Unit II : Balanced Trees : AVL Trees: Maximum Height of an AVL Tree, Insertions and Deletions. 2-3 Trees: Insertion, Deletion.

Semester Regular and Supplementary Examinations, December - 2013
1 A) How do you represent an AVL tree. What is the height of an AVL tree
B) What is a 2-3 tree? Briefly describe the operations on 2-3 tree
2 A) Write down the procedure for inserting an element into 2-3 tree
B) How to perform deletion operation in an AVL tree? Explain with an example
3 A) Explain how insertion operation is performed in an AVL tree using single double rotations.
B) Write down the applications of AVL trees and 2-3 trees
4 A) What is an AVL tree? Briefly describe operations in AVL tree
B) With example, briefly describe insertion of 2-3 tree

Semester Supplementary Examinations, May 2013
1 A) What are the differences between an AVL tree and a binary tree? When would you use an AVL tree?
B) What is a 2-3 tree? Explain its insert operation with illustrative example
2 A) Create 2-3 tree from the following lists of data items.
   30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40
B) Define AVL tree. Discuss in what way is an AVL tree better than a binary tree
3 A) Describe the AVL tree’s insert method with proper illustration
B) What is an AVL tree? In what way is an AVL tree better than a Binary tree
4 A) Insert the following keys into an AVL tree:
   342, 206, 444, 523, 607, 301, 142, 183, 102, 157, and 149
   Illustrating the problematic insertions.
B) Describe the AVL tree’s delete method with proper illustration

Semester Regular Examinations, November/December - 2012
1 A) What is meant by height balanced tree? Write a program to do the insertion of an element into the AVL tree?
2 A) Construct the AVL tree for the following numbers
   1, 2, 3, 4, 5, 6, 7, 8, 9, 8.5, 14.
3 A) What are the rules for constructing 2-3 tree?
   B) How insertion is done in 2-3 tree?
4 A) Write an algorithm for deletion an element from the AVL tree.
   B) How deletion is done in 2-3 tree?
Tree: Tree is non-linear data structure that consists of root node and potentially many levels of additional nodes that form a hierarchy.

- A tree can be empty with no nodes called the null or empty tree.
- A tree is a structure consisting of one node call the root and one or more subtrees.
- **Descendant**: A node reachable by repeated proceeding form parent to child.
- **Ancestor**: a node reachable by repeated proceeding from child to parent.
- **Degree**: the number of sub-trees of a node, means the degree of an element (node) is the number of children it has. The degree of a leaf node is always 0(zero).
- **Siblings**: Nodes with the same parent.
- **Height**: number of nodes which must be traversed from the root to the reach a leaf of a tree.

**Examples**
Tree associated with a document

**Binary Tree**: - A binary tree is a tree data structure in which each node has at most two children, which referred as the left and right child.

**Full Binary Tree or Complete Trees**: A binary tree of height is ‘h’ and contains exactly “$2^h - 1$” elements is called full binary tree.

**Full Binary Tree**

$$H=4 \text{ (levels+1 of root node)}$$

$$n \rightarrow \text{number elements}= 2^4 - 1 = 15$$

(Another definition of full binary tree is, each leaf is same distance from the root)

**Linked list representation of binary tree:**

// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
}*temp;
Void create()
{  
Int x;
Temp=(struct BinaryTreeNode *)malloc(1*sizeof(structure BinaryTreenod));
Temp->value=x;
Temp->left=Temp->right=NULL;
}

Operations on Binary Tree:
Create(), Insert(), Delete(), Size(), Inorder(), Preorder(), Postorder()

Pre-order
This can be summed up as
Visit the root node (generally output this)
Traverse to left subtree
Traverse to right subtree
outputs the following:
F, B, A, D, C, E, G, I, H

In-order
This can be summed up as
Traverse to left subtree
Visit root node (generally output this)
Traverse to right subtree
outputs the following:
A, B, C, D, E, F, G, H, I

