Trees

(Section 6)

Binary trees

Trees and their application

By:

Pramod Parajuli,
Department of Computer Science,
St. Xavier’s College,
Nepal.

Binary Trees

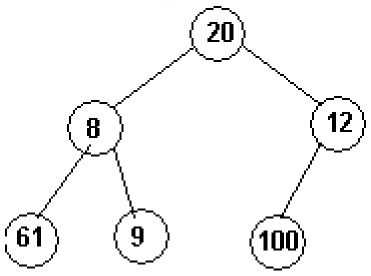
Definition

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees, called the left and right sub-trees of the original tree. A left or right sub-tree can be empty. Each element of a binary tree is called a node of the tree.

- Langsham, Augenstein, Tanenbaum, *Data Structures using C and C++*

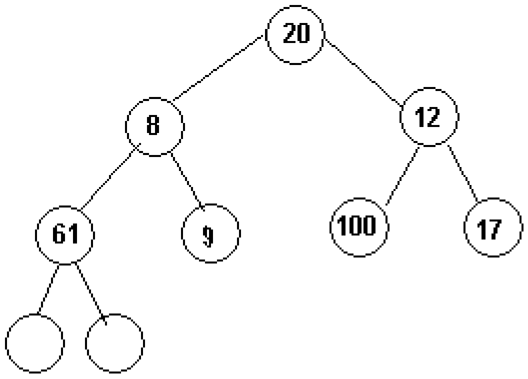
Simple binary tree

The binary tree that uses the definition.



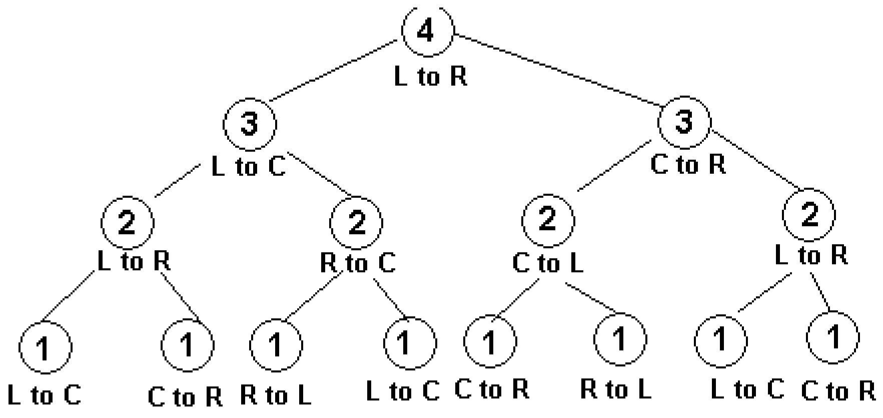
Strictly binary tree

All of the nodes either have two children or none.



Complete binary tree

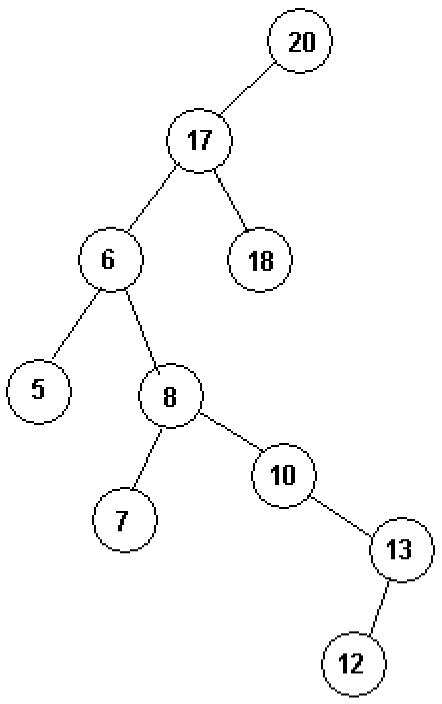
Height of left hand and right hand of root node must be same. The tree for TOH solution is a complete binary tree.



Binary search tree

It is a simple binary tree with following property;

All the nodes in the right sub-tree of a node contain greater values and all the nodes in the left sub-tree contain lower values than a given node. The left and right sub-trees themselves are also binary search tree.



