Data Structures – CST 201
Module - 4

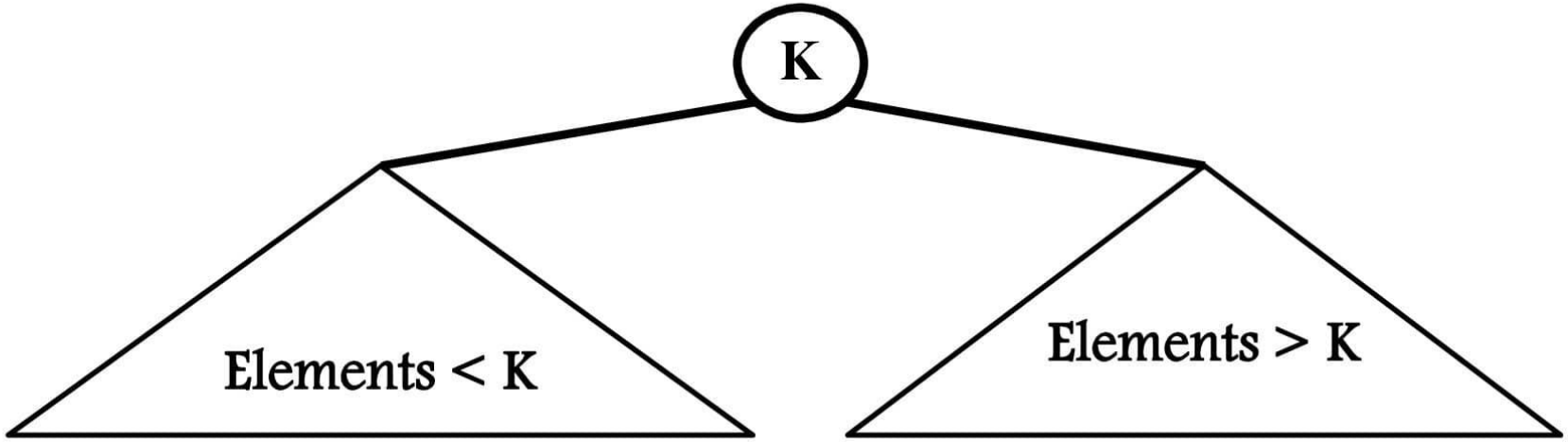
Syllabus

- **Trees and Graphs**
  - Trees
    - Binary Trees
      - Binary Tree Representation
      - Binary Tree Operations
      - Binary Tree Traversals
    - **Binary Search Trees**
      - **Binary Search Tree Operations**
  - Graphs
    - Representation of Graphs
    - Depth First Search and Breadth First Search on Graphs
    - Applications of Graphs

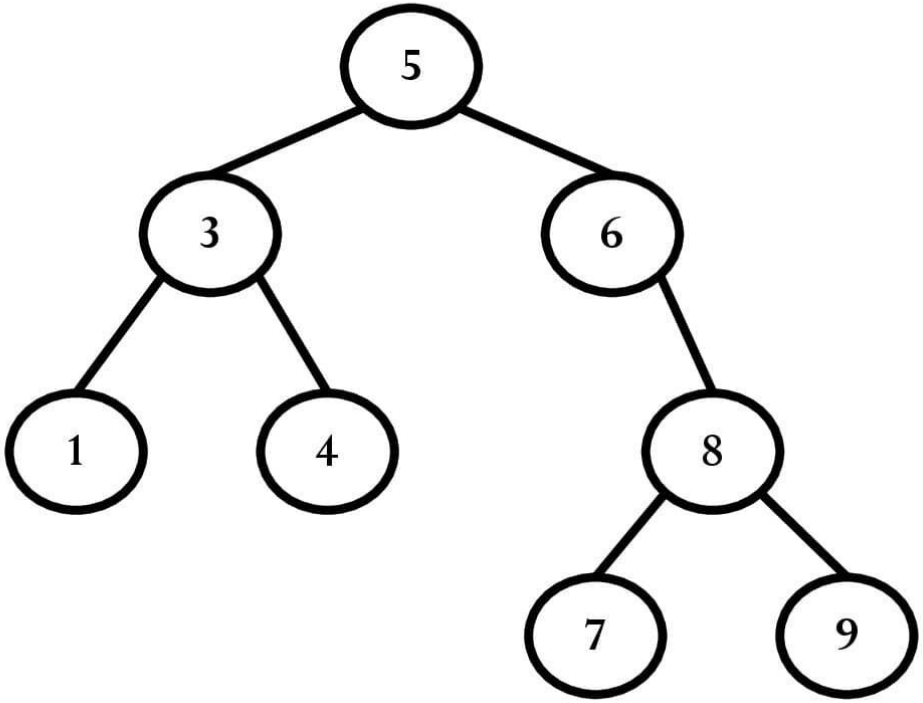
Binary Search Trees

- Binary search tree(BST) is a binary tree that is empty or each node satisfies the following properties:
  - Every element has a key, and no two elements have the same key
  - The keys in a nonempty left subtree must be smaller than the key in the root of the subtree. The keys in a nonempty right subtree must be larger than the key in the root of the subtree
  - The left and right subtrees are also BST

