Binary Search Trees

- **Definition** of binary search tree:
  - Every element has a unique key
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree
  - The left and right subtrees are also binary search trees

Binary Search Trees

- Example: (b) and (c) are binary search trees

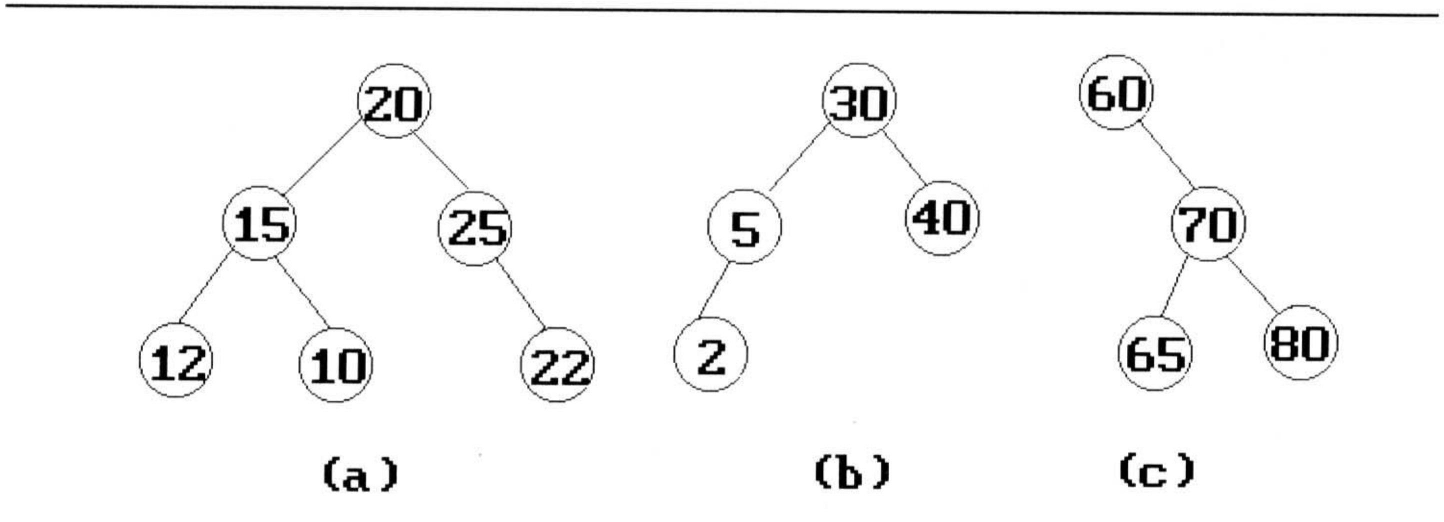
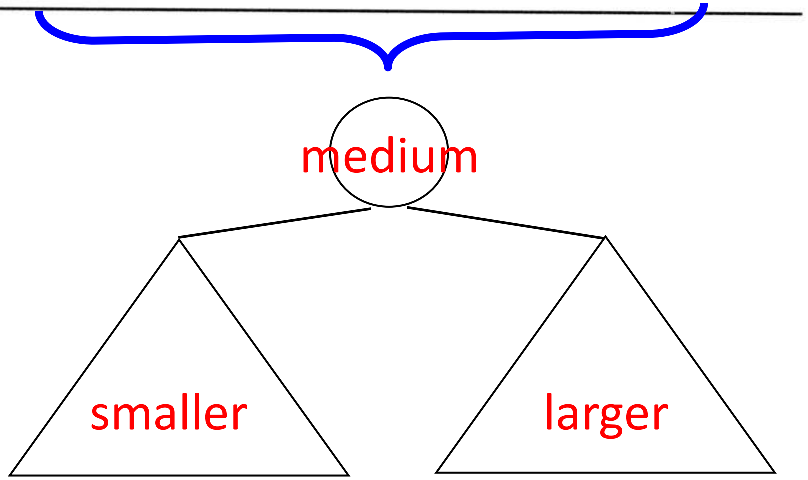
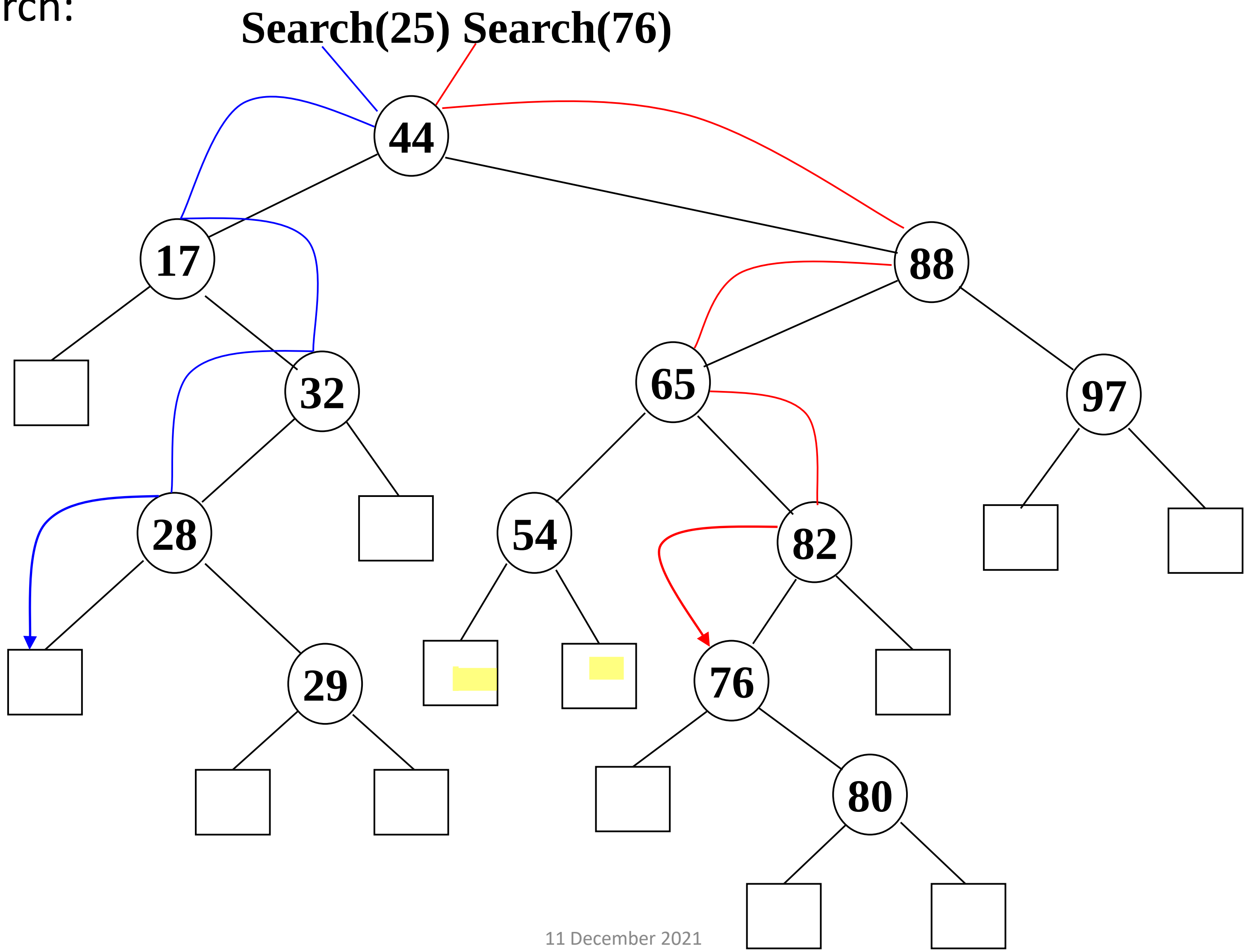


