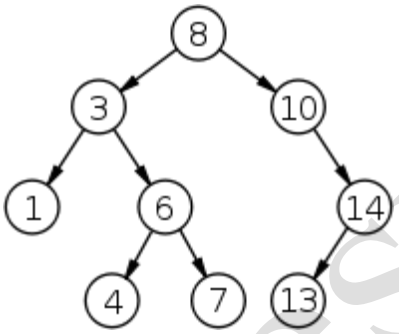
Binary Search Trees:

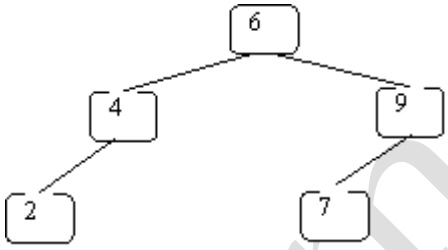
An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node **n** in the tree:

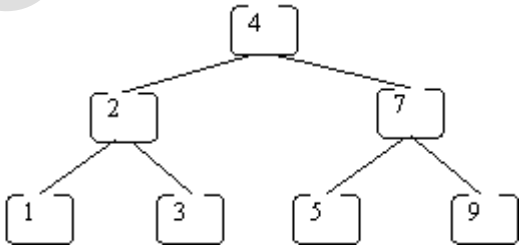
- All keys in n's **left subtree are less** than the key in **n**, and
- All keys in n's **right subtree are greater** than the key in **n**.

In other words, binary search trees are binary trees in which all values in the node's left subtree are less than node value all values in the node's right subtree are greater than node value.

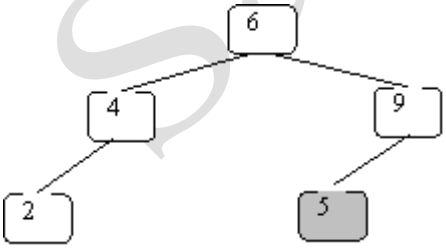


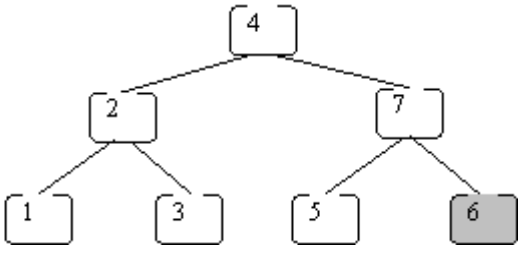
Here are some more examples on BSTs in which each node just stores an integer key:





These are not BSTs:

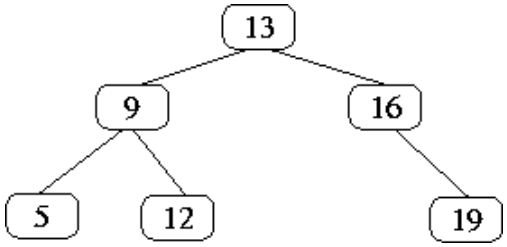




In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.
The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

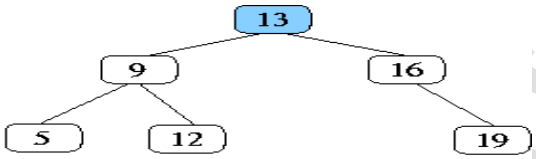
- insert a key value
- determine whether a key value is in the tree or not
- Delete a key value from the tree
- print all of the key values in sorted order

Let's illustrate what happens using the following BST:

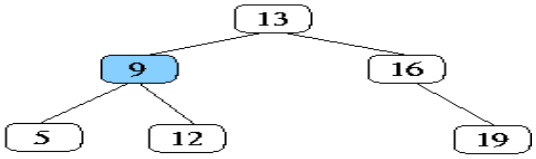


Searching for 12:

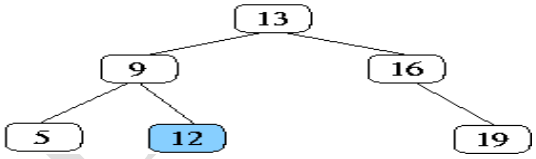
12 < 13 so go to left subtree



12 > 9 so go to right subtree.

