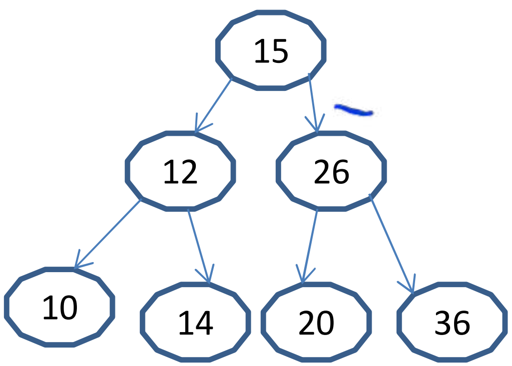
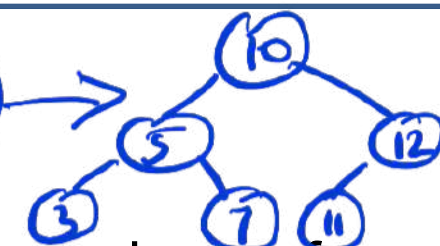
Binary Search Tree

- A binary search tree is a binary tree such that
  - ~~Data at every node is greater than that at its left child and less than that at its right child.~~
  - Data at every node is greater than that at every node in its left subtree
  - Data at every node is less than that at every node in its right subtree.



What happens if 14 is changed as 24?
Or 20 is changed as 10?

$\lfloor \log_2(n) \rfloor$



- Search, insert and delete operations can be performed in a binary search tree.
- In the above tree insert 18, search for 20, delete 14, delete 26

Binary Search Tree

- Insert 18:
  - Compare 18 with root. It is more. Move to right subtree
  - Compare 18 with 26. It is less. Move to left subtree.
  - Compare 18 with 20. It is less. Move to left subtree.
  - The left link in the node with data 20 is NULL. It is null tree.
  - Create a new node p, with data as 18, left and right links as null
  - Make p as left link of node 20.
- Search 20
  - Same as above.
  - Either the element is found or reach a null tree. In such case element is not present in the tree.

Binary Search Tree

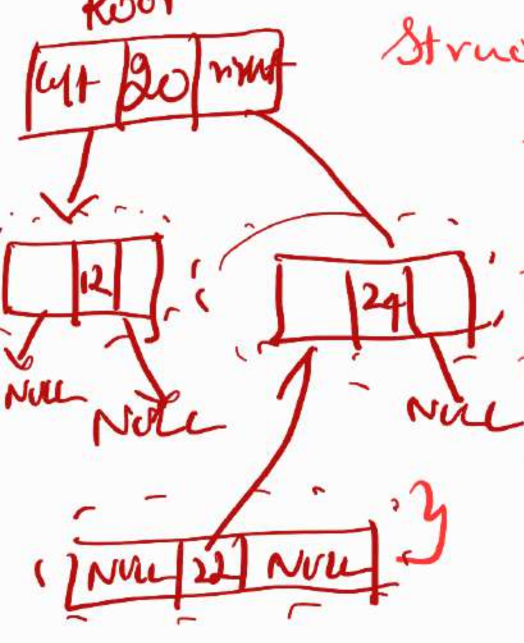
- Delete 14, Delete 26
  - Search for 14 / Search for 26.
  - It can be
    - » a leaf node.
    - » Node with only left child
    - » Node with only right node
    - » Node with both children
  - In the first case, the node can be deleted.
  - In the second and third case, the node can be replaced with the left child or right child respectively
  - In the fourth case, the data at the node can be swapped with the data at the left most node (say node x) in the right subtree and node x can be deleted.

Binary Search Tree.

ADT :-

1. Insertion.
2. Deletion.
3. Search.
4. Print / Traverse.

```
struct node
{
    int data;

    Struct node *left;   // store the address of left node.
    Struct node *right;  // Stores the address of right node
}

struct node * Create (int Value)
{                         // function to create node;
    Struct node *temp = malloc;
    temp -> data = value;
    temp -> left = NULL;
    temp -> right = NULL;
    Return temp;   // -> Address of newly created node
}
```



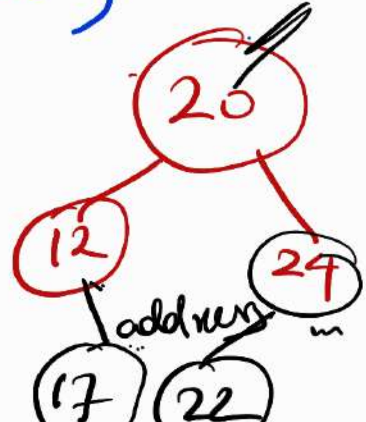
```
Struct node *Insert(*node, int element)
{
    if (node == NULL)
    {
        return Create(element);
    }

    else if (element > node->data)
    {
        node->right = Insert(node->right,
                             element);
    }
    else (element < node->data)
    {
        node->left = Insert(node->left,
                            element);
    }
    return node;
}
Void main()
{
    Struct node *root = NULL;
    root = Insert(root, 20)
    Insert(root, 12)
    Insert(root, 24)
}
```

node->left (Insert) ← address (//Created)