Figure 5.30: Binary trees



Binary Search Trees

- Search:



binary search tree (BST)

```
struct bst
{
    struct bst *lchild;
    int data;
    struct bst *rchild;
};
struct bst *root=NULL;
```

Insert into binary search tree

```
void insert(int ele)
{       struct bst *temp=(struct bst *)(malloc(sizeof(struct bst)));
        temp->data=ele;   temp->left=NULL; temp->right=NULL;
        if(root==NULL){root=temp;return;}
        else
        {       struct bst *curr=root,*prev=NULL;
                while(curr)
                {
                        prev=curr;
                        if(temp->data<curr->data)      curr=curr->lchild;
                        else if(temp->data>curr->data)      curr=curr->rchild;
                        else
                        {       printf("Insertion is not possible. %d already present in BST“,ele);
                                return;
                        }
                }
                if(temp->data>prev->data)                         prev->rchild=temp;
                else if(temp->data<prev->data)                          prev->lchild=temp;
        }
        return;
}
```

binary search tree: Display

```
void display(struct bst *ptr)
{
    if(ptr){
        display(ptr->lchild);
        printf(“%d ”,ptr->data);
        display(ptr->rchild);
    }
}
```

```
void searchI(int ele)
{
    if(root==NULL)
    {   printf(“BST is empty“);   return; }

    struct bst *curr=root;
    while(curr!=NULL)  //while(curr)
    {
        if(curr->data==ele)
        {   printf(“Element found“); return; }

        else if(curr->data>ele)      curr=curr->lchild;
        else                         curr=curr->rchild;    //curr->data<ele
    }

    printf(“%d not found in BST“,ele);
}
```

```
struct bst* searchr(struct bst *r,int key)
{
        // return the node address of key element if key is found
        // if key not found return NULL
    if(!r) //if(r==NULL)
            return NULL;
    if(key==r->data)      return r;

    if(key<r->data)       return(searchr(r->lchild, key));
    return(searchr(r->rchild, key));

}
```

Deletion from a binary search tree

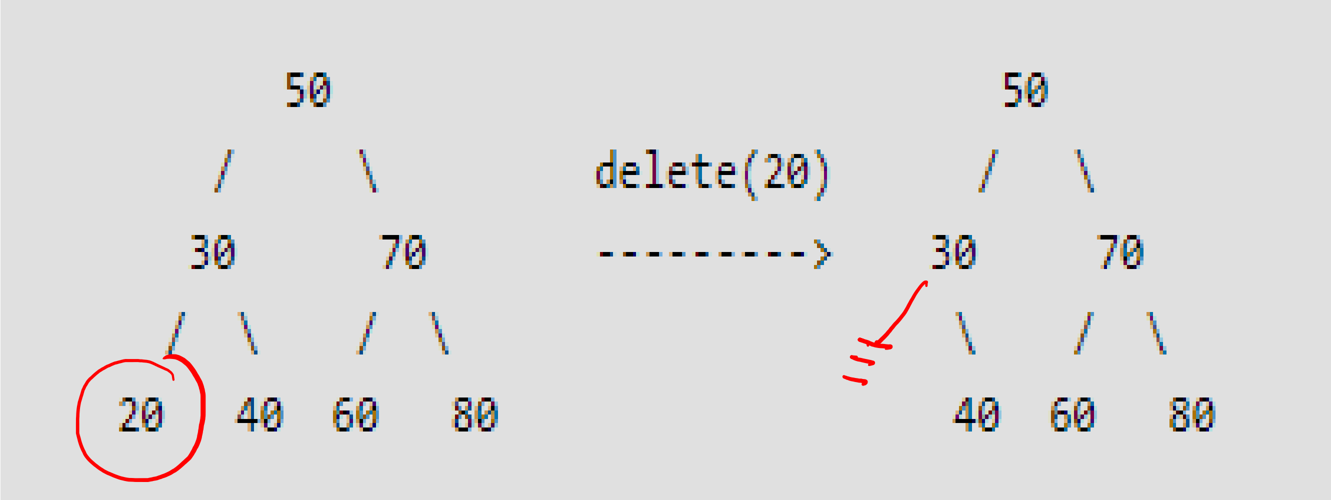
Three cases should be considered

- Case 1. leaf → delete

- Case 2. two children → replace the deleted element with either the smallest element in the right subtree or the largest element in the left subtree

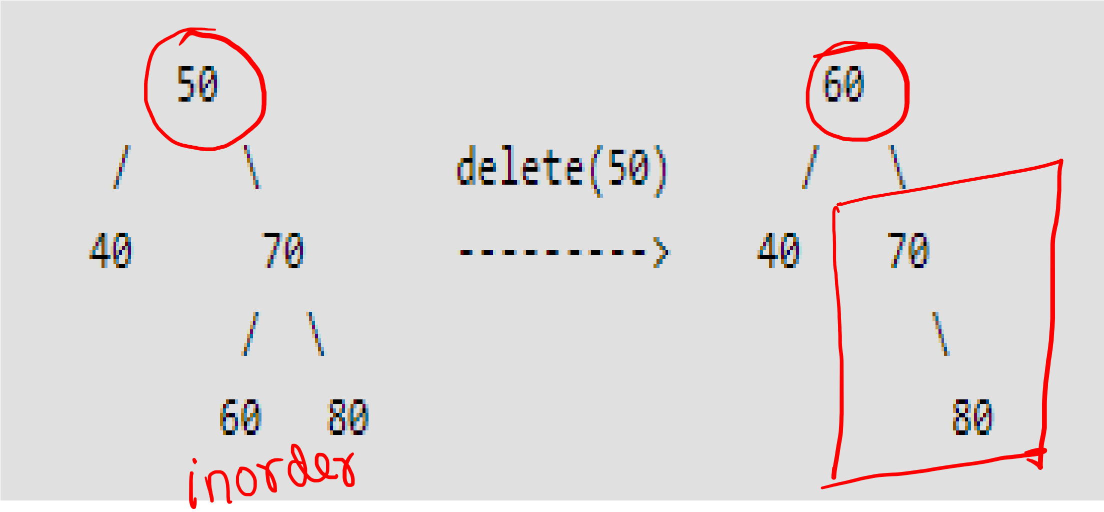
- Case 3. one child → delete and change the pointer to this child

