T1---T2-----T2---T3---
//-----
//-----

```

Δηλαδή αυτά που αλλάζουν είναι ότι πλέον ως δεξί παιδί του x θα έχουμε τον κόμβο y, ενώ ως αριστερό παιδί του y (εκεί που βρισκόταν προηγουμένως το x) θα έχουμε το δεξί υποδέντρο του x. Ενώ το T1 που είναι το αριστερό υποδέντρο του x θα παραμείνει αριστερό υποδέντρο του x. Φυσικά χρησιμοποιώντας την συνάρτηση height θα ανανεώσουμε τις τιμές στα πεδία heightt που είναι το ύψος του κάθε κόμβου. Τέλος επιστρέφουμε τον κόμβο x ο οποίος έχει πάρει την θέση πλέον του y.

Αντιστοίχως στην συνάρτηση leftRotate, η οποία υλοποιεί μια αριστερή περιστροφή δοθέντων 2 κόμβων κάνουμε το εξής

```

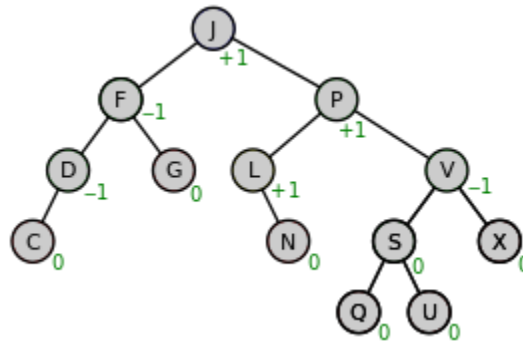
//-----
//-----X-----Y-----
//-----/-\-----/-\-----
//---T1---Y---LEFT---X---T3---
//---/-\---ROTATION---/-\---
//---T2---T3-----T1---T2---
//-----
//-----

```

Ως είσοδο δίνουμε στην συνάρτηση τον κόμβο x, και θέλουμε να τον περιστρέψουμε αριστερά. Επομένως θέλουμε το x να γίνει αριστερό παιδί του κόμβου y. Αυτό θα έχει σαν αποτέλεσμα το αριστερό υποδέντρο του y (το T2) να γίνει δεξί υποδέντρο του x. Είναι φανερό ότι αυτή η διαδικασία είναι ακριβώς η αντίστροφη από αυτήν που περιγράψαμε προηγουμένως. Και φυσικά και σε αυτήν την περίπτωση θα πρέπει να ενημερώσουμε το ύψος του κάθε κόμβου που ενδέχεται να έχει αλλάξει. Τέλος επιστρέφουμε τον κόμβο y που θα έχει πάρει πλέον την θέση του x.

Συνάρτηση balance

Αυτή η συνάρτηση είναι μια βοηθητική συνάρτηση που μας επιτρέπει δοθέντων τον υψών του κάθε κόμβου να υπολογίσουμε τον balance factor ακριβώς όπως φαίνεται στην εικόνα:



Για κάθε κόμβο δηλαδή υπολογίζουμε την διαφορά του ύψους του αριστερού υποδέντρου (του μέγιστου ύψους) με το δεξί υποδέντρο. Για παράδειγμα στον κόμβο J έχουμε +1 γιατί το δεξί υποδέντρο έχει μέγιστο ύψος =4 (j->p->v->s->q|o) ενώ το αριστερό υποδέντρο έχει ύψος =3 (j->f->d->c) για αυτό έχουμε +1 στην διαφορά ύψος_δεξί-ύψος_αριστερό. Μια εναλλακτική προσέγγιση θα ήταν να κάνουμε ακριβώς την αντίθετη (ύψος_αριστερό-ύψος_δεξί) που είναι αυτή που υλοποιήθηκε παρακάτω και δεν έχει ακριβώς καμία διαφορά απλά θα πρέπει να προσέξουμε πως θα χειριστούμε τις συνθήκες μέσα στα rotates.

```
template <class Type>
int AVLNode<Type>::balance (AVLNode<Type>* Node)
{
    if (Node == NULL)
        return 0;
    return height(Node->left) - height(Node->right);
}
```

Συνάρτηση Insert και σενάρια Rotate μετά το Insert

Το πρώτο μέρος της συνάρτησης είναι ολόιδιο με την αντίστοιχη συνάρτηση που υλοποιήθηκε για να κάνει εισαγωγή στην BST δομή και φαίνεται παρακάτω:

```

template <class Type>
AVLNode<Type>* AVLNode<Type>::Insert(AVLNode<Type>* root, Type data) /
{
    if(root==NULL)
    {
        root = AVLNode::GetNewNode(data); //if tree is empty we create
    }
    else if(data ==(root->data)) //its the same string appearing again
    {
        root->appearances++;
    }
    else if(data <(root->data) )
    {
        root->left = Insert(root->left,data); //we add the Node to the
    }
    else
    {
        root->right = Insert(root->right,data);
    }
}

```

Στη συνέχεια όμως, αφού κάναμε εισαγωγή του νέου κόμβου, χρειάζεται να ανανεώσουμε το ύψος του root, και να υπολογίσουμε αν ο balance factor του έχει μείνει μέσα στα επιθυμητά όρια για να συνεχίσει η δομή μας να είναι AVL Tree και μετά την εισαγωγή.