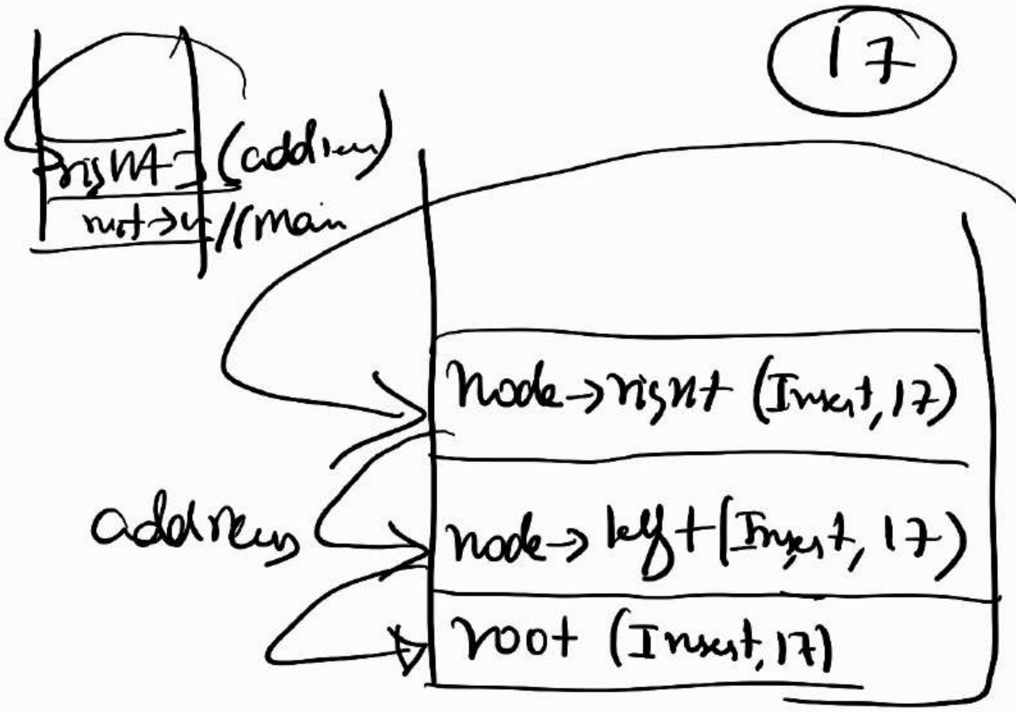
node->right (Insert) node->left = address

root -> Insert

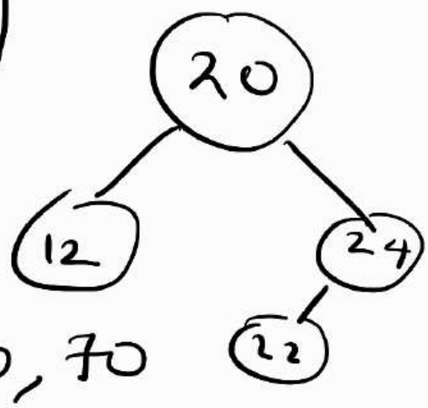
Main ()



17



// address of newly created node



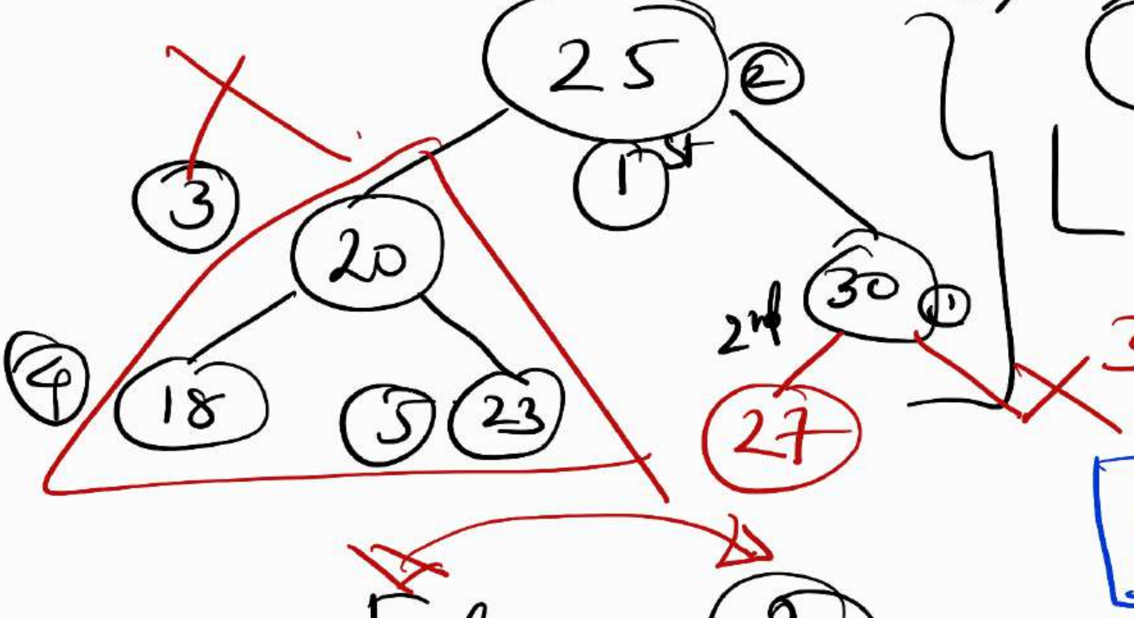
O(N)