1) Node to be deleted is leaf: Simply remove from the tree.



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

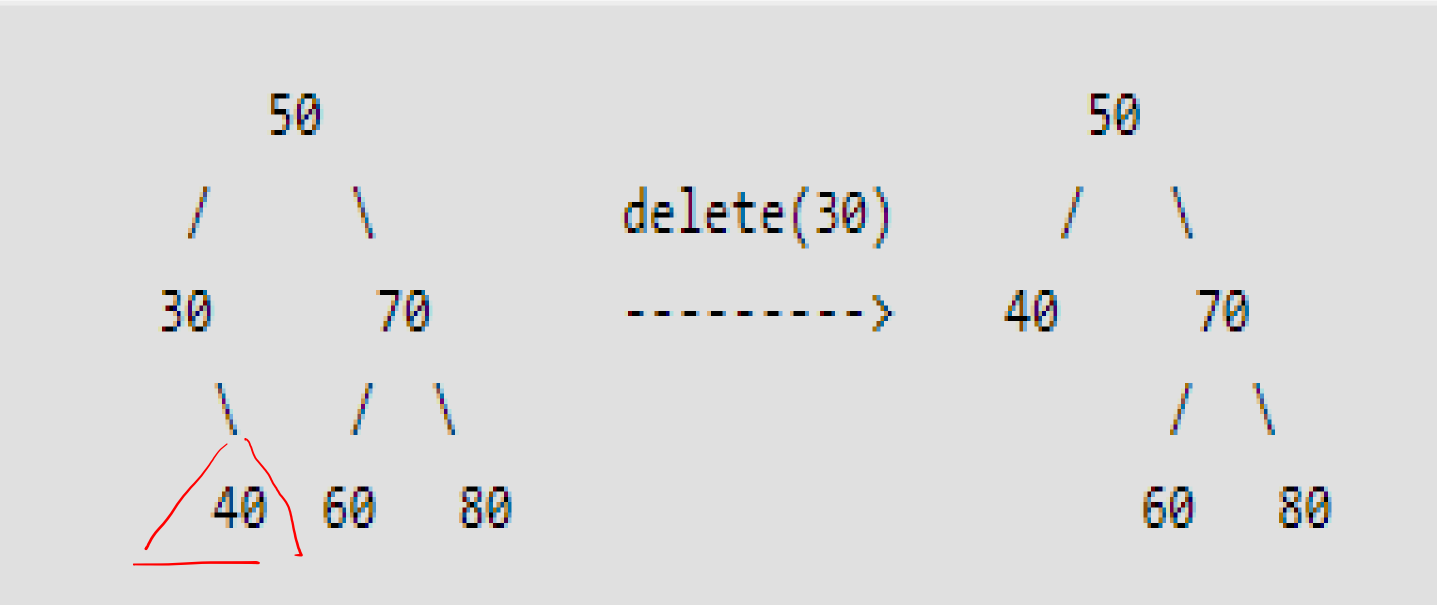
2)Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