```

//so we have to update its height
root->heightt = max(height(root->left), height(root->right));
//and we have to check if the root became unbalanced after the i;
int rootBalance = balance(root);
// RR rotation
if (rootBalance > 1 && data < root->left->data)
    return rightRotate(root);

// LL
if (rootBalance < -1 && data > root->right->data)
    return leftRotate(root);

// LR
if (rootBalance > 1 && data > root->left->data)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

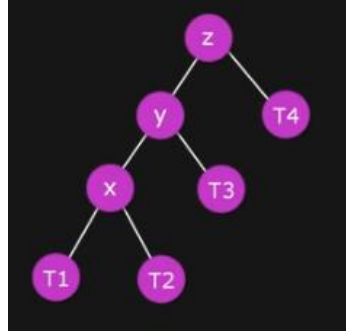
// RL
if (rootBalance < -1 && data < root->right->data)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;

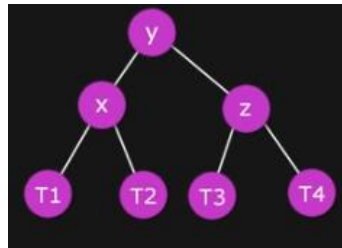
```

Αφού αλλάξαμε το ύψος του, υπολογίζουμε και τον νέο balance factor. Υπάρχουν 5 σενάρια στα οποία μπορεί να βρισκόμαστε.

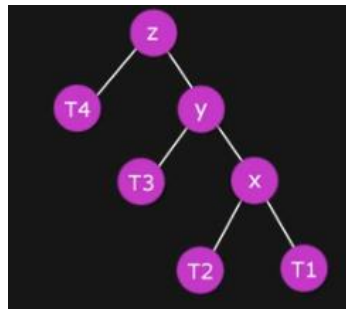
1. RR περιστροφή: Αν μετά την εισαγωγή έχουμε balance factor >1 και το data που βάλαμε είναι μικρότερο από το data που υπάρχει στο αριστερό υποδέντρο, αυτό σημαίνει ότι μετά την εισαγωγή βρισκόμαστε σε ένα τέτοιο σενάριο:



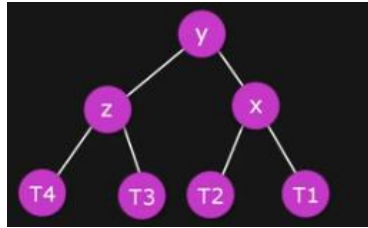
Με την δομή μας να μην είναι AVL Tree. Για να γίνει αυτή η δομή AVL Tree αρκεί να κάνουμε μια δεξιά περιστροφή στον κόμβο Z. Και όπως περιγράψαμε την δεξιά περιστροφή θα έχουμε σαν αποτέλεσμα το εξής:



2. LL περιστροφή: σε αυτήν την περίπτωση το balance factor μας όπως έχει οριστεί θα είναι μικρότερο του -1 (θα είναι -2) και θα ισχύει η συνθήκη $data > root \rightarrow right \rightarrow data$. Επομένως θα έχουμε κάτι της μορφής:

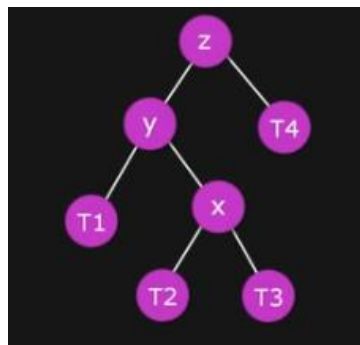


Και με μία LL περιστροφή στον κόμβο Z όπως αναλύθηκε προηγουμένως θα καταλήξουμε σε κάτι της μορφής:

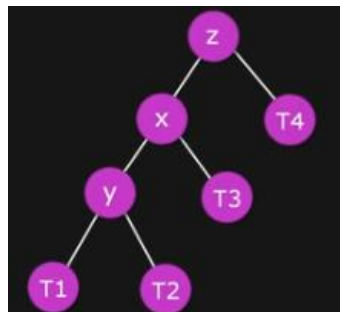


Που θα είναι ένα ισοσταθμισμένο AVL Tree.

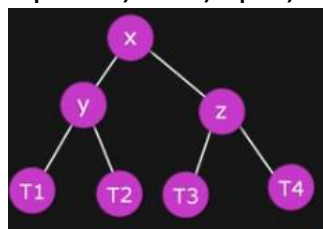
3. LR Περιστροφή: Σε περίπτωση που το Balance factor θα γίνει 2 και θα ισχύει η συνθήκη: $data > root \rightarrow left \rightarrow data$ σημαίνει ότι το δέντρο μας θα έχει πάρει την μορφή:



Και για να ισοσταθμιστεί χρειάζεται με Left περιστροφή στον κόμβο γ (που είναι το αριστερό παιδί του Root z) όπου θα δημιουργήσει το δέντρο της μορφής:

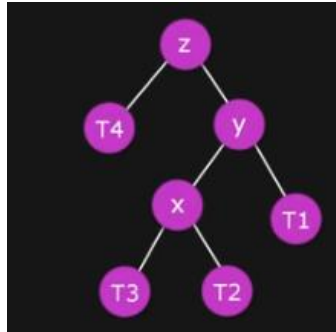


Το οποίο αν το περιστρέψουμε δεξιά ως προς τον root Node θα γίνει:



Το οποίο θα είναι AVL Tree.

4. RL περιστροφή: είναι η τελευταία συνθήκη που έχει μείνει όπου το Balance factor είναι μικρότερο του -1 και ισχύει η συνθήκη: $data < root \rightarrow right \rightarrow data$. Το δέντρο θα είναι αρχικά της μορφής:



Και με μία αριστερή περιστροφή πρώτα στον δεξί κόμβο του Z και ύστερα με μια αριστερή περιστροφή στον Z θα καταλήξουμε σε ισοσταθμισμένο BST δηλαδή σε AVL Tree.

5. Η Πέμπτη περίπτωση είναι να μην χρειάζεται να εφαρμόσουμε καμία περιστροφή όπου απλούστατα η συνάρτηση μας δεν μπαίνει μέσα σε καμία if που εξετάζει τον balance factor.

Σε όλες τις περιπτώσεις όπως και στην παραδοσιακή Inert συνάρτηση της BST δομής επιστρέφουμε τον κόμβο Root για να ανανεωθεί η διεύθυνση του στην συνάρτηση main και για να δουλέψει η αναδρομική εισαγωγή σύμφωνα με τους κανόνες που έχουμε θέσει.

Συνάρτηση Search

Η Συνάρτηση Search της AVLNode δομής δεν έχει καμία απολύτως διαφορά με την αντίστοιχη συνάρτηση της BST δομής. Βέβαια θα δείξουμε αργότερα από θεωρητικής πλευράς ότι παρόλο που η συνάρτηση είναι ίδια επειδή έχουμε βελτιώσει την δομή μας (το AVL είναι μια βελτιωμένη δομή BST) η απόδοση της συνάρτησης Search στο AVL δένδρο είναι σαφώς καλύτερη από αυτή της BST γιατί αν και ο κώδικας είναι ίδιος εφαρμόζονται σε διαφορετική δομή.

Συνάρτηση Delete

Η συνάρτηση Delete αποτελείται από 2 κομμάτια. Το πρώτο κομμάτι της είναι ολόιδιο με την Delete της BSTNode κλάσης.