20, 30, 40, 50, 60, 70

25, 20, 30, 18, 23

27

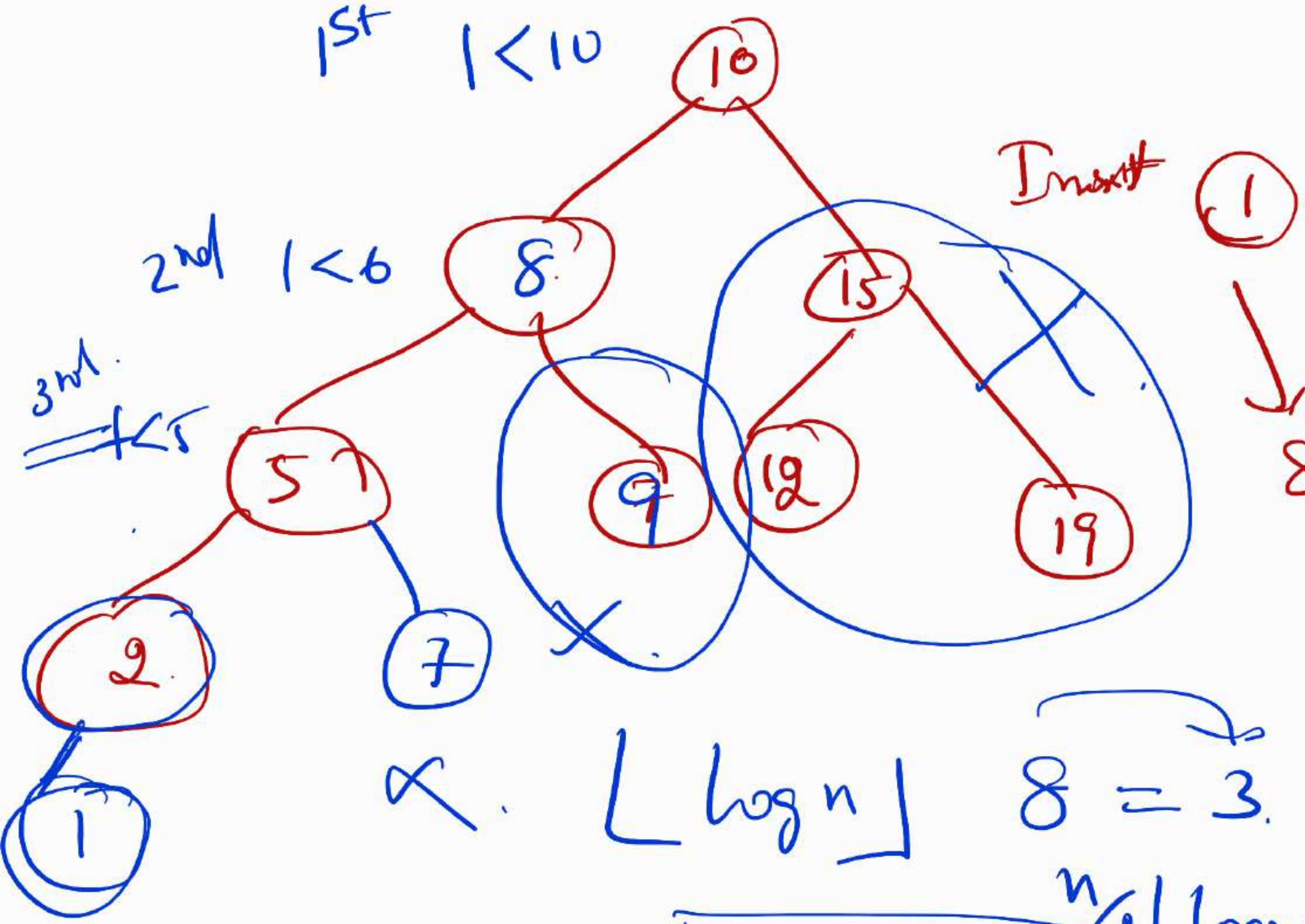
$\lfloor \log n \rfloor$



$\boxed{O(N)}$

5 element = 2 steps.