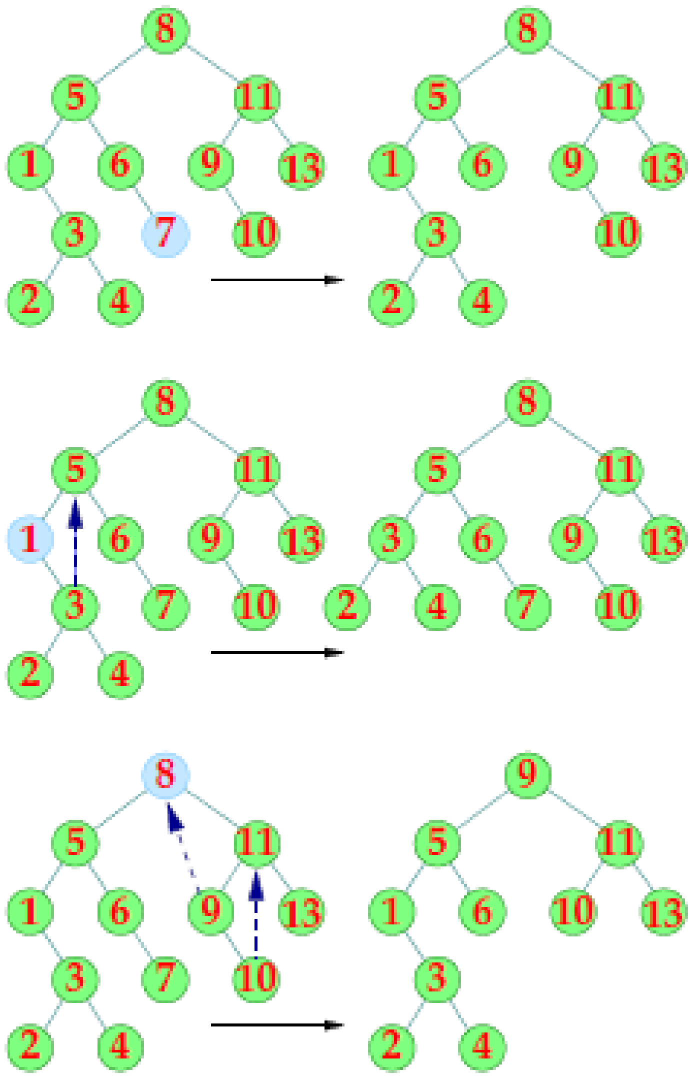
```
          50                              60
        /     \         delete(50)      /    \
      40       70       --------->    40      70
              /  \                               \
            60    80                              80
```