```

template <class Type>
AVLNode<Type>* AVLNode<Type>::Delete(AVLNode<Type>* root, Type data)
{
    if(root == NULL)
        return root;
    else if(data < (root->data) )
        root->left = Delete(root->left, data);
    else if (data > (root->data) )
        root->right = Delete(root->right, data);

    else//delete or edit appearances of this node
    {
        if(root->appearances > 1)
        {
            root->appearances--;
        }
        //Node has no child
        else if (root->left == NULL && root->right == NULL)
        {
            delete root;
            root = NULL;
        }
        //Node has 1 child
        else if (root->left == NULL)
        {
            AVLNode *temp = root;
            root = root->right;
            delete temp;
        }
        else if (root->right == NULL)
        {
            AVLNode *temp = root;
            root = root->left;
            delete temp;
        }
        //Node has 2 children
        else
        {
            AVLNode *temp = findMin(root->right);
            root->data = temp->data;
            root->right = Delete(root->right, temp->data);
        }
    }
}

```

Και το δεύτερο κομμάτι που ελέγχει αν το δέντρο παραμένει AVL (δηλαδή αν είναι ισοσταθμισμένο, ο κώδικας που υλοποιεί αυτόν τον έλεγχο είναι ακριβώς ίδιος με τον κώδικα που χρησιμοποιήθηκε στην συνάρτηση Insert.

Τέλος όπως και στην κλάση BSTNode χρησιμοποιήθηκε μια βοηθητική συνάρτηση findMin η οποία υπολογίζει το ελάχιστο στοιχείο του δεξιού υποδέντρου και χρησιμοποιείται από την συνάρτηση Delete ακριβώς με τον ίδιο τρόπο όπως αναλύθηκε στην Κλάση BstNode.

```

    }
    if(root == NULL)
        return root;