$\boxed{\text{Insertion} = O(\log n).}$

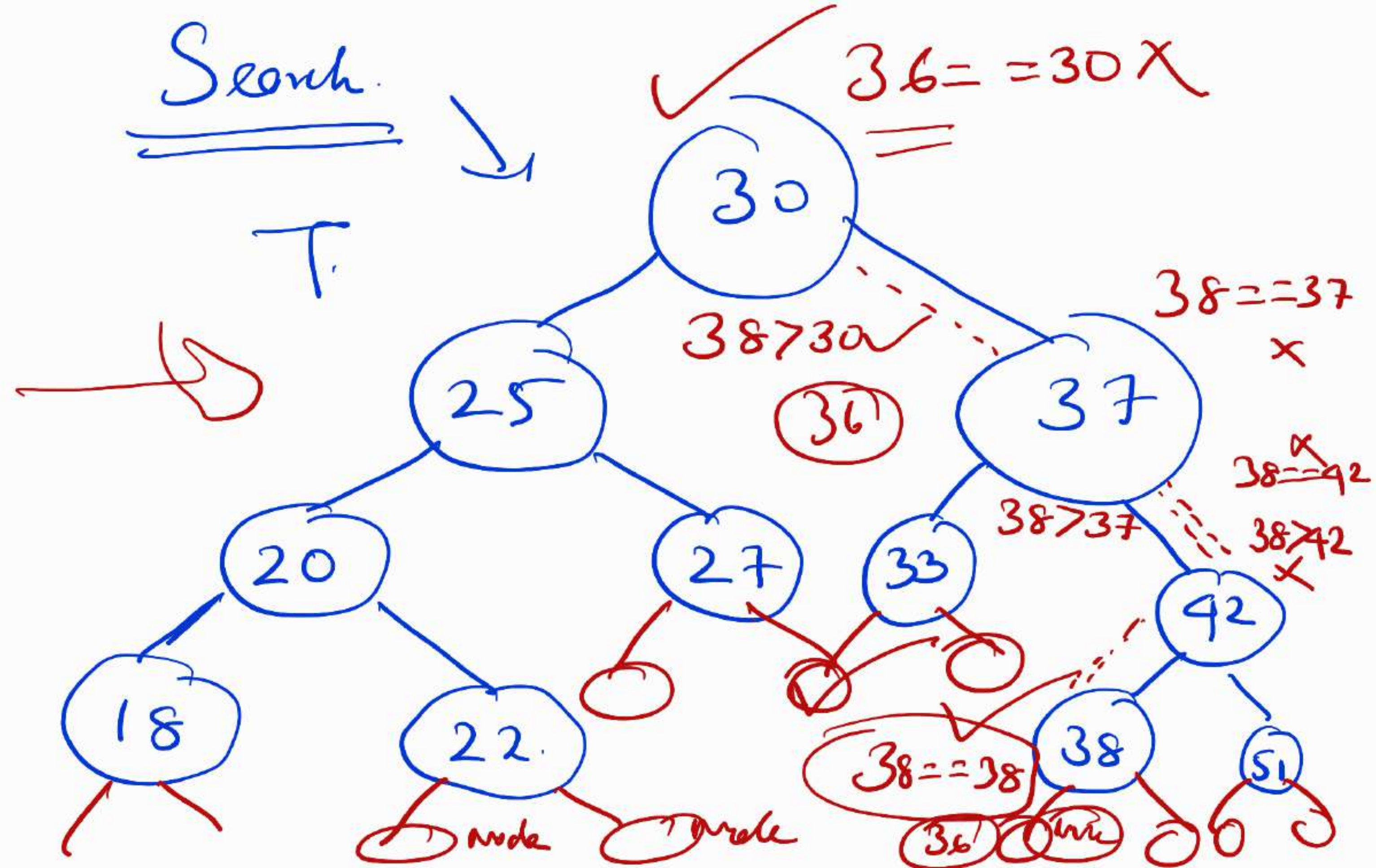


$\lfloor \log n \rfloor$ 8 = 3.

$n/2 \lfloor \log n \rfloor$

$\boxed{100 = 16}$

Search



```
Struct node * search(Struct node *node,
                                  int element)
{   if (element == node -> data)
    {
      pf ("Element is found");
    }   return node;

    if (element > node -> data)
    {
       return search (node -> right, element);
    }
    if ( element < node -> data)
    {
       return search (node -> left, element);
    }
    if( node == NULL)
    {
      printf ("Element is not found");
    }   return node;
}
```

Binary Trees – Issues in Construction

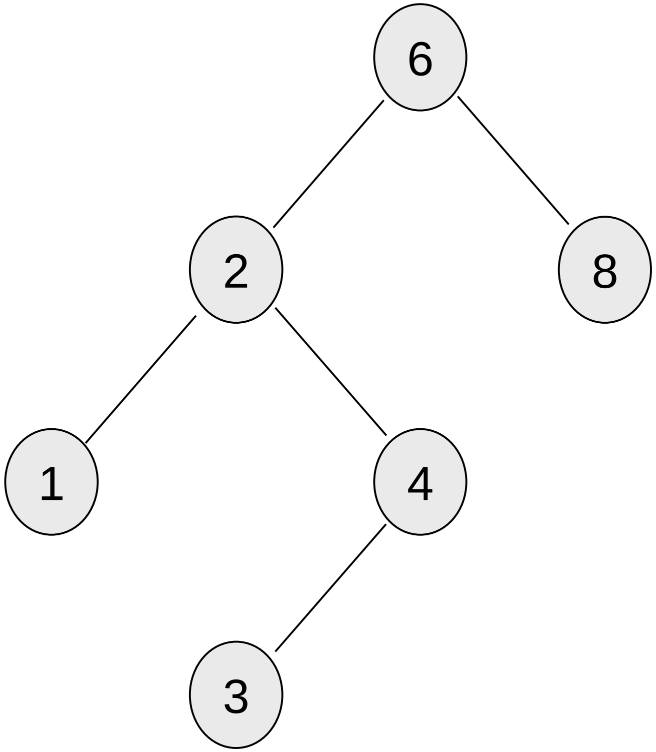
- How can we insert a node in to a binary tree?
  - Need to specify the location as a left or right child of an existing node in the tree
  - What needs to be done if a node is already present at that location?
  - The tree constructed can be of height n -1 (n is number of nodes in the tree
  - The operations of insertion, search and deletion can be of complexity O(n).
- ***Binary search tree*** is an alternative to make the searching convenient and also with average time complexity of O(log n).