Leaf node

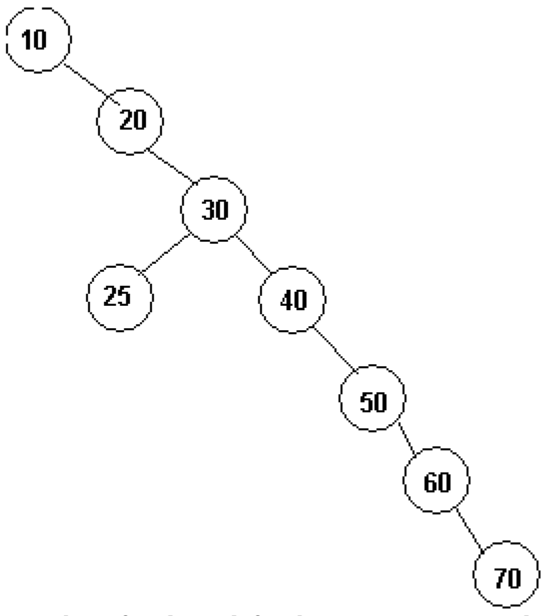
Nodes that contain no child is called leaf node. In above figure, nodes containing 5, 7, 12, 18 are leaf nodes.

Internal node

Nodes that contain at least one child except root node is called internal node.
e.g. 17, 6, 8, 10, 13

Right/left skewed tree

If the inclination of nodes in the right hand side is high, then it is known as right skewed tree.



Similarly, if the number of nodes in left sub-tree is comparatively high enough, then it is known as left skewed tree. Both of these are called unbalanced tree. In an unbalanced tree, the average searching time is high.

Balanced Tree

A tree in which no. of nodes in left sub-tree nearly equal to nodes in right sub-tree.

Level of node

Number of steps to come to the node from the root.

Height of a node

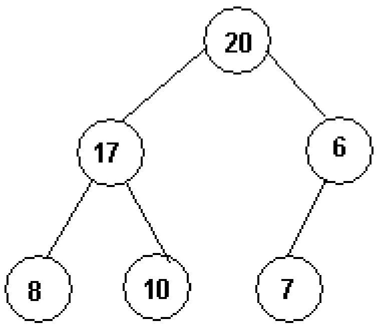
Number of steps to go to the NULL child from the node.

Operations and applications

Constructing binary trees

Let's say, we have data like; 20, 17, 6, 8, 10, 7, 18, 13, 12

Just start making new node at left hand side and fill towards right. After filling, go to the next level.



Constructing binary search trees

While constructing binary search trees, all the nodes in the tree must fulfill BST property.

For first node:

A single node is created. This node contains NULL as children in both sides. This will be root node for the binary tree.

For second and other:

We start from the root node. The value to be inserted is compared with the root node. If it is lower than the current node (root node), then left sub-tree is traversed. If it is higher than the current node (root node), then right sub-tree is traversed. This is done for each and every node encountered on the way and traversal is done until a NULL value is not found. If NULL value is found, then that will be the exact position to insert new node.

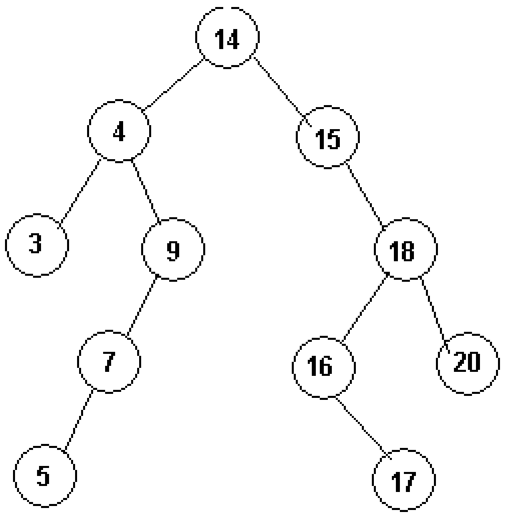
e.g. Classwork;

Removing nodes from binary search trees

While deleting a node in BST, there are three cases:

1. For leaf node - Set NULL to its parents left/right.
2. Node with one child - Redirect address of its (node being deleted) address to parent's left/right.
3. Node with two children - Replace the node being deleted by leftmost node of right sub-tree or rightmost node by left sub-tree.

Example:



Delete 15, 9, 16 in order.

