BINARY SEARCH TRESS

(Prof.) Dr. Aparna Kumari

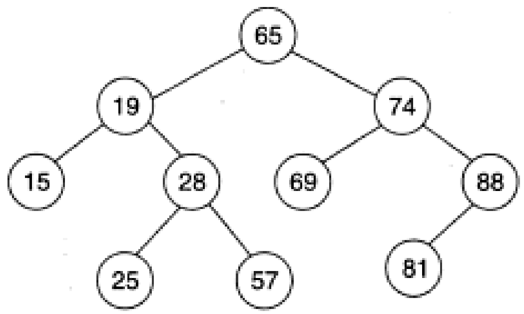
CSE Dept.,

ICT-Ganpat University

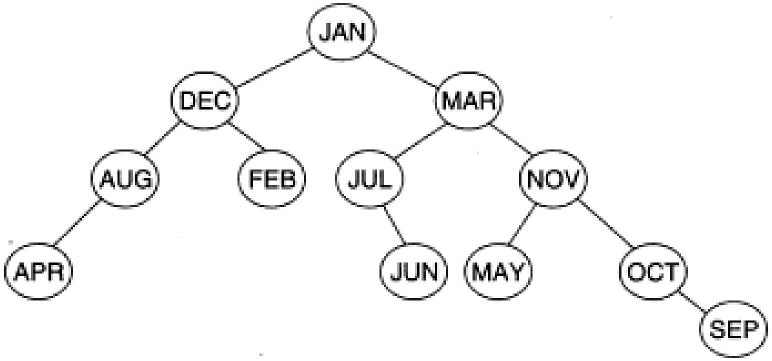
abk03@ganpatuniversity.ac.in

Binary Search Tree (BST)

- A binary tree T is termed binary search tree (or binary sorted tree) if each node N of T satisfies the following property:
  - The value at N is greater than every value in the left sub-tree of N
  - The value at N is less than every value in the right sub-tree of N
- Binary search tree operations:
  - Searching
  - Inserting
  - Deleting
  - Traversing



(a) A binary search tree with numeric data



(b) A binary search tree with alphabetic data

Binary Search Tree: Creation

```
void insert(int data) {
   struct node *tempNode = (struct node*)
malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
```

```
while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
         current = current->leftChild;

         //insert to the left
         if(current == NULL) {
            parent->leftChild = tempNode;
            return;
         }
      }  //go to right of the tree
      else {
         current = current->rightChild;

         //insert to the right
         if(current == NULL) {
            parent->rightChild = tempNode;
            return;
         }
      }
    }
  }
}
```

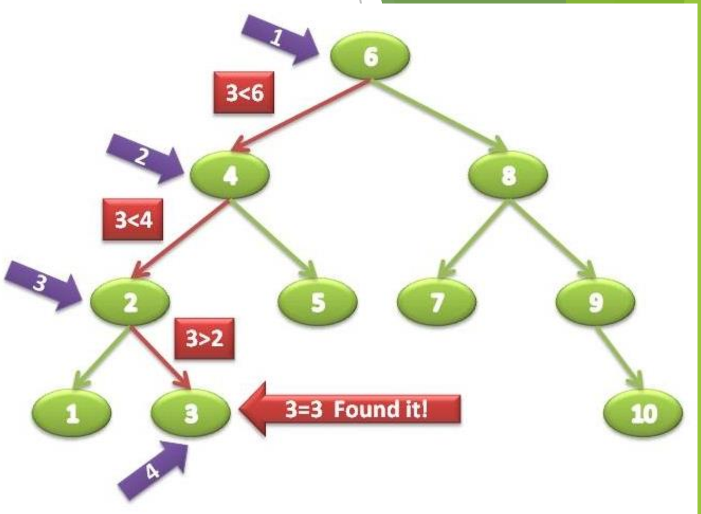
Exercise: Binary Search Tree

Create BST from the below list of elements

```
int main() {
   int i;
   int array[7] = { 20, 15, 30, 10, 19, 21, 42 };

   for(i = 0; i < 7; i++)
      insert(array[i]);
}
```