Binary Search Trees

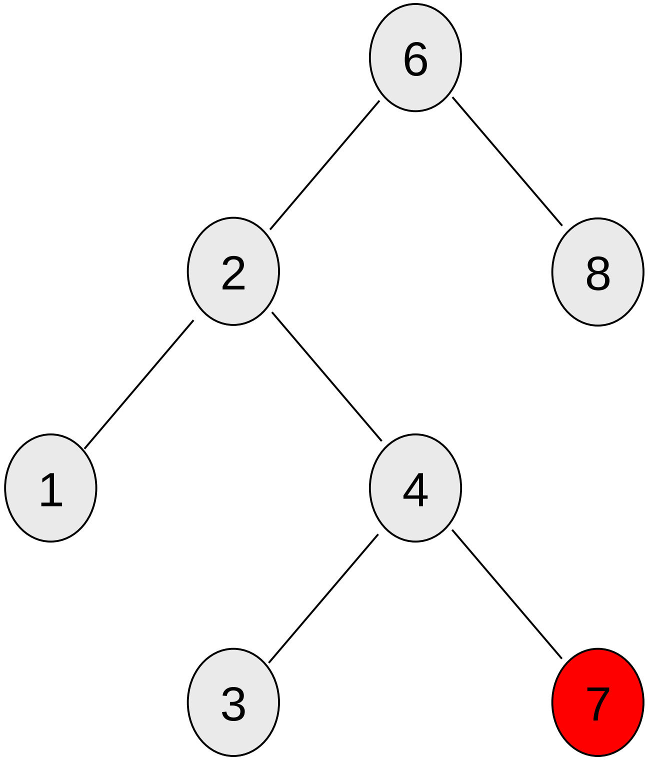
- An important application of binary trees is their use in searching.
- ***Binary search tree*** is a binary tree in which every node X contains a data value that satisfies the following:

  a) all data values in its left subtree are smaller than the data value in X
  b) the data value in X is smaller than all the values in its right subtree.
  c) the left and right subtrees are also binary search tees.

Example

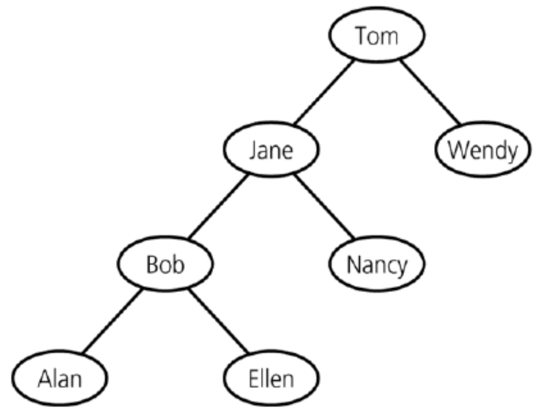


A *binary search tree*

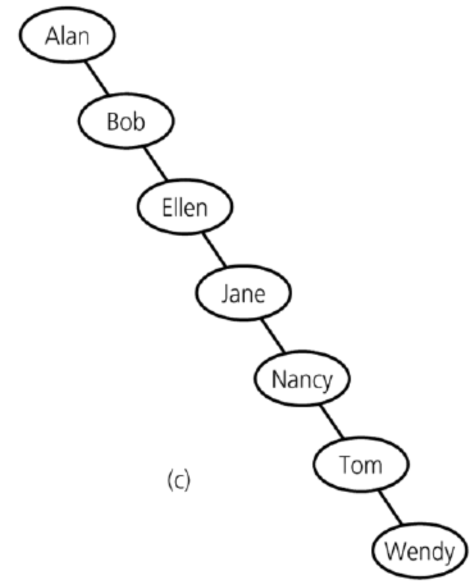


Not a *binary search tree,* but a *binary tree*

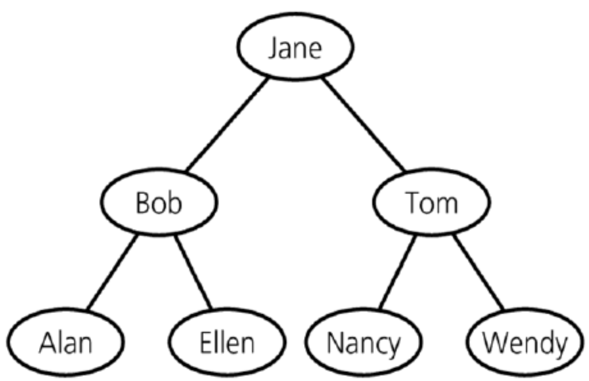
Binary Search Trees – containing same data