Classwork;

Form binary search tree of

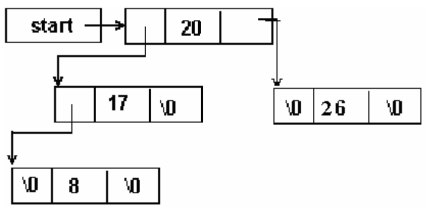
10, 45, 26, 1, 23, 56, 43, 76, 64, 88, 8, 2, 6, 7 , 245, 98, 63, 69

And delete the numbers 43, 56, 1, 245, 63 in order.

Binary tree representation

In binary trees, a node contains information about three things:

- left sub-tree
- the data
- right sub-tree



```
struct node{
      struct node *left;
      int info;
      struct node *right;
};
```

Presentation 3: Array representation of binary tree

Binary tree traversals

A given binary tree can be of any size. The number of nodes is not known at the coding time as the nodes are created and deleted in run-time. Further, the property of binary tree is recursive. Therefore, the tree traversal process can be done using recursive procedure.

Preorder Traversal (VLR)

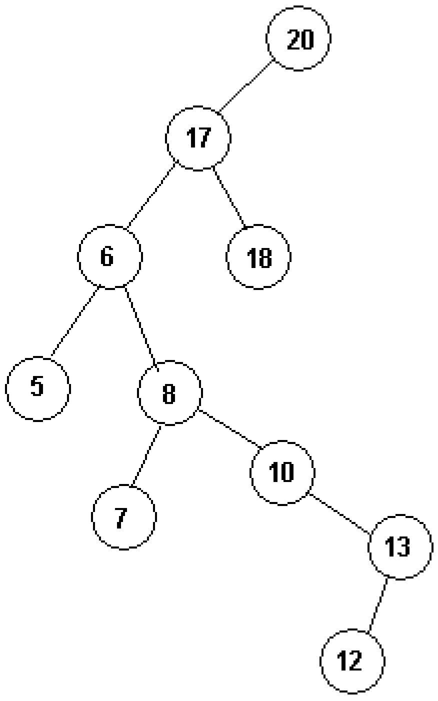
1. Visit the root
2. Traverse left sub-tree in pre-order
3. Traverse right sub-tree in pre-order

In-order traversal (LVR) **(produces ascending output)**

1. Traverse left sub-tree in in-order
2. Visit the root
3. Traverse the right sub-tree.

Post order traversal (LRV)

1. Traverse the left sub-tree
2. Traverse the right sub-tree
3. Visit the root



Preorder (VLR): 20, 17, 6, 5,

After finding the end or leaf node then start right traverse.

8, 7, 10, 13, 12, 18

Inorder (LVR): 5, 6, 7, 8, 10, 12, 13, 17, 18, 20
Postorder (LRV): 5, 7, 12, 13, 10, 8, 6, 18, 17, 20

Huffman algorithm

Lots of compression systems use Huffman algorithm for the compression of text, images, audio, data, etc. The general idea behind the Huffman algorithm is that, use less number of bits to represent mostly used key.

Example,

Let's say we get four kind of signal in a system.

A B C D.

For such system, we might use 2-bit representation as;

A	00
B	01
C	10
D	11

and we have data string as,

AAABCAABBCDDAAAABAAABABABADDAAAABC

If the number of A is 10,000, B is 234, C is 45, D is 134, and then the total storage required will be;

(10,000 + 234 + 45 + 134) * 2 = 20,826 bits to store.

Since the data 'A' is repeated ten thousand times, if the storage for A can be reduced, then the total space required will automatically be reduced.

Therefore, in Huffman algorithm, the data (symbol) that is used most is represented by a single bit.

Now, while assigning the code word, the most frequently used symbol is assigned a code word of length ‘1’, and the final one always gets code word with all ones.

Let’s assign codeword for the symbols as;

A	0
B	10
C	110
D	111

Now, for the given data string, the binary value will be,
000101100010101101111110000100001001001001111111000010110

Now, if we calculate the total space required to store the data, it will be:

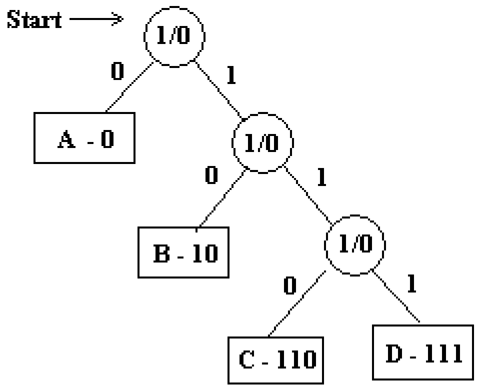
(10,000 * 1 + 234 * 2 + 45 * 3 + 134 * 3) = 11,005 bits.