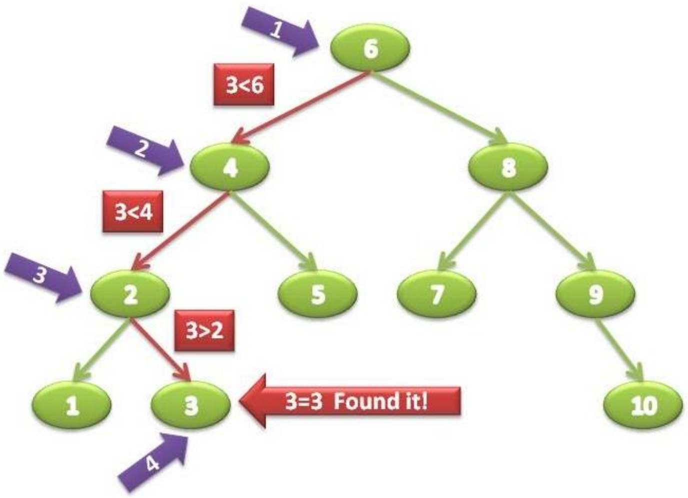
Searching in BST

```
1. [Tree Empty?]
If ROOT == NULL
        Then Write ('Empty Tree')
        Return

2. [Initialize pointer ptr and flag]
ptr ← ROOT, flag ← FALSE                          //Start from root node

3. [Traverse and search]
Repeat while (ptr!= NULL) and (flag == FALSE)
        If ITEM == INFO (ptr)                     //Search successful
                Then flag ← TRUE
                Exit
        If ITEM < INFO (ptr)                      //Go to left sub-tree
                Then ptr ← LPTR (ptr)
        Else
                If ITEM > INFO (ptr)              //Go to right sub-tree
                        Then ptr ← RPTR (ptr)

4. [Search successful or not?]
If (flag == TRUE)
        Then Write ('ITEM found at node', ptr)
Else
        Write ('ITEM does not exist')

5. [Finished]
```



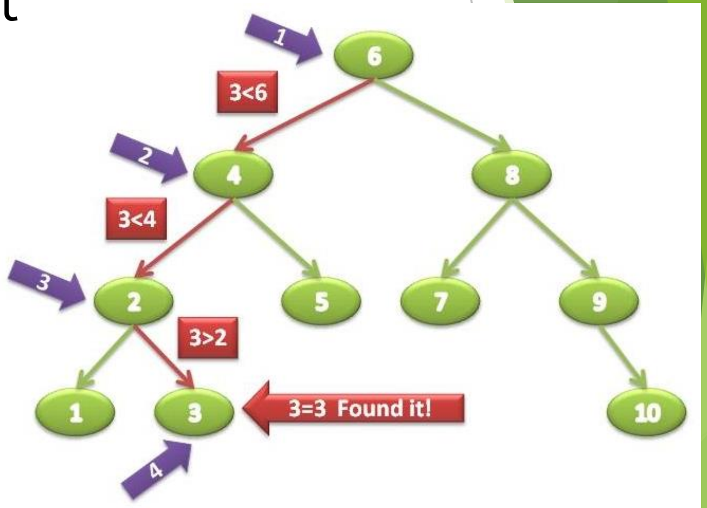
Searching in BST

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
         printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }
   }