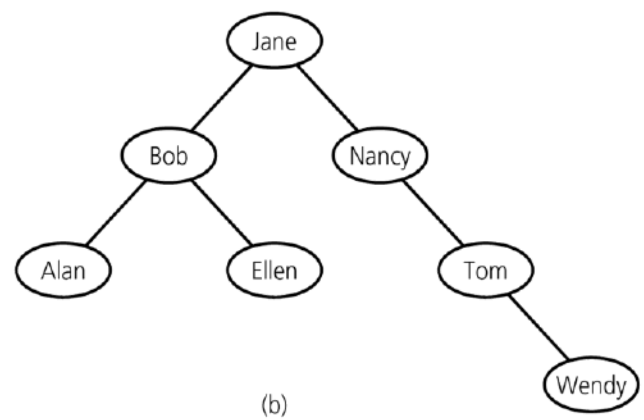


(a)



(c)





(b)

Operations on BSTs

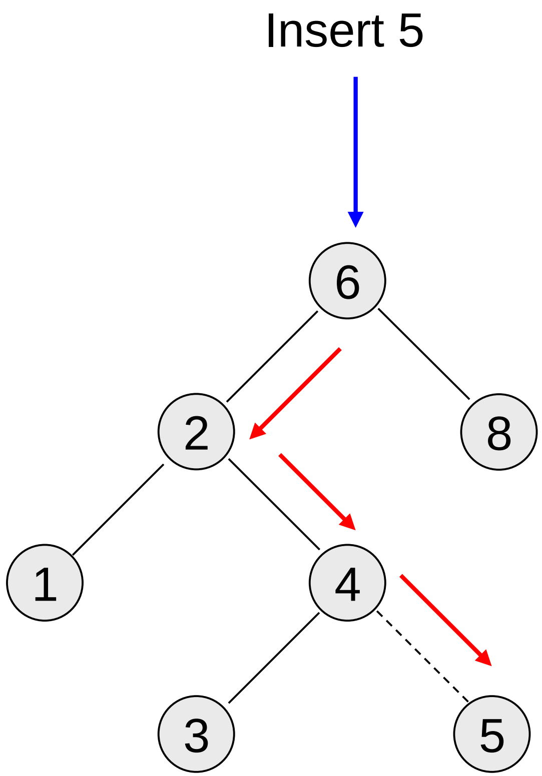
- Most of the operations on binary trees are O(log$N$).
  - This is the main motivation for using binary trees rather than using ordinary lists to store items.
- Most of the operations can be implemented using recursion.
  - we generally do not need to worry about running out of stack space, since the average depth of binary search trees is O(log$N$).

Insert operation

Algorithm for inserting X into tree T:

– Proceed down the tree as you would with a find operation.
– if X is found
    do nothing, (or "update" something)
  else
    insert X at the last spot on the path traversed.

Example



Deletion operation

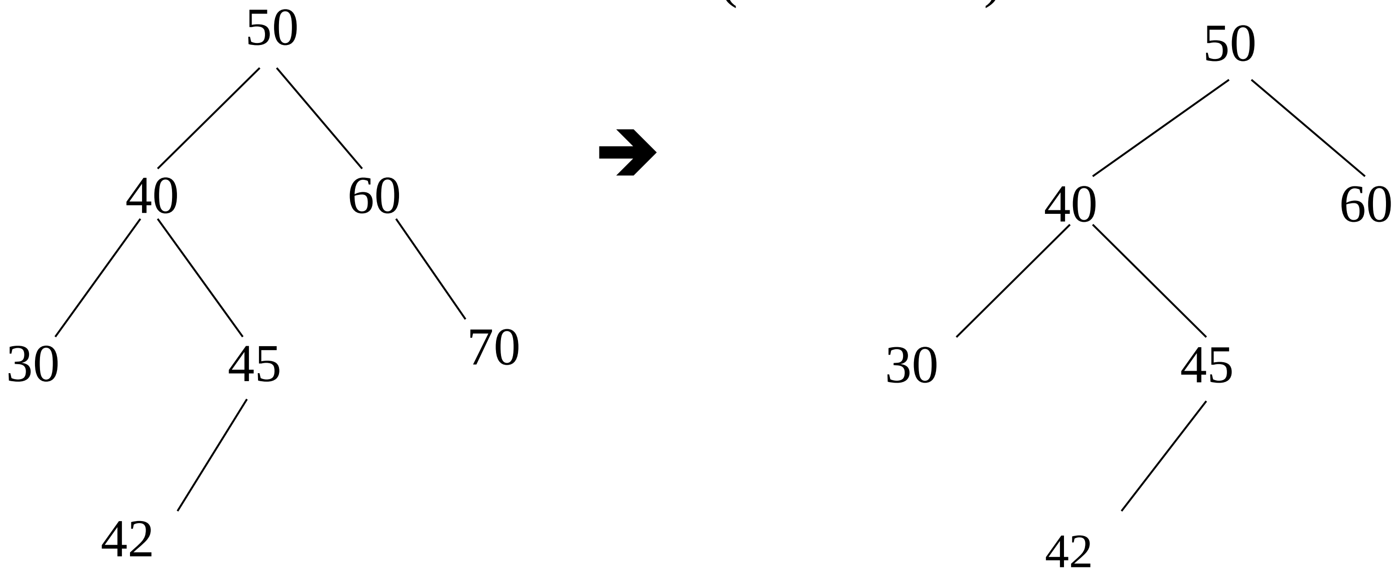
There are three cases to consider:

1. Deleting a leaf node
   - Replace the link to the deleted node by NULL.
2. Deleting a node with one child:
   - The node can be deleted after its parent adjusts a link to bypass the node.
3. Deleting a node with two children:
   - The deleted value must be replaced by an existing value that is either one of the following:
     - The largest value in the deleted node's left subtree
     - The smallest value in the deleted node's right subtree.