So we saved 20,826 – 11,005 = 9821 bits

If the data to be stored is very large, then more bits will be saved.

Now, the problem is, how do we recover the original data from compressed string;
000101100010101101111110000100001001001001111111000010110
Here, a tree is made according to the codeword assignment.

For our example, we have to make a tree like;



Now, to decode the encoded string, read a bit at a time. Compare with root node. If it is 0, then you are at the end decision point. So, it must be ‘A’. If it was 1, then you again have to read next bit and check. If it was 0, then it is ‘B’. Otherwise, read next bit and check until you do not reach decision point.

Functions

Adding node in BST

To add a node in BST, we search a proper place by traversing either left or right depending on the comparisons between the value and value of nodes.

```
int add(node *startNode, int value){
      node *tempNode;
      node *previousNode;
      tempNode = startNode;
      previousNode = startNode;

      // if the starting node is NULL

      if(startNode == NULL){
            startNode = createNode();
            startNode -> data = value;
            startNode -> left = NULL;
            startNode -> right = NULL;
            return SUCCESS;
      }

      // traverse until a NULL node i.e. empty leaf node is not found

      while(tempNode != NULL){
            previousNode = tempNode;
            // if the value is less, traverse left
            if(value < tempNode -> data){
                  tempNode = tempNode -> left;
            }
            // if the value is greater, traverse right
            else if(value > tempNode -> data){
                  tempNode = tempNode -> right;
            }
            // if equal, just return
            else if(value == tempNode -> data){
                  return SUCCESS;
            }
      }

      // if the value is greater than parents' data, then put on the
      // right side of the node.

      if(previousNode -> data < value){
            tempNode = createNode();
            previousNode -> right = tempNode;
            tempNode -> left = NULL;
            tempNode -> right = NULL;
            tempNode -> data = value;
      }

      // if the value is less than parents' data, then put on the left
      // side of the node.

      else if(previousNode -> data > value){
            tempNode = createNode();
            previousNode -> left = tempNode;
            tempNode -> left = NULL;
            tempNode -> right = NULL;
            tempNode -> data = value;
      }

      return SUCCESS;
}
```

Removing node from BST

Deletion of a node is quite tricky. To delete a node, first we must find the position of the node. After that, we sit on the parent node of the deleting node and then, search either for leftmost node in right sub-tree or rightmost node in left sub-tree. Then we backup the value of the leftmost or rightmost node and delete that. The deletion will be either for leaf node or for a node with one child only. Then the value of originally deleting node will be replaced by the backup value. And we are done.

```
int delete(node *startNode, int value){
	node *tempNode;
	node *nextNode, *parentNode, *tempParent, *tempLeft;
	tempNode = startNode;
	previousNode = startNode;

	// as said, we must search for the node that contains the value.
	// we search until, we do not find the node with the given value
	// or we encounter with NULL i.e. end of the tree.  It’s because,
	// the tree may or may not contain node with given value.

	while(tempNode !=  NULL && value != tempNode -> data){

		// now, just look, we must take care of the parent node
		// while deleting.  Therefore, for each and every traversal
		// we backup the address of parent node.
		parentNode = tempNode;

		if(value < tempNode -> data)
			tempNode = tempNode -> left;
		else
			tempNode = tempNode -> right;
	}

	// at this point, we encounter either with NULL or with the right
	// node

	// so, let’s check whether we encountered with NULL or not?
	// if it is, then the tree do not contain a node with appropriate
	// value.  In such case, we display an error message

	if(tempNode == NULL){
		printf(“The tree do not contain the search key.”);
		return DEL_INVALID;
	}

	// if we do not encounter with NULL, we have the appropriate
	// node with given value

	// at this point, the node exists, we also have information about
	// parent node in ‘parentNode’. Now, we check the number of
	// children.

	// if the node do not contain any child i.e. it is a leaf node
	// then just delete the node

	// again another trick:
	// if the left child is NULL, then just replace with right node,
	// and if the right child is NULL, then just replace with left
	// node

	if(tempNode -> left == NULL){
		nextNode = tempNode -> right;
	}
	else if(tempNode -> right == NULL){
		nextNode = tempNode -> left;
	}
```

```
        else            // the node contains two children
        {
                // for the next search again, we need to hold information
                // of parent nodes
                tempParent = tempNode;

                nextNode = tempNode -> right;

                tempLeft = nextNode -> left;

                while(tempLeft != NULL){
                        tempParent = nextNode;
                        nextNode = tempLeft;
                        tempLeft = nextNode -> left;
                }

                // here, nextNode is parent of tempNode

                // now, just swap the values

                if(tempParent != tempNode){
                        tempParent -> left = nextNode -> right;
                        nextNode -> right = tempNode -> right;
                }
        }

        if(parentNode == NULL){
                startNode = nextNode;
        }
        else if(tempNode == parentNode -> left)
                parentNode -> left = nextNode;
        else
                parentNode -> right = nextNode;

        free(tempNode);

        return SUCCESS;
}
```