Post-order
This can be summed up as
Traverse to left subtree
Traverse to right subtree
Visit root node (generally output this)
outputs the following:
A, C, E, D, B, H, I, G, F
Binary Search Tree:
Binary search tree is also called ordered/sorted binary tree. Means Binary Search Tree is a node based binary tree data structure but it should satisfies following properties

- Every element (node) has a key or value & no two elements have the same key or value, therefore all keys are distinct.
- The left sub-tree of a node contains only nodes with key less than the root node’s key value.
- The right sub-tree of a node contains only nodes with key greater than the root node’s key value.
- The left and right sub-tree each must also be a binary search tree.
- A unique path exists from the root to every other node.

<table>
<thead>
<tr>
<th>Type</th>
<th>Binary search tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invented</td>
<td>1960</td>
</tr>
<tr>
<td>Invented by</td>
<td>P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard</td>
</tr>
</tbody>
</table>

**Time complexity in big O notation**

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Indexed binary search tree:
The indexed binary search tree derived from ordinary binary search tree by adding the field left size to each & every node in that tree.
This field gives the number of elements in the left sub-tree.
Balanced Tree
Balancing or self-balancing (Height balanced) tree is a binary search tree. Balanced tree is any node based binary search tree that automatically keeps its height (Maximum number of levels below the root) small in the face of arbitrary item insertion and deletion.

Use of Balanced tree:
Tree structures support various basic dynamic set operations including search, minimum, maximum, insert and deletion in the time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be “log N” where → number of nodes in the tree.
To ensure that the height of the tree is as small as possible for provide the best running time.
Examples of balancing tree

AVL Trees:
Introduction: An AVL tree (Adelson-Velskii and Landis' tree, named after the inventors) is a self-balancing binary search tree, invented in 1962
Definition: An AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference b/w the heights of the node’s left & right sub trees is either 0 or +1 or -1.
Balance factor = ht of left sub tree – ht of right sub tree.
Where ht=height

Example:
Structure or pseudo code for avl tree:

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

node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);

node *insert(node *,int);
int height(node *);
node *Delete(node *,int);
int BF(node *);
```

**Inserting and Deleting on AVL Trees**

**Problem:**
After insert/delete: load balance might be changed to +2 or -2 for certain nodes. _re-balance load after each step

**Requirements:** re-balancing must have O (log n) worst-case complexity

**Solution:** Apply certain “rotation” operations

**AVL tree insertion:**
After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. The balance factor is calculated as follows: balanceFactor = height (left subtree) - height(right subtree). If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

Let the node that needs rebalancing be \( \alpha \).

4 possible situations to insert in a tree
1. Insert into the left sub-tree of the left child
2. Insert into the right sub-tree of the right child
3. Insert into the left sub-tree of the right child
4. Insert into the right sub-tree of the left child

If an insertion of a new node makes an avl tree unbalanced, we transform the tree by a rotation. There are 4-types of rotation we have.
apply Right rotation for unbalanced node.

apply left rotation for unbalanced node.

Right rotation for unbalanced node and left rotation for the nearest node.
apply right-left rotation.
Left rotation for unbalanced node and right rotation for the nearest node.

Types of Imbalance due to Insertion and Corresponding “Fixes”

<table>
<thead>
<tr>
<th>cause of imbalance</th>
<th>BF(Y)</th>
<th>BF(RC)</th>
<th>required rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>new node inserted in the right-subtree of Y’s right child (RC)</td>
<td>+2</td>
<td>+1</td>
<td>single-left</td>
</tr>
<tr>
<td>new node inserted in the left-subtree of Y’s right child (RC)</td>
<td>+2</td>
<td>-1</td>
<td>double right-left</td>
</tr>
<tr>
<td>new node inserted in the left-subtree of Y’s left child (LC)</td>
<td>-2</td>
<td>-1</td>
<td>single-right</td>
</tr>
<tr>
<td>new node inserted in the right-subtree of Y’s left child (LC)</td>
<td>-2</td>
<td>+1</td>
<td>double left-right</td>
</tr>
</tbody>
</table>
Example of Insertion of 1, 2, 3, 4, 5, 0, 7, 6 into an AVL Tree

All insertions are **right-right** and so rotations are all **single rotate** from the **right**. All but two insertions require re-balancing:

All insertions are **right-left** and so double rotations take place form **left-right** and right-to **left**
Another example. The insertion sequence is: 50, 25, 10, 5, 7, 3, 30, 20, 8, 15
```c
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 sub-tree
            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) // insert in left sub-tree
                T->left=insert(T->left, x);
            if(BF(T)==2) {
                if(x < T->left->data)
                    T=LL(T);
                else
                    T=LR(T);
            } else {
                T->ht=height(T);
                return(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);
}