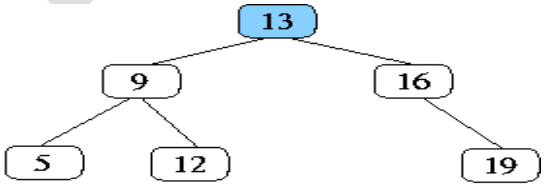


found!

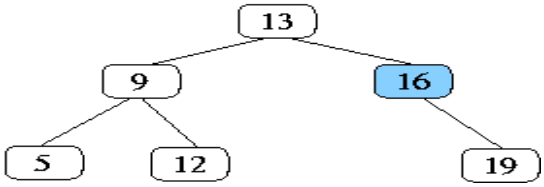


What if we search for 15:

15 > 13 so go to right subtree



15 < 16 so go to left subtree. It does not exist so search fails and it returns false



Properties and Operations:

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. **Duplicate node keys are not allowed in BST.**

Inserting a node

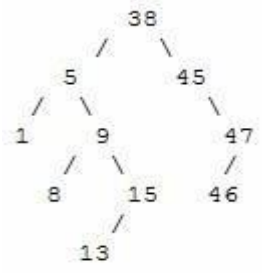
A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Searching for a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.
Search should return a true or false, depending on the node is found or not.

Deleting a node

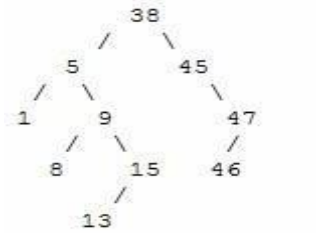
A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



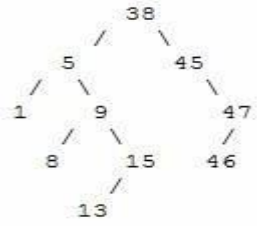
Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node?

What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

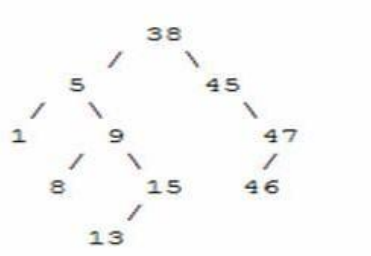
Case 1 : The node to delete is a leaf node
This is a very easy case. Just delete the node 46. We are done



Case 2 : The node to delete is a node with one child.
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.

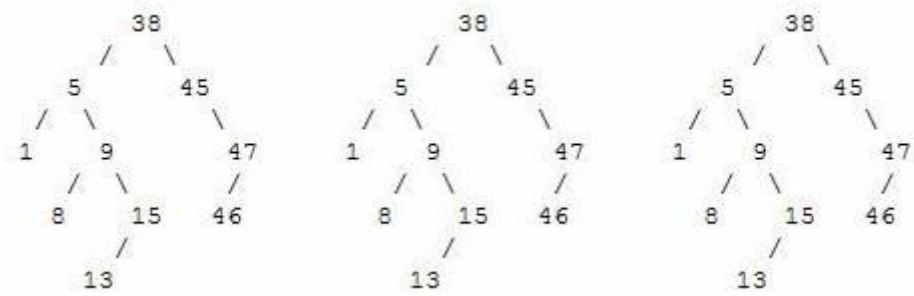


Case 3: The node to delete is a node with two children
This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)
Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.