Recursive In-order traversal

```
trav_in_order(struct tree *node){
        if(node == NULL){
                return;
        }

        trav_in_order(node -> left);
        printf("\t%d", node -> info);
        trav_in_order(node -> right);
}
```

Classwork: Write recursive function for pre-order and post-order traversal.

Printing terminal nodes of a BST

```
trav_in_order(struct tree *node){
        if(node == NULL)
                return;
        trav_in_order(node -> left);
        if(node -> left == NULL && node -> right == NULL)
                printf("\t%d", node -> info);
        trav_in_order(node -> right);
}
```

Printing internal nodes of a BST
```
if(node -> left == NULL || node -> left == NULL){
      printf("\t%d", node -> info);
}
```

Printing nodes with one child
```
if((node -> left != NULL && node -> right == NULL) ||
      (node -> left == NULL && node -> right != NULL){
            printf("\t%d", node -> info);
}
```

Finding minimum value
```
Presentation – 4
```

Finding maximum value
```
Presentation - 4
```

Applications

Sorting of data (alphabetic/numeric)

To sort any set of given data, a binary tree is formed using the elements and the tree is traversed in LVR order to get ascending order and RVL traversal generates elements in descending order.

Class work: Form a BST of following data and read elements in RVL order.

40, 3, 534, 456, 2, 46, 7, 8, 23, 34, 17, 843, 5, 7, 54, 34, 29, 71, 1

Use of expression tree

Construction of an expression tree from post-fix expression

Algorithm:

```
1.    read symbol at a time
2.    if symbol is an operand
      Create one node tree and push the pointer to it onto a stack.

3.    if symbol is an operator
      pop pointers of two trees T2 & T1 from stack & form a new tree
      whose root is the operator and whose left & right children point
      to T1 & T2 respectively.
```

Ex: Construct expression tree and show traversals.

a b + c d e + * *

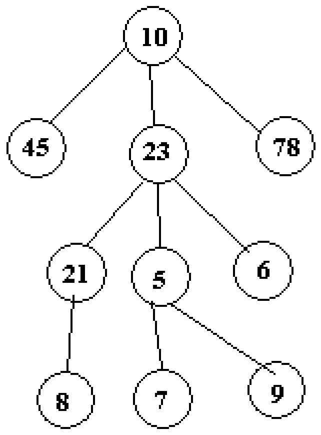
Pre-order (VLR)

In-order (LVR)

Post-order (LRV)

General Trees

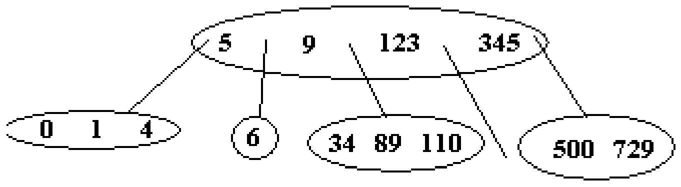
General trees can contain as many children as you like. Some of the constraints are same as binary tree and some are different. For example, the elements must be placed so that it follows binary tree rule. One possible implementation is B-tree. Other general purpose trees might even does not follow any standard at all.



General Tree

One possible use of such tree might be to represent sets
Given tree represents: (10, 45, 23, (21, 8), (5, 7, 9), 6), 78).

A B-tree might contain more than one value in a node.