   return current;
}
```



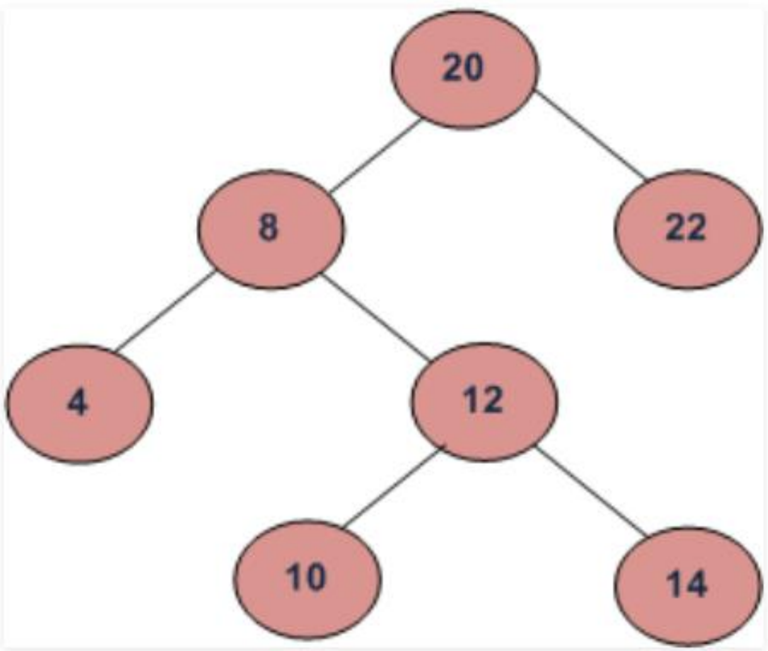
Searching in BST (Recursion)

```
// C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
   // Base Cases: root is null or key is present at
root
   if (root == NULL || root->key == key)
      return root;

   // Key is greater than root's key
   if (root->key < key)
      return search(root->right, key);