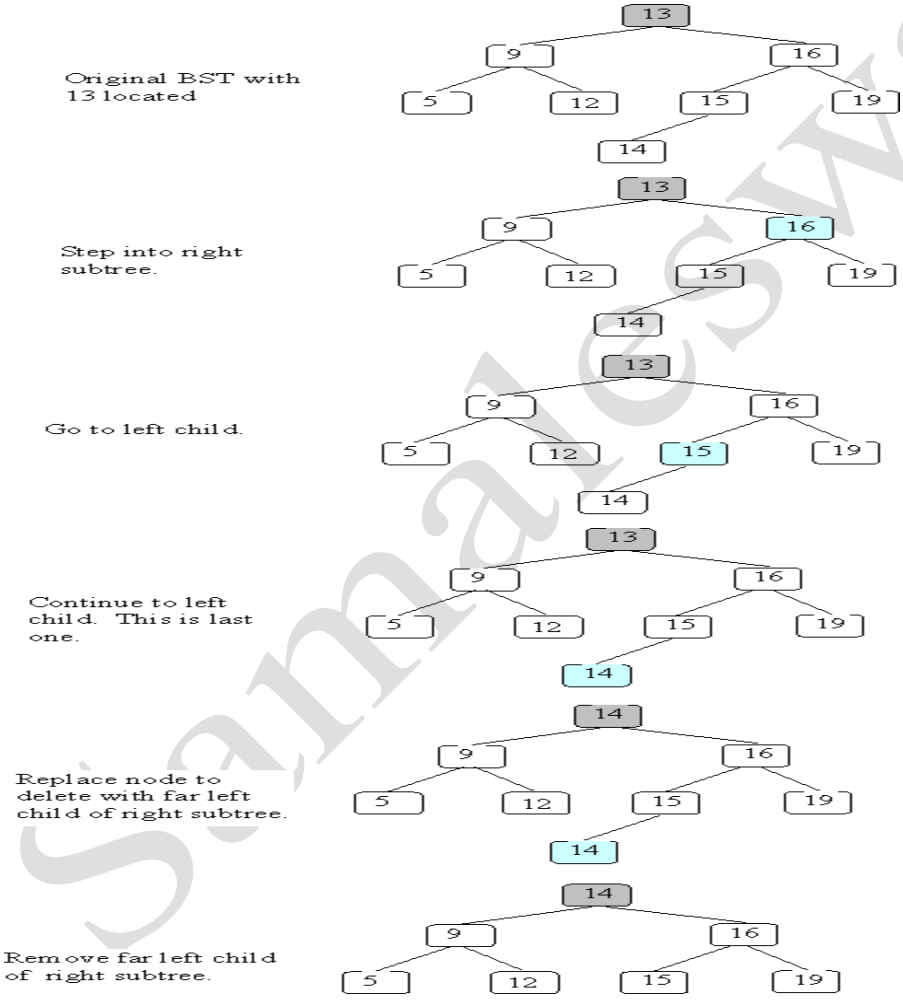


Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree

(45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.



Let's see when we delete 13 from that tree.



NB: This delete can also be done by substituting with one left of 13 then extreme right node i.e. 12. Then you have to delete that repeated node 12.

Q:- Write a menu driven program to implement all operations of BST.

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

struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
};

void insert();
void inorder(struct node *);
void preorder(struct node *);
void postorder(struct node *);
void smallest();
void largest();
void delete();
struct node *root=NULL;

main()
{
        int choice,option;
        while(1)
        {
                printf("\n1.Insert\n2.Delete\n3.Display\n4. Smallest\n5. Largest\n6. EXIT\n");
                printf("Enter your choice");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: insert();
                                        break;
                        case 2: delete();
                                        break;
                        case 3: printf("\n1.Preorder\n2.Inorder\n3.Postorder\n");
                                        printf("Enter your choice for display of result");
                                        scanf("%d",&option);
                                        switch(option)
                                        {
                                                case 1: preorder(root);
                                                                break;
                                                case 2: inorder(root);
                                                                break;
                                                case 3: postorder(root);
```