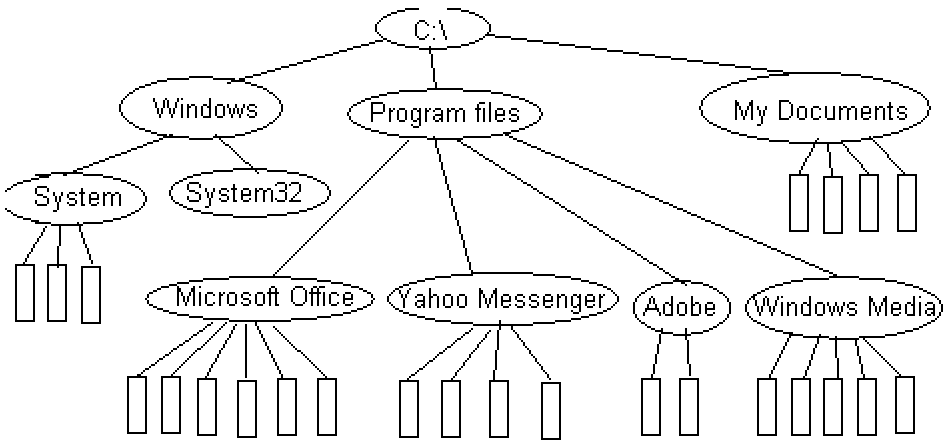
B-tree

All of the operating system use B-trees to manage file system in magnetic disks.

File system implementation

All of the file management routines in operating system manage the file system using trees.

Every file system contains a 'root' of the file system. For example, in Microsoft® Windows, the drive letters like A:, B:, C:, are the root of the file system. The directories in the root of the file system are organized as general tree.



Similar concepts are used in other file systems.

Game Trees

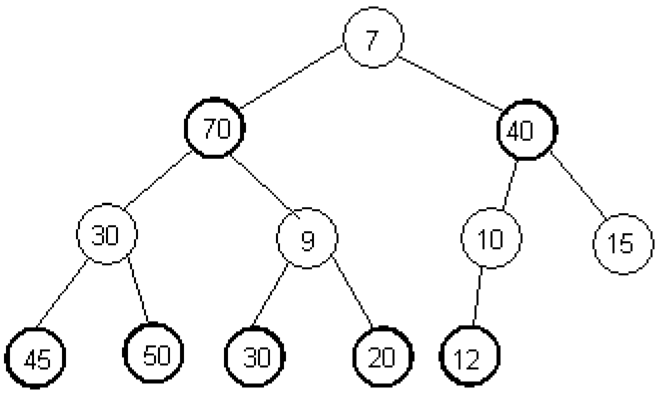
Game tree is most popular algorithm/structure for determining the best move for a player during a game.

- nodes of game tree represent possible game positions
- children of a node represent positions that can be arrived at via one move from the parent position

Example

General idea is to increase the game winning possibility for one player.

Let's see how the winning strategy is applied.



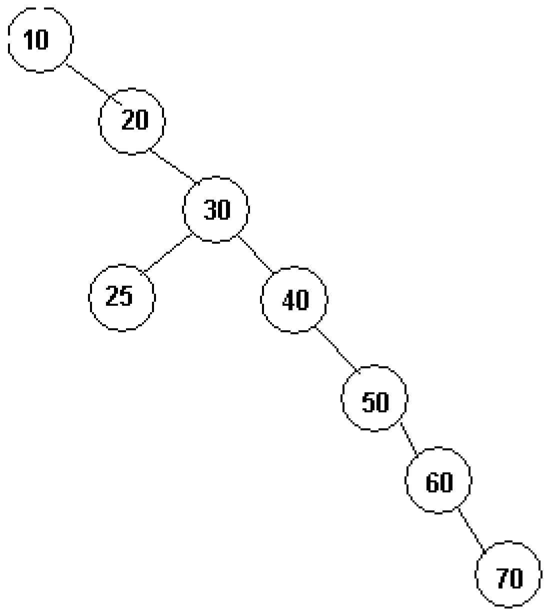
The tree here is known as min-max tree. Here, we search for best move.

The best move for this time is calculated by analyzing the moves in further levels. For example, in given example, if 7 is your turn, then your opponent will definitely choose 70, as it can lead up to 50, the maximum value at 4th level. Now, you strategy is that, if he chooses 70, then do not choose 30, because it will lead to 50. Therefore, choose 9. After level 4, you might get chance of winning the game.