Deletion – Case1: A Leaf Node

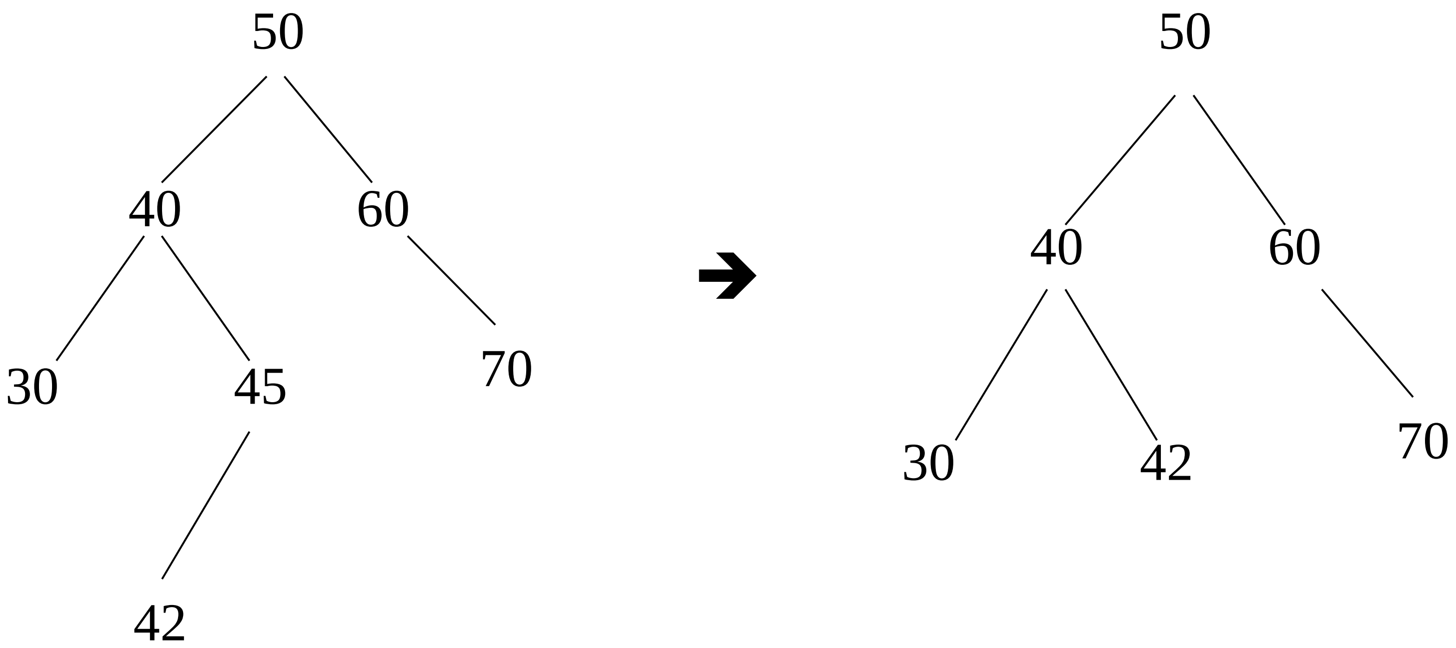
To remove the leaf containing the item, we have to set the pointer in its parent to NULL.

Delete 70 (A leaf node)