    //balance the tree
    //o kwdikas einai idios me to insert
    root->heightt = max(height(root->left), height(root->right)) + 1;
    int rootBalance = balance(root);
    // RR rotation
    if (rootBalance > 1 && balance(root->left) >= 0 )
    {
        return rightRotate(root);
    }
    // LL
    if (rootBalance < -1 && balance(root->right) <= 0)
    {
        cout<<"MPIKE"<<endl;
        return leftRotate(root);
    }
    // LR
    if (rootBalance > 1 && balance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // RL
    if (rootBalance < -1 && balance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;

```

Συναρτήσεις printInorder, printPostorder, printPreorder

Παρακάτω υπάρχει ο κώδικας που υλοποιεί τις γνωστές διασχίσεις του δέντρου ακριβώς όπως περιγράψαμε στην προηγούμενη ενότητα που αφορά την κλάση BSTNode. Οι συναρτήσεις είναι ολόιδιες και δεν έχει γίνει καμία διαφορά.

Σε αυτό το σημείο καλό είναι να γίνει μια παρένθεση σχετικά με τον χρόνο που χρειάζεται για να δημιουργηθεί το AVL Tree. Το AVL Tree κάνει περίπου 2 λεπτά να δημιουργηθεί βάση του αρχείου που ζητήθηκε από την εκφώνηση να αποθηκεύσουμε σε μορφή AVL Tree. Αυτό συμβαίνει επειδή υπάρχουν πάρα πολλές αναδρομές στον υπολογισμό των υψών των κόμβων της δομής. Έγινε προσπάθεια απαλοιφής όσων περισσότερων άσκοπων αναδρομών ήταν δυνατό. Η απόδοση της Insert έχει βελτιωθεί αρκετά, όμως θεωρώ πως είχε ακόμη πολλά περιθώρια βελτίωσης.

Μια εναλλακτική που θα μπορούσαμε να χρησιμοποιήσουμε για να βελτιώσουμε την απόδοση δημιουργίας του δέντρου, θα ήταν αφού το

δημιουργήσουμε πρώτη φορά (με τον κλασσικό τρόπο όπως δημιουργείται από την εκτέλεση της συνάρτησης main) θα μπορούσαμε να αποθηκεύσουμε με την βοήθεια της συνάρτησης `printPreorder` ένα αντίγραφο του δέντρου σε ένα αρχείο `txt`, και επομένως την επόμενη φορά που θα εκτελούσαμε το πρόγραμμα αντί να ξανά φτιάχναμε το δέντρο από την αρχή, θα μπορούσαμε να το «διαβάσουμε» από το αρχείο που έχουμε δημιουργήσει. Για προφανείς λόγους (αφού κάτι τέτοιο θα θεωρούνταν κλεψιά ως προς την σύγκριση των 2 δομών) αν και χρησιμοποιήθηκε αυτή η εναλλακτική όταν γινόταν το `testing` του κώδικα ως προς την συνάρτηση `Search`, όταν παραδόθηκε η εργασία αυτή η λύση διαγράφηκε εντελώς από το Project και αναφέρεται μόνο σε αυτό το σημείο του project.

Λοιπές συναρτήσεις που χρησιμοποιήθηκαν για να διευκολυνθεί η ανάπτυξη του Project Όπως και στην περίπτωση του BST χρησιμοποιήθηκαν και εδώ ακριβώς οι ίδιες συναρτήσεις για να γίνει έλεγχος της δομής. Πιο συγκεκριμένα αυτές είναι οι :

1. `Print2DUtil`
2. `Print2D`
3. `GetSize`

Καθώς έγιναν και οι απαραίτητες αλλαγές στον κώδικα για μπορεί να δέχεται και τύπου `int` στοιχεία και τύπου `float` άλλα και τύπου `string`.

Hash Table

Για την κλάση HashTable δημιουργήθηκαν 2 κλάσεις. Αυτή που θα αναλύσουμε πρώτα είναι η κλάση HashNode και στη συνέχεια θα αναλυθεί και η HashTable η οποία βασίζεται στην πρώτη.

Κλάση HashNode

Χρειάστηκε να υλοποιηθεί μόνο ο header της κλάσης αυτής. Πιο συγκεκριμένα έχουμε το εξής:

```
class HashNode
{
public:
    string data;
    int appearances;
    HashNode(string data);
    void changeAppearances(int newAppearances);
};
```

Η κλάση υλοποιεί ένα αντικείμενο που αποτελεί μια εγγραφή στο Hash Table που ζητήθηκε να σχεδιάσουμε. Αυτό αποτελείται από 2 πεδία . Το ένα περιέχει το data (συμβολοσειρά) που θέλουμε να αποθηκεύσουμε, ενώ υπάρχει ο constructor και η συνάρτηση changeAppearances που υλοποιήθηκε με σκοπό όταν εισάγουμε μια λέξη δεύτερη φορά απλά να προσθέτουμε +1 στο πόσες φορές εμφανίζεται (είναι Setter για το πεδίο appearances) .

Ο constructor της κλάσης και η συνάρτηση changeAppearances υλοποιήθηκαν στο ένα και μοναδικό .cpp αρχείο που αναπτύχθηκε κατά την υλοποίηση της δομής. Παρακάτω φαίνονται οι υλοποιήσεις:

```
HashNode::HashNode(string data)
{
    this->data = data;
    this->appearances = 1;
}

void HashNode::changeAppearances(int newAppearances)
{
    appearances=newAppearances;
}
```

Κλάση Hash Table

Για να υλοποιηθεί η κλάση που να είναι ένας πίνακας που θα περιέχει μέσα αντικείμενα HashNode, φτιάχτηκε ένας διπλός pointer (pointer to pointer). Το πως ο pointer σε pointer λειτουργεί σαν 2d array το οποίο δεσμεύει δυναμικά μνήμη εξηγείται εδώ: <https://stackoverflow.com/questions/16001803/pointer-to-pointer-dynamic-two-dimensional-array> .