Presentation 5: Game tree of tic-tac-toe.

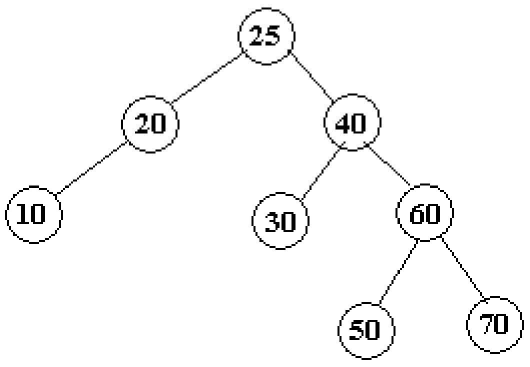
Performance issue

Let's consider, we have a right skewed binary search tree as shown below.



To search 10, 20, 30, 25 i.e. the nodes closer to root node, the search time is not much significant. But, to search 60, 70, we need to compare 6 and 7 times respectively which is roughly double than that of 30, 25, 40. Though the concept of binary search tree is good for searching, the performance drops drastically, if the tree is a skewed tree.

With the same data set, 10, 20, 30, 25, 40, 50, 60, 70, we can make a binary search tree in which searching time for all of the nodes is not too much different.



Now, notice that, the maximum number of comparisons is only 4. One of the most popular technique to change the tree structure properly so that it holds BST property and search time is also optimized is to implement AVL tree.

AVL tree

AVL tree was defined by Avel-son Vel & Landis. It is a binary search tree with following properties:

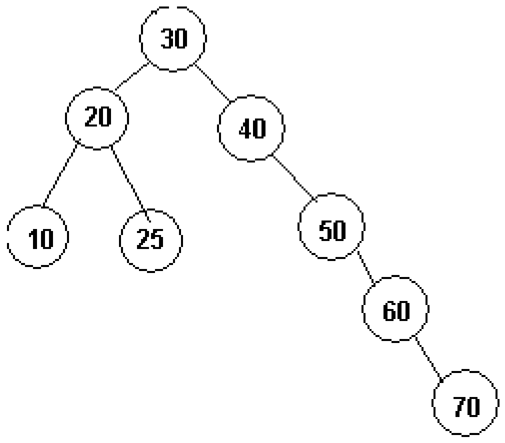
i. height of left sub-tree of root & height of right differs at most by 1.
ii. left & right sub-trees are again AVL trees.

More specifically;

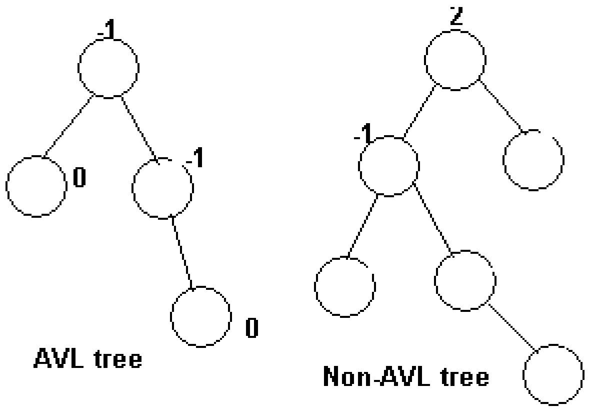
Balance factor for a node (BF) = left height – right height

BF of nodes in an AVL tree should be 1, 0, or -1. If other than these, then it is not an AVL tree.

Class demonstration: Calculate balance factor for all of the nodes in following tree.



More examples:



Balancing techniques

A BST can be converted to AVL tree by using balancing techniques i.e. after formation of BST. But generally, AVL trees are formed as the data is being added in the tree and also while deleting a node. However, the method to balance the tree is both of the cases is same.

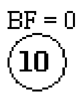
To form an AVL tree with given set of data, we use two basic balancing techniques:

1. Right rotation
2. Left rotation

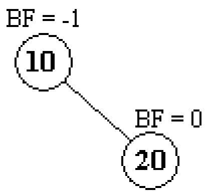
How do we apply these techniques? Let's say, I have some data like,

10, 20, 30, 25, 27, 7, 4, 23, 26, & 21.

10 - make a root node and calculate balance factor.