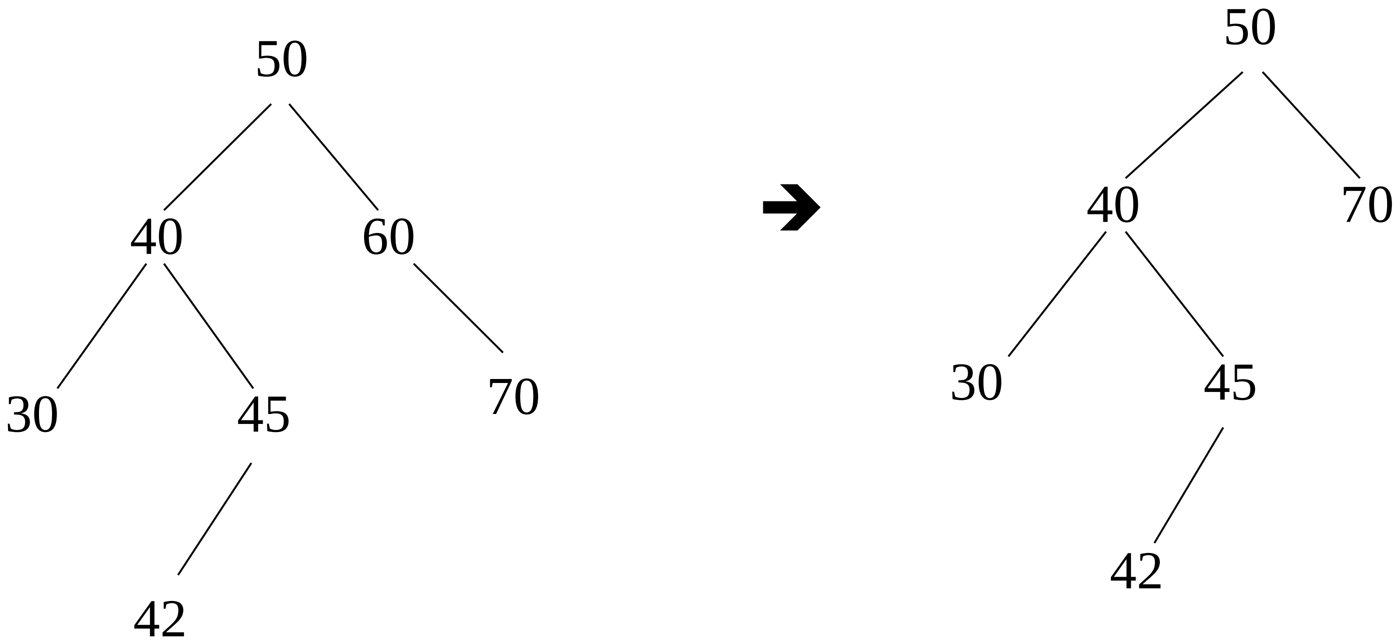
Deletion – Case2: A Node with only a left child

Delete 45 (A node with only a left child)



Deletion – Case2: A Node with only a right child

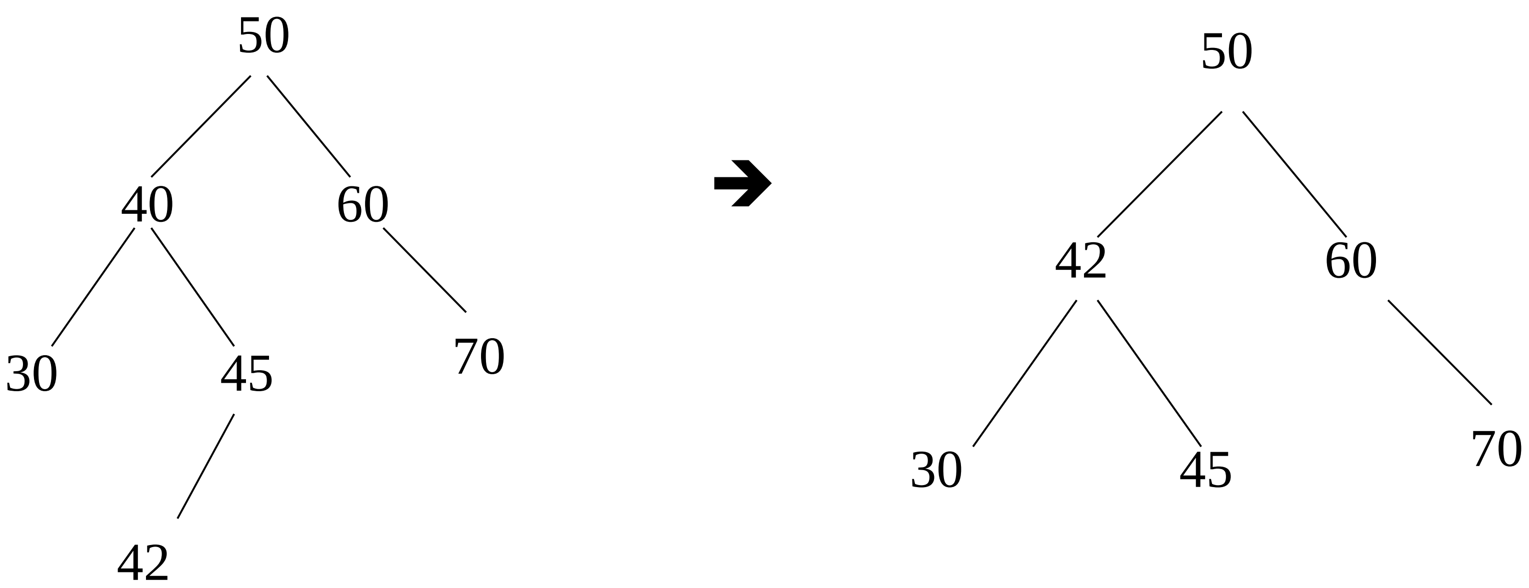
Delete 60 (A node with only a right child)



Deletion – Case3: A Node with two children

- Locate the inorder successor of the node.
- Copy the item in this node into the node which contains the item which will be deleted.
- Delete the node of the inorder successor.

Delete 40 (A node with two children)



Analysis of BST Operations

- The cost of an operation is proportional to the depth of the last accessed node.
- The cost is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate tree.
- In the best case we have logarithmic access cost, and in the worst case we have linear access cost.