   // Key is smaller than root's key
   return search(root->left, key);
}
```



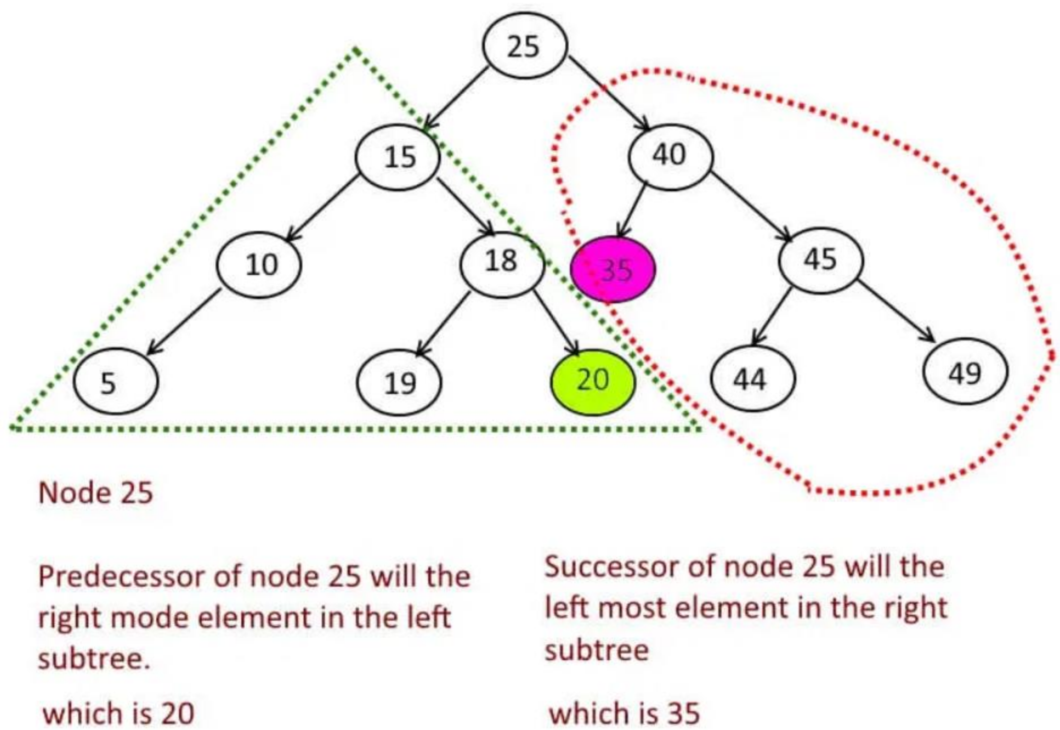
Inorder Successor in BST

- In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.
- In BST, Inorder Successor of an input node can be a node with the smallest key greater than the key of the input node.
- Inorder successor of **8** -> **10**, inorder successor of **10** -> **12** and inorder successor of **14** -> **20**.

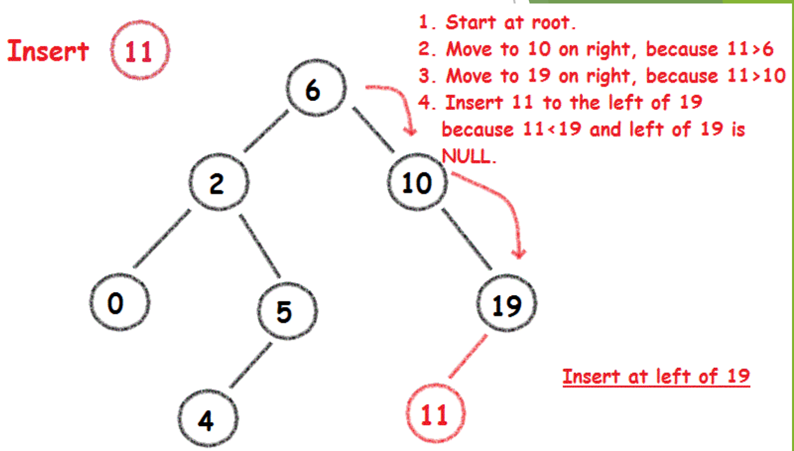


Inorder Predecessor in BST

- *In inorder traversal of a binary tree, th*
  neighbors of given node are
  called ***Predecessor****(the node lies behin*
  of given node).
- In BST, Inorder ***Predecessor*** of an inpu
  node can be a node with the biggest
  key lesser than the key of the input
  node.
- Inorder Predecessor of **25** -> **20**,
  inorder Predecessor of **15** -> **10**



Insertion in BST



1. [Initialize pointer ptr and flag]
ptr ← ROOT, flag ← FALSE   //Start from the root node

2. [Traverse and search]
Repeat while (ptr!= NULL) and (flag == FALSE)
    If ITEM == INFO (ptr)   //Node exists
        Then flag ← TRUE
        Write ('ITEM already exists')
        Exit   //Quit execution
    If ITEM < INFO (ptr)   //Go to left sub-tree
        Then ptr1 ← ptr
        ptr ← LPTR (ptr)
    Else
        If ITEM > INFO (ptr)   //Go to right sub-tree
            Then ptr1 ← ptr
            ptr ← RPTR (ptr)

3. [Get availability of node]
If (ptr == NULL)   //Insert when search halts at dead end
Then
    NEW ← AVAIL
    INFO (NEW) ← ITEM   //Avail a node and initialize it
    LPTR (NEW) ← NULL
    RPTR (NEW) ← NULL

4. [Insert node into tree]
If (INFO (ptr1) < ITEM)   //Insert as right child
    Then RPTR (ptr1) ← NEW
Else
    LPTR (ptr1) ← NEW   //Insert as left child

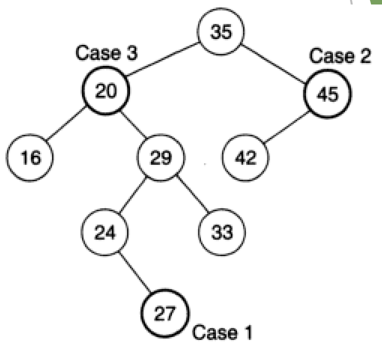
5. [Finished]

Insertion in BST