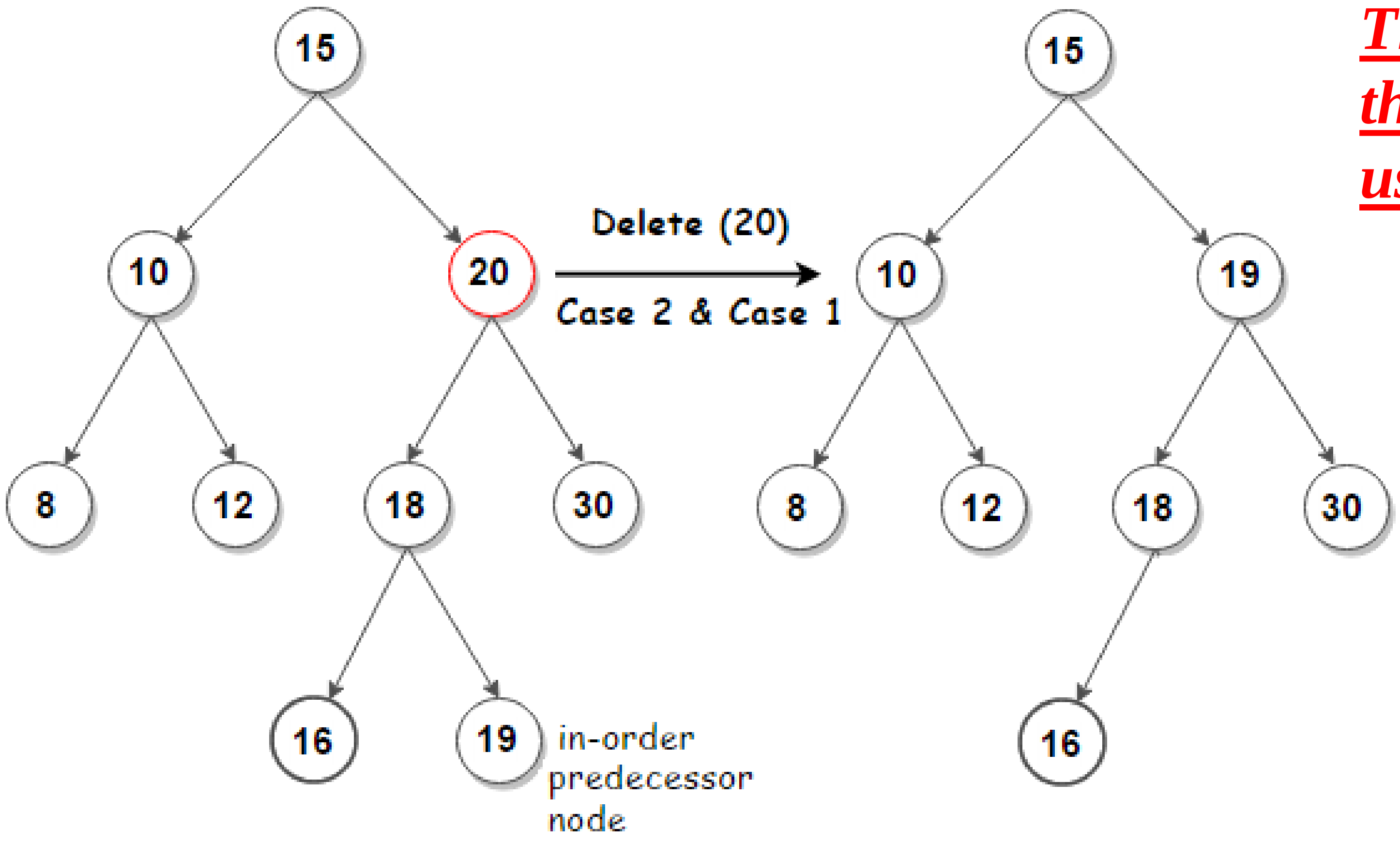
inorder
succ

3) *Node to be deleted has only one child:* Copy the child to the node and delete the child



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


8
5 11
1 6 9 13
3 7 10
2 4
8
5 11
1 6 9 13
3 10
2 4
8
5 11
1 6 9 13
3 7 10
2 4
8
5 11
3 6 9 13
2 4 7 10
8
5 11
1 6 9 13
3 7 10
2 4
9
5 11
1 6 10 13
3
2 4



This slide shows an example for "Note that inorder predecessor can also be used."

Delete(20) function call will do the following:

a. **Case 2 will run first. According to case 2 the node to be deleted will get replaced by it's inorder predecessor which is 19 (right-most node on the left subtree of 20)**
b. **Case 1 (As 19 is leaf node) will run next to delete node 19**
c. **According to case 1, 19's parent 18->rchild must be made NULL.**

```
// Helper function to find minimum value node in subtree rooted at curr
struct bst* minimumKey(struct bst* curr)
{
    while(curr->lchild != NULL)        curr = curr->lchild;

    return curr;
}
```

```
// Iterative function to search in subtree rooted at curr & set its parent
// Note that curr & parent are passed by reference
void searchKey(struct bst* curr, int key, struct bst* parent)
{
    // traverse the tree and search for the key
    while (curr != NULL&& curr->data != key)
    {
        // update parent node as current node
        parent = curr;

        // if given key is less than the current node, go to lchild subtree
        // else go to rchild subtree
        if (key < curr->data)   curr = curr->lchild;
        else                    curr = curr->rchild;
    }
}
```