```
class HashTable
{
public:
    HashNode **ptr;
    int capacity;
    int size;
    HashTable(int capacity);
    int hashCode(string data);
    void insertNode(string data);
    int Search(string data);
    int sizeofMap();
    bool isEmpty();
    void display();
    HashTable* reHashing(HashTable* h1, double newCapacity);
};
```

Επίσης υπάρχει και μια μεταβλητή που λέγεται capacity και την χρησιμοποιούμε για να ορίσουμε την χωρητικότητα που θα έχει το Hash Table που σχεδιάζουμε. Αρχικά ξεκινούμε με capacity=5, και στη συνέχεια δεσμεύουμε μνήμη με τρόπο θα αναλυθεί όταν θα αναλύουμε την main συνάρτηση του project. Ενώ στην μεταβλητή size αποθηκεύουμε το πόσες θέσεις είναι δεσμευμένες για να μπορούμε να προβλέψουμε πότε θα χρειαστεί να κάνουμε το rehashing και να δεσμεύσουμε περισσότερη μνήμη (προφανώς το αρχικό μέγεθος =5 δεν επαρκεί για να χωρέσουμε το αρχείο Gutenberg) .

Ακολουθεί η δήλωση του constructor της κλάσης που δέχεται ως παράμετρο μόνο το μέγεθος που θα έχει το HashTable.

```
HashTable::HashTable(int capacity)
{
    //Initial capacity of hash table
    this->capacity = capacity;
    size=0;
    ptr = new HashNode*[capacity];

    //Initialise all elements of table as NULL
    for(int i=0 ; i < capacity ; i++)
    {
        ptr[i] = NULL;
    }
}
```

Στον constructor δεσμεύουμε τη μνήμη που είναι απαραίτητη θέτοντας όλους τους δείκτες =NULL που σημαίνει ότι σε εκείνη τη θέση δεν έχουμε αποθηκεύσει ακόμα κάποια λέξη. Στην αρχή πχ που το μέγεθος είναι 5, δεσμεύουμε 5 Pointers σε αντικείμενο HashNode και τους θέτουμε σε NULL.

Hash Function

Χρησιμοποιήθηκε έτοιμη Hash Function που προσφέρεται από βιβλιοθήκη της c++. Ενώ φυσικά γίνεται και η κατάλληλη πράξη για να δούμε σε ποια θέση του πίνακα μας θα πάμε να τοποθετήσουμε την λέξη που θέλουμε να εισάγουμε.

```
//https://stackoverflow.com/questions/18174988/how-c
int HashTable::hashCode(string data)
{
    hash<string> mystdhash;
    //cout << mystdhash(data)%capacity<<endl;
    return mystdhash(data)%capacity;
}
```

Συνάρτηση insertNode

Αυτή την συνάρτηση την χρησιμοποιούμε όταν θέλουμε να εισάγουμε ένα νέο HashNode στο Table μας. Αρχικά δημιουργούμε το νέο αντικείμενο HashNode που θέλουμε να εισάγουμε, στη συνέχεια περνάμε το data από την Hash Function για να δούμε σε ποια θέση του πίνακα θα το εισάγουμε. Και τελικά ψάχνουμε την πρώτη άδεια θέση μετά από αυτήν που βγήκε ως αποτέλεσμα (από αυτήν και μετά) από την Hash Function για να το εισάγουμε. Φυσικά αν βρούμε ήδη την λέξη που θέλουμε να εισάγουμε σε κάποιον από αυτούς τους ελέγχους που κάνουμε στο data με την εντολή «ptr[hashIndex]->data.compare(data) == 0» στην γραμμή 57, τότε προφανώς δεν θέλουμε να εισάγουμε ξανά την ίδια λέξη αλλά απλά πάμε και ανανεώνουμε τις φορές που εμφανίζεται αυτή η λέξη και σταματάμε την αναζήτηση. Μόλις βρούμε κάποια άδεια θέση στον πίνακα σημαίνει ότι βάζουμε πρώτη φορά αυτήν την λέξη στο Hash Table μας επομένως απλά την τοποθετούμε, και αυξάνουμε το μέγεθος που μετράει τις δεσμευμένες θέσεις του πίνακα.

Η εντολή «hashIndex %= capacity;» υπάρχει για να μην φύγουμε έξω από τα όρια του πίνακα στην επανάληψη που κάνουμε (αν φτάσει στην τελευταία θέση τότε πηγαίνει στην πρώτη).

Συνάρτηση Search