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);
}

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);
}

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 * 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);
}
```
Search Operation in AVL tree:

Search operation of AVL tree is same as the search operation of binary search tree. Means given element is checked with the root element,

- If the given element is match with the root element then return the value
- If the given element is less than the root element then the searching operation is continued at left sub-tree of the tree.
- If the given element is greater than the root element then the searching operation is continued at right sub-tree of the tree.

```c
node *search(node *root, int key, node **parent) {
    node *temp;
    temp = root;
    while (temp != NULL) {
        if (temp->data == key) {
            printf("The %d Element is Present", temp->data);
            return temp;
        }
        *parent = temp;
        if (temp->data > key)
            temp = temp->lchild;
        else
            temp = temp->rchild;
    }
    return NULL;
}
```
Deletion of node in AVL tree:
- Deletion:
  - Case 1: if X is a leaf, delete X
  - Case 2: if X has 1 child, use it to replace X
  - Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)

Algorithm:
Step 1: Search the node which is to be deleted.
  - If the node to be deleted is a leaf node then simply delete that node and make be null
  - If the node to be deleted is not a leaf-node, i.e., that node have one or two children then that node must be swapped with its inorder successor. Once the node is swapped we can remove the required node.
Step 2:- Now we have to traverse back up the path towards the root node checking balance factor of every node along that path.
Step 3:- If we encounter unbalancing in some sub tree than balance that sub tree using appropriate single or double rotations.

Delete 55 (case 1)

Delete 50 (case 2)
Delete 60 (case 3)

Delete 55 (case 3)
Delete 50 (case 3)
Delete 40 (case 3)

Delete 40 : Rebalancing

Delete 40: after rebalancing
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); 

<table>
<thead>
<tr>
<th>AVL tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>Invented</strong></td>
</tr>
<tr>
<td><strong>Invented by</strong></td>
</tr>
</tbody>
</table>

| Time complexity in big O notation | 
|-------------------------------|--------|--------| 
| **Space** | Average   | Worst case  | 
|          | O(n)      | O(n)      | 
| **Search** | O(log n)  | O(log n)  | 
| **Insert** | O(log n)  | O(log n)  | 
| **Delete** | O(log n)  | O(log n)  |
B-Tree:
B-Tree is a self balancing search tree.
B-Tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions and deletions in logarithmic time (O(log n)).

Properties of B-Tree:
- The root has at least one key.
- All leaves (external node) are at the same level.
- Keys are stored in non-decreasing order.
- A B-tree of order M is a tree then
  - The root is either a leaf or has between 2 and m Childs.
  - Non-root nodes have at least \( \lceil \frac{M}{2} \rceil \) sub-trees.

Example:

2-3-4 tree:
A B-tree of order 4 is known as a 2-3-4 tree.
A 2–3–4 tree (also called a 2–4 tree) is a self-balancing data structure that is commonly used to implement dictionaries. The numbers mean a tree where every node with children (internal node) has either two, three, or four child nodes:
- a 2-node has one data element, and if internal has two child nodes;
- a 3-node has two data elements, and if internal has three child nodes;
- a 4-node has three data elements, and if internal has four child nodes.

Properties
- Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively.
- All leaves are at the same depth (the bottom level).
- All data is kept in sorted order.
Example:

\[
\begin{array}{c}
10 \ 20 \ 24 \\