```
      100
     /   \        Insert 40
   20     500
  /  \          ---------->
10    30
```

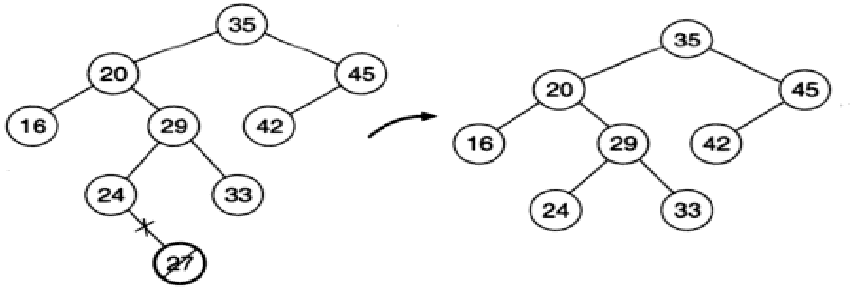
?

Deletion in BST



- If N is leaf node
- N has exactly one child
- N has two children

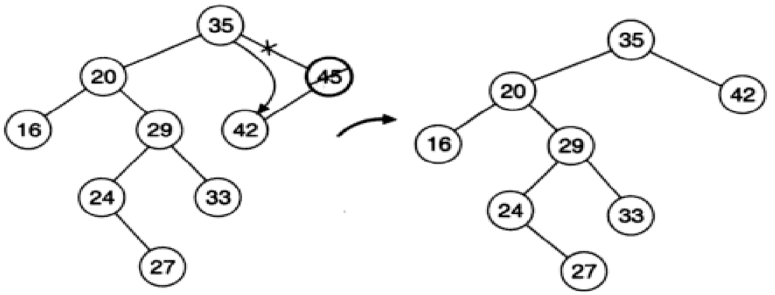
- Case 1: N is deleted from T by simply setting the pointer of N in the parent node PARENT(N) by null value.



1. Deletion of the node 27

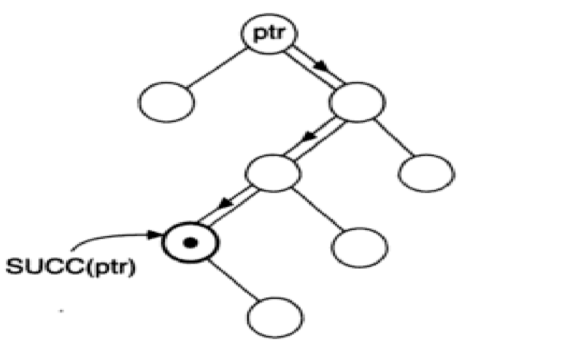
Deletion in BST

- Case 2: N is deleted from T by simply replacing the pointer of N in PARENT(N) by the pointer of the only child of N.

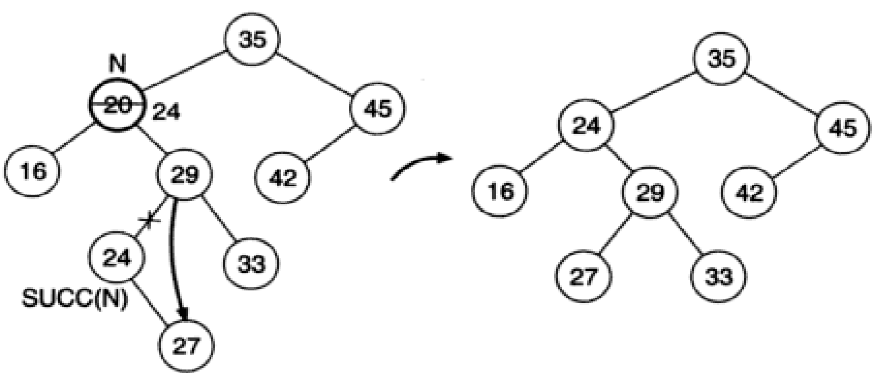


2. Deletion of the node 45

- Case 3: N is deleted from T by first deleting SUCC(N) from T (by using case1 or case2 it can be verified that SUCC(N) never has a child) and then replacing the data content in node N by the data content in node SUCC(N). Reset the left child of the parent of SUCC(N) by the right child of SUCC(N)



Inorder successor of a node ptr.



(c) Deletion of the node 20

Deletion in BST