```
                                                        break;
                                }
                                break;
                        case 4: smallest();
                                break;
                        case 5: largest();
                                break;
                        case 6: exit(0);
                        default: printf("\nWrong Choice\n");
                }
        }
}

void insert()
{
        struct node *temp,*ptr,*par;
        int item;
        temp=(struct node *)malloc(sizeof(struct node));
        if(temp==NULL)
        {
                printf("\n Memory is not allocated\n");
        }
        else
        {
                printf("Enter value to insert");
                scanf("%d",&item);
                temp->info=item;
                temp->lchild=NULL;
                temp->rchild=NULL;
                if(root==NULL)
                {
                        root=temp;
                }
                else
                {
                        ptr=root;
                        while(ptr!=NULL)
                        {
                                par=ptr;
                                if(item < ptr->info)
                                {
                                        ptr=ptr->lchild;
                                }
                                else if(item > ptr->info)
                                {
                                        ptr=ptr->rchild;
```

```
                                }
                                else
                                {
                                        printf("\nDuplicate element is not allowed\n");
                                        break;
                                }
                                if(ptr == NULL)
                                {
                                        if(item < par->info)
                                        {
                                                par->lchild = temp;
                                        }
                                        else
                                        {
                                                par->rchild = temp;
                                        }
                                }
                        }
                }
        }
}

void smallest()
{
        struct node *ptr;
        int item;
        if(root == NULL)
        {
                printf("\nTree is empty\n");
        }
        else
        {
                ptr=root;
                while(ptr->lchild != NULL)
                {
                        ptr=ptr->lchild;
                }
                printf("\nSmallest node =%d",ptr->info);
        }

}
void largest()
{
        struct node *ptr;
        int item;
        if(root == NULL)
```

```
	{
		printf("\nTree is empty\n");
	}
	else
	{
		ptr=root;
		while(ptr->rchild != NULL)
		{
			ptr=ptr->rchild;
		}
		printf("\nSmallest node =%d",ptr->info);
	}

}
void inorder(struct node *ptr)
{
	if(ptr!=NULL)
	{
		inorder(ptr->lchild);
		printf("%d ",ptr->info);
		inorder(ptr->rchild);
	}
}
void preorder(struct node *ptr)
{
	if(ptr!=NULL)
	{
		printf("%d ",ptr->info);
		preorder(ptr->lchild);
		preorder(ptr->rchild);
	}
}

void postorder(struct node *ptr)
{
	if(ptr!=NULL)
	{
		postorder(ptr->lchild);
		postorder(ptr->rchild);
		printf("%d ",ptr->info);
	}
}
void delete()
{
	struct node *ptr,*par;
	int item;
```

```
if(root == NULL)
{
        printf("\nTree is empty\n");
}
else
{
        ptr=root;
        par=NULL;
        printf("\nEnter value to delete\n");
        scanf("%d",&item);
        while(ptr != NULL)
        {
                if(ptr->info == item)
                {
                        break;
                }
                par=ptr;
                if(item < ptr->info)
                {
                        ptr=ptr->lchild;
                }
                else
                {
                        ptr=ptr->rchild;
                }
        }
        if(ptr == NULL)
        {
                printf("\n Node not found\n");
        }
        else
        {
                printf("\n Deleted node=%d\n",ptr->info);
                if((ptr->lchild != NULL) && (ptr->rchild != NULL))
                {
                        case_C(par,ptr); //Node having both children
                }
                else if((ptr->lchild == NULL) && (ptr->rchild == NULL))
                {
                        case_A(par,ptr); //Node having zero child
                }
                else
                {
                        case_B(par,ptr); //Node having one child
                }
        }
```

```
        }     }

void case_A(struct node *par, struct node *ptr)
{
        if(par == NULL)
        {
                root=NULL;
        }
        else if(ptr->info < par->info)
        {
                par->lchild=NULL;
        }
        else
        {
                par->rchild=NULL;
        }
        free(ptr);
}
```

Assignment

Complete the code of

1) void case_B(struct node *par, struct node *ptr)
2) void case_C(struct node *par, struct node *ptr)

The only way to be a good programmer is through love programming. Programs can be written through examples not through eyes.

With Regards
Yours Sam Sir