```
//Function to search the value for a given key
int HashTable::Search(string data)
{
    // Apply hash function to find index for given key
    int hashIndex = hashCode(data);
    int counter=0;
    //finding the node with given key
    while(ptr[hashIndex] != NULL)
    {
        //Not found
        if(counter++>capacity)    //to avoid infinite loop
        {
            cout<<"Not Found"<<endl;
            return false;
        }
        //Found
        if(ptr[hashIndex]->data == data)
        {
            cout<<ptr[hashIndex]->data<<" "<<"Found: ";
            return ptr[hashIndex]->appearances ;
        }
        hashIndex++;
        hashIndex %= capacity; //an ginei iso me to capac
    }
    if(ptr[hashIndex] == NULL)
    {
        cout<<"Not Found"<<endl;
        return false;
    }
    return false;
}
```

Η συνάρτηση παίρνει ένα μοναδικό όρισμα το οποίο είναι το data (η λέξη) που αναζητούμε.

Αρχικά περνάμε το data από το Hash Function που έχουμε για να ξέρουμε από που και μετά θα αναζητήσουμε την λέξη μέσα στο Hash Table. Στην While ψάχνουμε την λέξη που θέλουμε. Αν το counter της while ξεφύγει και γίνει μεγαλύτερο του capacity του Table σημαίνει ότι η λέξη που ψάχνουμε δεν υπάρχει. Επίσης λέξη δεν υπάρχει αν βρούμε κάποια τιμή ενός Pointer μέσα στο Hash Table Να είναι NULL. Αν την βρούμε απλά τυπώνουμε ότι βρέθηκε και

επιστρέφουμε το πόσες φορές είναι καταχωρημένη αυτή η λέξη μέσα στο Hash Table μας.

Συνάρτηση rehashing

Χρησιμοποιούμε αυτήν την συνάρτηση σε περίπτωση που ξεμείνει το Hash Table που υλοποιήσαμε από χώρο, ως παραμέτρους δέχεται ένα Hash Table και μία τιμή που αποτελεί το νέο Capacity (την νέα χωρητικότητα του Hash Table).

```
HashTable* HashTable::reHashing(HashTable* h1, double newCapacity)
{
    double newCapacityInt;
    modf(newCapacity, &newCapacityInt);
    HashTable* h2 = new HashTable((int) newCapacityInt);
    //cout<<newCapacityInt<<endl;
    for(int i=0; i<h1->capacity; i++)
    {
        if(h1->ptr[i] != NULL)
        {
            h2->insertNode(h1->ptr[i]->data);
            h2->ptr[h2->hashCode(h1->ptr[i]->data)]->appearances=h1->ptr[i]->appearances;

            delete h1->ptr[i];
        }
    }
    //delete old hashtable
    delete h1;
    return h2;
}
```

Αυτό που κάνει η συνάρτηση είναι να φτιάχνει ένα καινούργιο Hash Table το οποίο θα έχει μέγεθος ίσο με newCapacity (το h2) και με την for που ακολουθεί αντιγράφουμε το παλιό Hash Table στο καινούργιο κάνοντας Insert μια προς μία τις λέξεις . Δεν μπορούμε απλά να αντιγράψουμε το παλιό Hash Table στο καινούργιο γιατί η χωρητικότητα του Hash Table παίζει σημαντικό ρόλο στο που θα τοποθετηθεί μια λέξη. Φυσικά αποδεσμεύουμε και την μνήμη που είχαμε δεσμεύσει για το παλιό Hash Table σταδιακά. Στο τέλος διαγράφουμε τον pointer h1 και επιστρέφουμε τον h2 που είναι το νέο Hash Table .

Βοηθητικές συναρτήσεις

Χρησιμοποιήθηκαν επίσης οι συναρτήσεις sizeofMap που είναι ένας Getter για το πεδίο size. Η συνάρτηση isEmpty που ελέγχει αν το Table μας είναι άδειο και επιστρέφει τιμή bool. Ενώ υλοποιήθηκε και μια συνάρτηση display που τυπώνει

όλο το Hash Table στο Command Line για να γίνει ο κατάλληλος έλεγχος για την λειτουργία του.


```
//Return current size
int HashTable::sizeofMap()
{
    return size;
}

//Return true if size is 0
bool HashTable::isEmpty()
{
    return size == 0;
}

//Function to display the stored key value pairs
void HashTable::display()
{
    for(int i=0 ; i<capacity ; i++)
    {
        if(ptr[i] != NULL)
        {
            cout << "data = " << ptr[i]->data
                << " appearances = " << ptr[i]->appearances << endl;
        }
        else
        {
            cout<<"NULL"<<endl;
        }
    }
}
```

Main

Αφού αναλύθηκαν επαρκώς όλες οι κλάσεις και όλες οι συναρτήσεις που υλοποιήθηκαν για τις δομές θα αναλύσουμε και την λειτουργία της main.

Αρχικά σημειώνουμε ότι υπάρχουν 7 κομμάτια στην main . Τα 1 έως 3 φτιάχνουν αντίστοιχα τις 3 δομές : BST, AVL Tree και Hash Table. Διαβάζουν το αρχείο γραμμή-γραμμή και αφού πετάξουν κάποια άχρηστα σύμβολα που ίσως υπάρχουν μέσα σε μια λέξη , όπως τα ()-#*+!@#\$%^&*,_'\?/<>;][\|ι~|H|| και αφού γίνουν και οι κατάλληλες αλλαγές για να πάρουμε από κάθε γραμμή του αρχείου που διαβάζουμε τις λέξεις τότε κάνουμε Insert την λέξη στη δομή. Επίσης υπάρχει και μια βοηθητική εντολή που μας επιτρέπει να

δούμε κατά την εκτέλεση του προγράμματος πόσο ποσοστό του αρχείου έχουμε διαβάσει (υλοποιήθηκε κυρίως για την δομή AVL Tree που συγκριτικά με τις άλλες αργεί πολύ να δημιουργηθεί εξαιτίας των πολλών αναδρομών και τον πολλών rotate που κάνει). Επίσης για κάθε δομή τυπώνεται ο χρόνος που χρειάστηκε για να δημιουργηθεί.

Αξίζει να δώσουμε μια προσεκτική ματιά στο 3^ο τμήμα που κάνουμε την δημιουργία του Hash Table. Αρχικά βλέπουμε ότι το αρχικό μέγεθος του HashTable είναι 5.