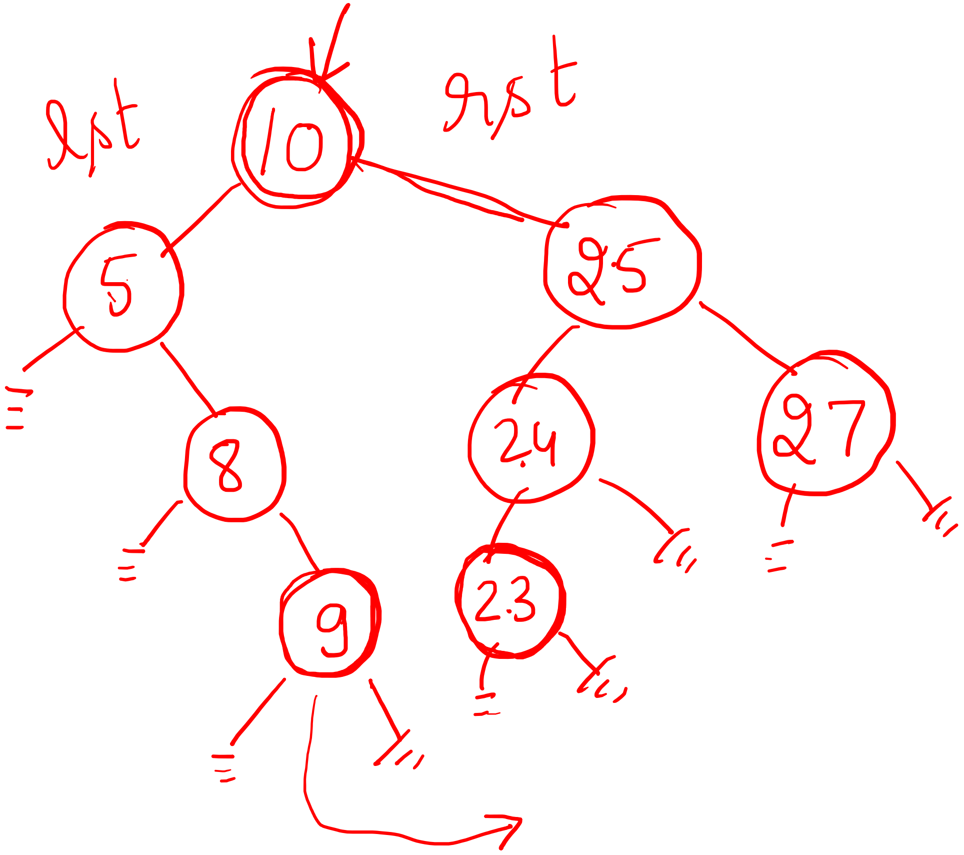
```
struct bst* deleteIterative(struct bst* root, int key)
{
      struct bst* curr = root;
      struct bst* prev = NULL;

      while (curr != NULL && curr->data != key) {
            prev = curr;
            if (key < curr->data)               curr = curr->left;
            else                                curr = curr->right;
      }

      if (curr == NULL) {
          printf("%d not found in BST",key);
            return root;
      }
```

```
// Check if the node to be deleted has atmost one child.
    if (curr->left == NULL || curr->right == NULL) {
        // newCurr will replace the node to be deleted.
        struct bst* newCurr;

        if (curr->left == NULL)              newCurr = curr->right; // if the left child does not exist.
        else                              newCurr = curr->left; // if the right child does not exist.

        // check if the node to be deleted is the root.
        if (prev == NULL)                 return newCurr;

        if (curr == prev->left)              prev->left = newCurr;
        else                                 prev->right = newCurr;

        free(curr);
    } //if
```

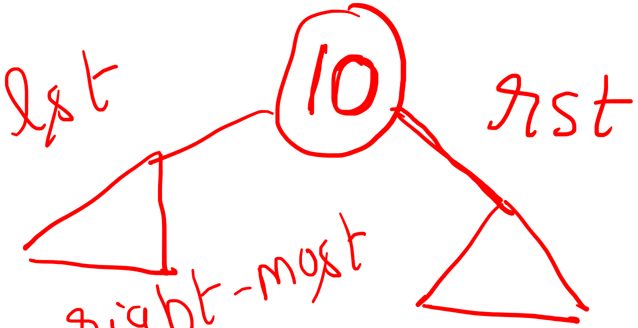
```
// node to be deleted has two children.
    else {
        struct bst* p = NULL;
        struct bst* temp;
        // Compute the inorder successor
        temp = curr->right;
        while (temp->left != NULL) {
            p = temp;
            temp = temp->left;
        }
        if (p != NULL)              p->left = temp->right;
        else                        curr->right = temp->right;

        curr->data = temp->data;
        free(temp);
    } //else
    return root;
}//deleteIterative
```

```
//delete Recursive
struct bst *deleteNode(struct bst *root, int value){
   struct bst* iPre;
   if (root == NULL)    return NULL;

   if (root->left==NULL&&root->right==NULL){
      free(root);       return NULL;
   }

   //searching for the node to be deleted
   if (value<root->data)       root->left = deleteNode(root->left,value);

   else if (value > root->data) root->right = deleteNode(root->right,value);

   //deletion strategy when the node is found
   else{
      iPre = inOrderPredecessor(root);
      root->data = iPre->data;
      root->left = deleteNode(root->left, iPre->data);
   }
   return root;
}
```

```
struct bst *inOrderPredecessor(struct bst* root){
   root = root->left;
   while (root->right!=NULL)
         root = root->right;

   return root;
}
```