5 \ 17 \ 22 \ 29
\end{array}
\]

Insert 25

\[
\begin{array}{c}
10 \ 20 \ 24 \\
5 \ 17 \ 22 \ 25 \ 29
\end{array}
\]

2-3-Tree:
A b-tree of order 3 is known as 2-3-tree.
A 2-3 tree is a tree (B-tree) in which each internal node (non leaf) has either 2 or 3 children and all leaves are at the same level.

Properties of 2-3-tree:
- Every non-leaf is a 2-node or a 3-node. A 2-node contains one data item and has two children. A 3-node contains two data items and has 3 children.
- All leaves are at the same level (the bottom level)
- All data is kept in sorted order
- Every leaf node will contain 1 or 2 fields.

Example:
**Operations on a 2-3 Tree:**

**The lookup operation (Search)**

Recall that the lookup operation needs to determine whether key value \( k \) is in a 2-3 tree \( T \). The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:

1. \( T \) is empty: return false
2. \( T \) is a leaf node: return true iff the key value in \( T \) is \( k \)

And there are 3 recursive cases:

1. \( k \leq T\.leftMax \): look up \( k \) in \( T \)'s left subtree
2. \( T\.leftMax < k \leq T\.middleMax \): look up \( k \) in \( T \)'s middle subtree
3. \( T\.middleMax < k \): look up \( k \) in \( T \)'s right subtree

**Constructing 2-3-tree:**

Example: Construct a 2-3 tree by using the following elements:

\[30, 20, 35, 15, 60, 65, 25, 5, 65, 70, 10, 40, 95\]
Example: Construct a 2-3 tree by using the following elements: 45, 23, 29, 35, 39, 11, 79, 38, 48.
Inserting Items
The goal of the insert operation is to insert key k into tree T, maintaining T's 2-3 tree properties. Special cases are required for empty trees and for trees with just a single (leaf) node.

How do we insert 32?

Final Result
Deleting Items

After deleting an element in the tree (2-3-tree), the resulting tree must be 2-3-tree, means it must the resulting tree must satisfy all the properties of B-tree of order 3.

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Consider the following example for deleting nodes from 2-3-tree.

Deleting 70:

Delete 100

Delete 80:
## Comparisons between binary search tree and 2-3 tree

<table>
<thead>
<tr>
<th></th>
<th>BST</th>
<th>2-3 Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>where are values stored</td>
<td>every node</td>
<td>leaves only</td>
</tr>
<tr>
<td>extra info in non-leaf nodes</td>
<td>2 child ptrs</td>
<td>leftMax, middleMax, 3 child ptrs</td>
</tr>
<tr>
<td>worst-case time for lookup, insert, and delete (N = values stored in tree)</td>
<td>O(N)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>average-case time for lookup, insert, and delete (N = # values stored in tree)</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
</tbody>
</table>

### Binary search tree

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Invented</th>
<th>Invented by</th>
<th>Time complexity in big O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average</td>
</tr>
<tr>
<td>Space</td>
<td>Tree</td>
<td>1960</td>
<td>P.F. Windley, A.D. Booth, A.J.T. Colin, T.N. Hibbard</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

### AVL tree

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Invented</th>
<th>Invented by</th>
<th>Time complexity in big O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average</td>
</tr>
<tr>
<td>Space</td>
<td>Tree</td>
<td>1962</td>
<td>G. M. (A)delson, (V)elskii &amp; E. M. (L)andis</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td></td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

### B-tree

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Invented</th>
<th>Invented by</th>
<th>Space</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average</td>
<td></td>
<td>Average</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Worst case</td>
<td></td>
<td>Worst case</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>