```
//HASH TABLE  
HashTable *h = new HashTable(5);
```

Ενώ σε κάθε εισαγωγή γίνεται έλεγχος για το αν χρειάζεται να κάνουμε rehashing

```
if((float) h->sizeofMap() == h->capacity)  
{  
    h=h->reHashing(h,h->capacity*3/2+h->capacity);  
}
```

Στην περίπτωση που χρειάζεται να κάνουμε rehashing το μέγεθος του νέου Hash Table θα είναι το αρχικό μέγεθος του Hash Table συν 1.5 φορά το αρχικό μέγεθος του. Δηλαδή κάτι παραπάνω από το διπλάσιο του. Η υλοποίηση αυτή είναι εμπνευσμένη από τον τρόπο που δεσμεύεται μνήμη στις ArrayLists στην Java.

Στο 4^ο τμήμα του κώδικα που υπάρχει στη Main δημιουργούμε το σύνολο Q. Στην μεταβλητή qsize ορίζουμε το μέγεθος που θέλουμε να έχει το σύνολο Q. Σημειώνεται πως για να κάνουμε testing τις δομές που υλοποιήθηκαν χρησιμοποιήθηκαν περισσότερες από 1000 λέξεις. Πιο συγκεκριμένα χρησιμοποιήθηκαν από 5000 μέχρι 50.000 λέξεις για να φανεί η διαφορά στις αποδόσεις των δομών.

```

int qsize=1000;
srand((unsigned int)time(NULL));
int random[qsize];
for(int i=0 ; i<qsize; i++)
{
    random[i] = rand() % 10000 + 1;
}
int n = sizeof(random)/sizeof(random[0]);
sort(random, random+n);
string Q[qsize];
int j=0;
int l=0;
myReadFile.open("small-file.txt");
if (myReadFile.is_open())
{
    while (!myReadFile.eof())
    {
        myReadFile >> line;
        for (unsigned int i = 0; i < strlen(chars); ++i)
        {
            line.erase (std::remove(line.begin(), line.end(), cha
        }
        transform(line.begin(), line.end(), line.begin(), ::tolower
        if(j==random[l])
        {
            int templ=1;
            while(random[templ]==random[l])
            {
                Q[l]=line;
                l++;
            }
            j++;
        }
    }
}
myReadFile.close();

```

Για να δημιουργήσουμε το σύνολο Q (όταν λέμε σύνολο δεν αναφερόμαστε στον παραδοσιακό ορισμό του συνόλου/Set όπου υπάρχει το κάθε στοιχείο μια φορά, γιατί στο σύνολο Q μια λέξη μπορεί να υπάρχει περισσότερες από μια φορές) αρχικά παίρνουμε 1000 τυχαίες τιμές από το 0 έως το 10.000. Τις κάνουμε sort μέσα σε έναν πίνακα random. Και βάζουμε μια μετρική j που μετράει τις επαναλήψεις μέσα στην while που βλέπουμε στην εικόνα. Όταν αυτή η μετρική j γίνει ίση με την πρώτη τιμή στην random τότε παίρνουμε την λέξη που υπάρχει εκείνη την στιγμή στην μεταβλητή Line και την βάζουμε στο σύνολο Q. Φυσικά γίνεται και ο έλεγχος αν αυτή η τιμή j είναι ίδια και με την επόμενη της random συνάρτησης που σημαίνει ότι ξανά παίρνουμε αυτή την λέξη δεύτερη φορά , 3^η κ.ο.κ.

Στα τμήματα 5 έως 7 γίνονται οι αναζητήσεις σε κάθε δομή και τυπώνεται το αποτέλεσμα τους στην Command Line.

Πειραματικά αποτελέσματα και σύγκριση των δομών

Χρόνοι Κατασκευής των δομών:

BST

```
Time to construct BST: 1.7714 seconds
```

AVL Tree

```
Time to construct AVL Tree: 136.541 seconds
```

Hash Table

```
Time to construct Hash Table: 1.50885 seconds
```

Εδώ αξίζει να σχολιάσουμε τον πολύ μεγαλύτερο χρόνο που κάνει η δομή AVL να φτιαχτεί. Αυτό όπως είπαμε οφείλετε στις πολλές αναδρομές που γίνονται μέσα στο δέντρο (και κυρίως σε αυτές που γίνονται πολύ συχνά που στην περίπτωση μας είναι η επαναλαμβανόμενη αναζητήσεις των υψών που γίνονται σε πολλά σημεία του κώδικα. Βασισμένος σε testing που έκανα στην δομή AVL είμαι στην θέση να πω ότι ο κώδικας τρέχει σωστά αλλά είναι πολύ πολύ αργός . Επομένως από θέματα απόδοσης στην δημιουργία του είναι πολύ κατώτερος συγκριτικά με τις άλλες 2 δομές που δημιουργούνται σε μόλις 1.5 δευτερόλεπτο.

Χρόνοι Αναζήτησης:

BST