```
/*DECIDE THE CASE OF DELETION*/


IF(ptr->LCHILD=NULL) and (ptr->RCHILD=NULL) then

Case=1

ELSE

      IF(ptr->LCHILD != NULL) and (ptr->RCHILD != NULL) then

      Case=3

      ELSE

      Case=2

EndIf

EndIf
```

```
/*DELETION CASE 1*/

IF(case=1) then

      If(parent->LCHILD = ptr) then

         Parent->LCHILD = NULL

      ELSE

      Parent->RCHILD = NULL

      EndIf

      Return Node(ptr)

EndIf
```

Deletion in BST

```
/*DELETION Case 2*/
IF(case=2) then
        If(parent->LCHILD = ptr) then
                If(ptr->LCHILD=Null) then
                        Parent->LCHILD = ptr->RCHILD
        ELSE
                Parent->LCHILD = ptr->LCHILD
        EndIf
Else
        If(parent->RCHILD = ptr) then
                If(ptr->LCHILD=NULL) then
                        Parent->RCHILD = ptr->RCHILD
                Else
                        Parent->RCHILD = ptr->LCHILD
                EndIf
        EndIf
EndIf
        ReturnNode(ptr)
EndIf
```

```
/*DELETION Case 3*/
If (case=3)
        ptr2=SUCC(ptr1)
        Item1=ptr2->DATA
Delete_BST(item1)
ptr1->DATA=item1
EndIf
Stop


SUCC(ptr)
PTR = ptr-> RCHILD
If  (PTR != NULL) then
        while (PTR -> LCHILD !=
NULL)
        PTR= PTR-> LCHILD
Return(PTR)
```

Deletion in BST

```
2. [Search for the node marked for deletion]
       FOUND ← false
       Repeat while not FOUND and CUR ≠ NULL
           If DATA(CUR) = X
           then   FOUND ← true
           else   If X < DATA(CUR)
                  then   (branch left)
                         PARENT ← CUR
                         CUR ← LPTR(CUR)
                         D ← 'L'
                  else   (branch right)
                         PARENT ← CUR
                         CUR ← RPTR(CUR)
                         D ← 'R'
       If FOUND = false
       then   Write('NODE NOT FOUND')
              Return
```

```
3. [Perform the indicated deletion and restructure the tree]
       If LPTR(CUR) = NULL
       then   (empty left subtree)
              Q ← RPTR(CUR)
       else   If RPTR(CUR) = NULL
              then   (empty right subtree)
                     Q ← LPTR(CUR)
              else   (check right child for successor)
                     SUC ← RPTR(CUR)
                     If LPTR(SUC) = NULL
                     then   LPTR(SUC) ← LPTR(CUR)
                            Q ← SUC
                     else   (search for successor of CUR)
                            PRED ← RPTR(CUR)
                            SUC ← LPTR(PRED)
                            Repeat while LPTR(SUC) ≠ NULL
                                PRED ← SUC
                                SUC ← LPTR(PRED)
                            (connect successor)
                            LPTR(PRED) ← RPTR(SUC)
                            LPTR(SUC) ← LPTR(CUR)
                            RPTR(SUC) ← RPTR(CUR)
                            Q ← SUC
       (Connect parent of X to its replacement)
       If D = 'L'
              then   LPTR(PARENT) ← Q
              else   RPTR(PARENT) ← Q
              Return
```

Delete in BST

Node to be deleted is the leaf

```
        50
     /     \         delete(20)
    30      70       --------->
   /  \    /  \
 20   40  60   80
```

?

Delete in BST

Node to be deleted has only one child

```
      50
   /     \          delete(30)
  30      70        --------->
    \    /  \
    40  60   80
```

?

Delete in BST

Node to be deleted has two children

```
      50
   /      \          delete(50)
  40       70        ---------->
          /  \
         60   80
```

?

Thankyou!!!