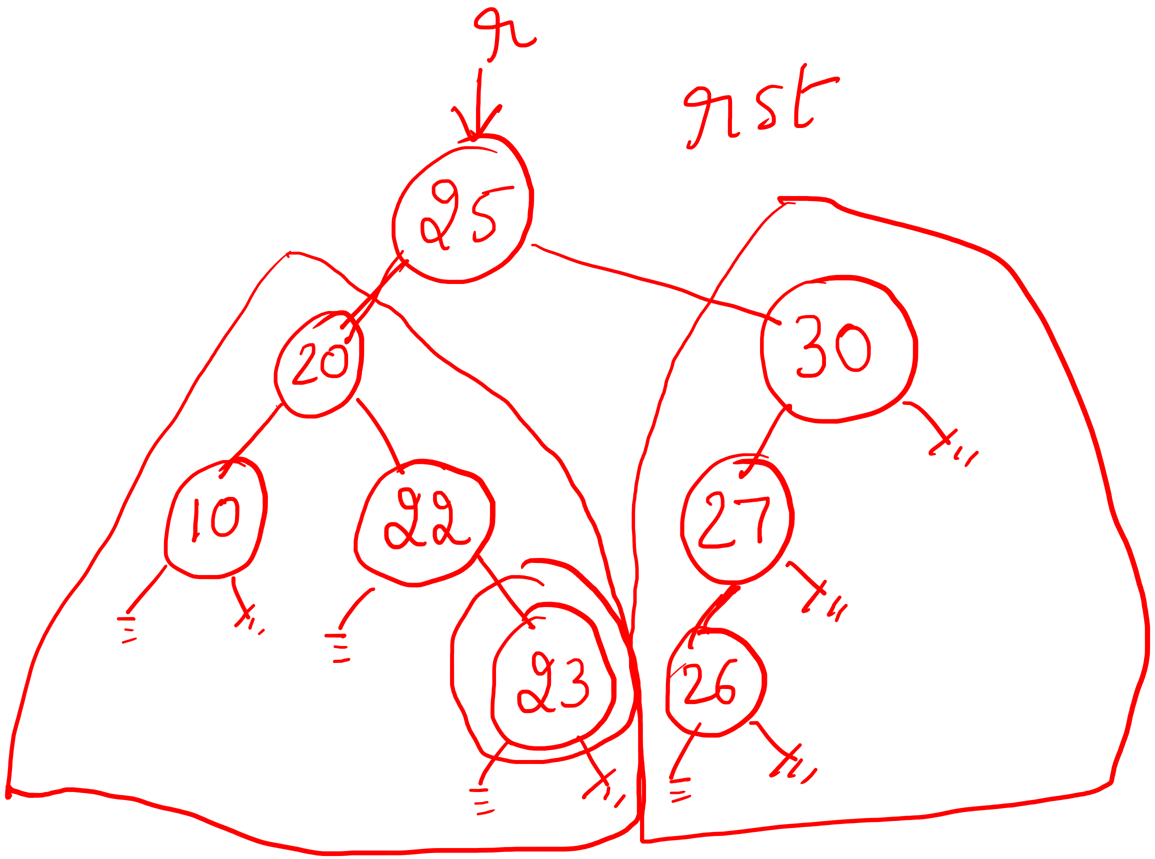


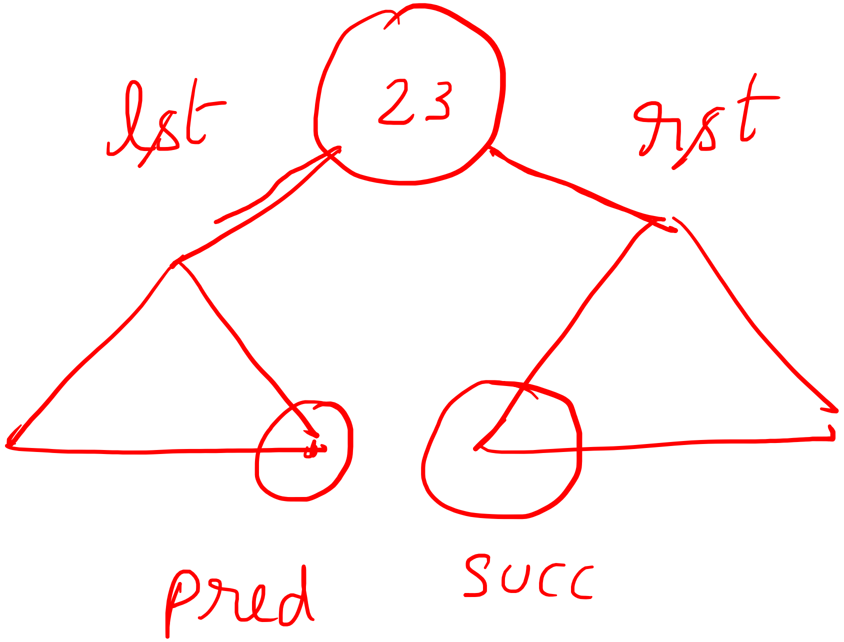
lst: left subtree
rst: right subtree



right-most node (inorder predecessor)

left most node (inorder successor)





inorder predessor : 23

inorder successor : 26

10 20 22 23 25 26 27 30