```
Time taken for Searches in BST: 0.492102 seconds
```

AVL Tree

```
Time taken for Searches in AVL Tree: 0.49011 seconds
```

Hash Table

```
Time taken for Searches in Hash Table: 0.487111 seconds
```

Με μια πρώτη ματιά στους χρόνους αναζήτησης φαίνεται πως δεν έχουμε και καμία τρομερή διαφορά στους χρόνους. Όμως αυτοί οι χρόνοι είναι σε μόνο 1000 αναζητήσεις σε κάθε δομή και αφού τυπώνουμε κάθε φορά το «word Found: x times || θα ήταν σωστό να παρατηρήσουμε ότι οι χρόνοι είναι τόσο αργοί και τόσο κοντά ο ένας στον άλλον επειδή η περισσότερη ώρα «χαραμίζεται» στα print των συναρτήσεων.

Αν τρέξουμε τον ίδιο κώδικα χωρίς όμως να μας επιστρέφει το πρόγραμμα αν βρήκε την λέξη η όχι και αν τρέξουμε για μεγαλύτερα σύνολα Q (πχ 70.000 λέξεις) θα πάρουμε τα εξής αποτελέσματα:

```
Time taken for Searches in BST: 0.09002 seconds
-----

Time taken for Searches in AVL Tree: 0.08902 seconds
-----

Time taken for Searches in Hash Table: 0.011003 seconds
-----
```

```
Time taken for Searches in BST: 0.075017 seconds
```

```
-----
```

```
Time taken for Searches in AVL Tree: 0.074017 seconds
```

```
-----
```

```
Time taken for Searches in Hash Table: 0.009001 seconds
```

```
-----
```

```
Time taken for Searches in BST: 0.076026 seconds
```

```
-----
```

```
Time taken for Searches in AVL Tree: 0.074154 seconds
```

```
-----
```

```
Time taken for Searches in Hash Table: 0.009002 seconds
```

```
-----
```

Σε αυτές τις περιπτώσεις είναι φανερό ότι η δομή AVL Tree τα πηγαίνει καλύτερα από την BST στις περισσότερες περιπτώσεις και αυτό είναι λογικό αν

μελετήσει κανείς τους χρόνους απόδοσης της κάθε δομής που φαίνονται παρακάτω:

Για το BST έχουμε

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Για το AVL Tree έχουμε

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)^{[1]}$	$O(\log n)^{[1]}$
Insert	$O(\log n)^{[1]}$	$O(\log n)^{[1]}$
Delete	$O(\log n)^{[1]}$	$O(\log n)^{[1]}$

Στη μέση περίπτωση δεν έχουμε καμία διαφορά , το θέμα είναι τι συμβαίνει στην χειρότερη περίπτωση όπου το AVL Tree έχει προφανώς καλύτερη πολυπλοκότητα από το BST.

Επίσης θα ήταν σωστό να παρατηρήσουμε ότι το Hash Table στη μέση περίπτωση έχει πολύ καλύτερη αποδοτικότητα από όλες τις δομές.

Αποδοτικότητα Hash Table

Algorithm	Average	Worst case
Space	$O(n)^{[1]}$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Όλες οι δομές έχουν ακριβώς την ίδια πολυπλοκότητα σε χώρο . Σε Insert Delete και Search όμως το Hash Table έχει πολύ καλύτερη απόδοση στη μέση περίπτωση.

Δεν υπάρχει γενικός κανόνας το ποια δομή είναι καλύτερο να ακολουθήσουμε. Εξαρτάται από το σύνολο που μας έχει δοθεί και από την μορφή των δεδομένων που εισάγουμε καθώς και από τον τρόπο με τον οποίο διαχειριζόμαστε τις δομές. Για παράδειγμα αν έχουμε να κάνουμε πολλές φορές διαγραφή, και για την δομή Hash Table έχουμε υλοποιήσει μια διαγραφή που ΔΕΝ διαγράφει το στοιχείο αλλά απλά κάνει dummy την θέση του στον πίνακα (είναι κάτι σαν NULL αλλά ΔΕΝ είναι NULL, σημαίνει ότι υπήρχε στοιχείο εκεί και διαγράφηκε) , τότε δεν θα ήταν και πολύ σοφό να επιλέξουμε την δομή Hash Table γιατί μετά από κάποια Insert θα ξεμέναμε από μνήμη, η θα έπρεπε να κάνουμε πολύ συχνά rehashing. Κάτι το οποίο είναι αρκετά χρονοβόρο.

ΤΕΛΟΣ ΕΡΓΑΣΙΑΣ