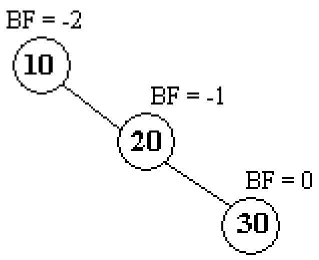


20 – add according to BST rule and calculate balance factor for all of the nodes.



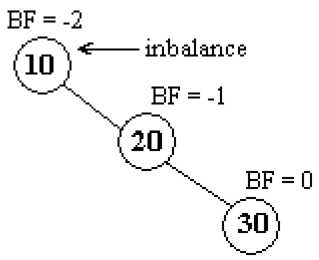
Till this point, the tree holds AVL property.

30 – do same as for 20

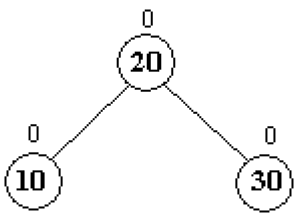


Notice that, after adding 30, the tree is not longer an AVL tree. It is due to balancing factor of root node which is -2.

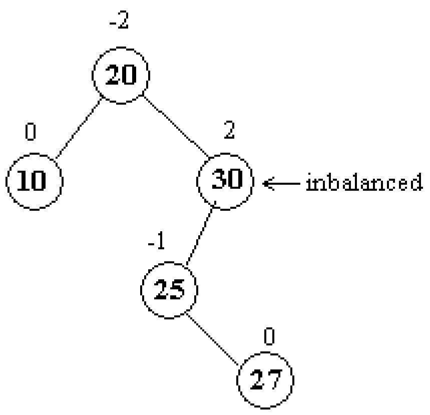
To indicate the inbalance, we put a 'inbalance' mark;



The root node is inbalanced due to right skewing of nodes. So, in this case, we rotate the inbalanced node towards left (also called as left rotation).



Similarly, let's add 25 and 27;



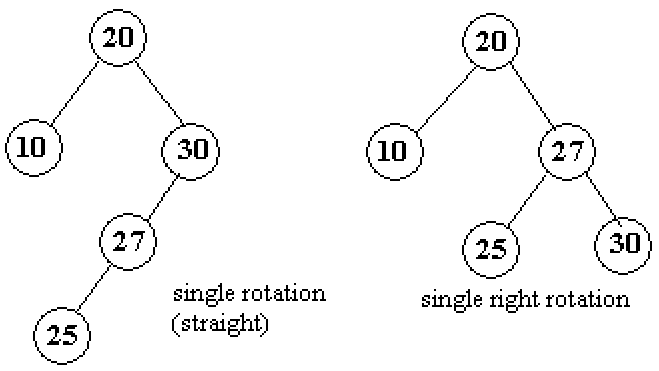
In the resulting tree, there are two inbalanced nodes (node containing 20, and 30). So now, how do we balance the tree in such condition?

To apply balancing techniques, we always start from the leaf and go up. It mean, to balance the tree, we first deal with node containing 30, and then we will see others.

Again, in previous example of inbalnaced nodes, the pattern of nodes was straight. But in this case, the nodes are in zig-zag fashion. 25 is on the left hand side, and 27 is again in right hand side of 27. This pattern is known as '**dog-leg**' pattern. In case of dog-leg pattern, we use, **double rotation**.

Steps for double rotation:

i. Leave the inbalanced tree as it is.
ii. If the child is in right, then make a straight pattern on the right hand side by using single rotation. Do the same if the child is in left hand side.
iii. Now, make single right/left rotation including the inbalanced node



Double rotation for dog-leg pattern

Do you notice that, after balancing 30, the root node (20) automatically got balanced? In other cases, it might not be so.

Let's finish adding the nodes and balancing. Class demonstration

As said already, not just while adding nodes, the AVL property should be maintained after deletion of nodes also.

Let's delete 27, 25, 7 in order maintaining AVL property.

Class demonstration.

Homework 5: (Submission date: April 16 – 2004)

1. **Construct expression tree from**

a b c * - d e * f + g * +

and show result of pre, in, and post-order traversals.

2. **Construct AVL tree with following data**;

10, 45, 26, 1, 23, 56, 43, 76, 64, 88, 8, 2, 6, 7, 245, 98, 63, 69

and then delete nodes with values 26, 7, 63, 76, 10.