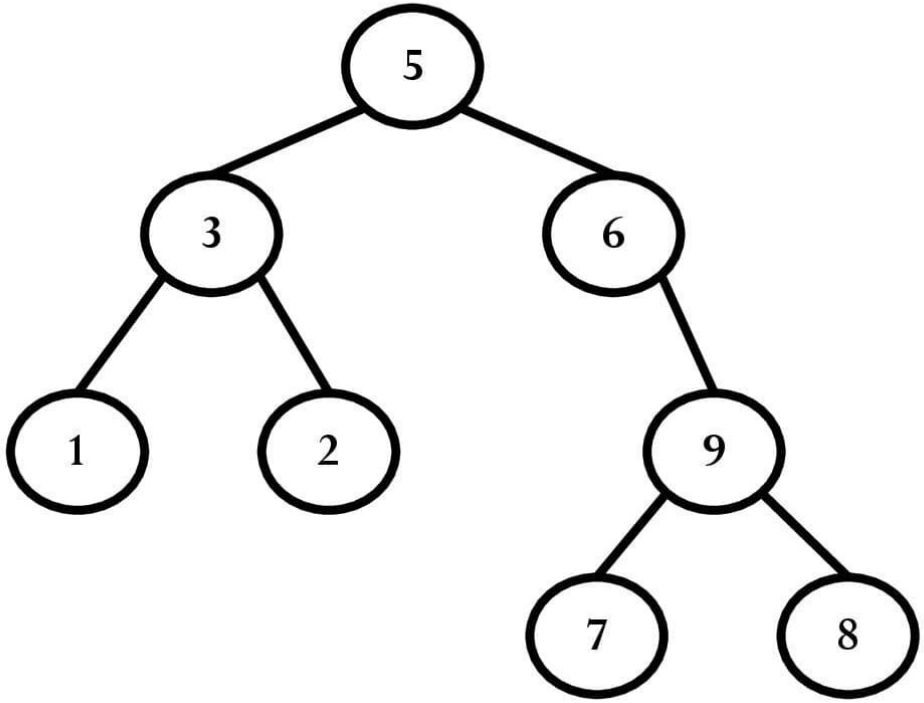
Binary Search Trees



Binary Search Trees

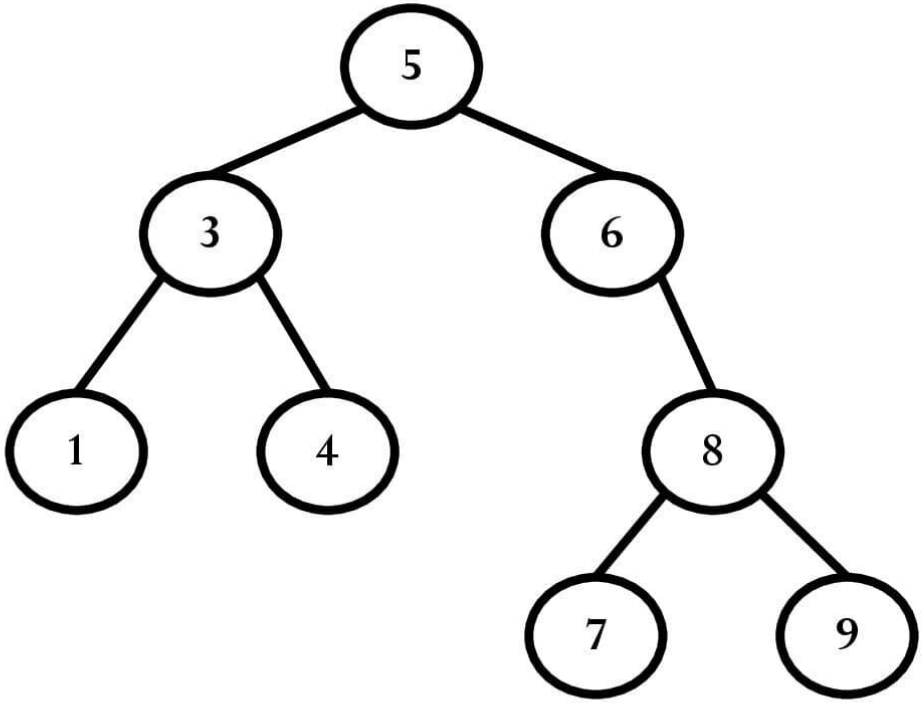


Binary Search Tree



Not a Binary Search Tree

Binary Search Trees



Inorder Traversal: 1 3 4 5 6 7 8 9

Inorder traversal of BST generate a sorted list

Operations on Binary Search Trees

- Four operations
  - Searching
  - Insertion
  - Deletion
  - Traversal

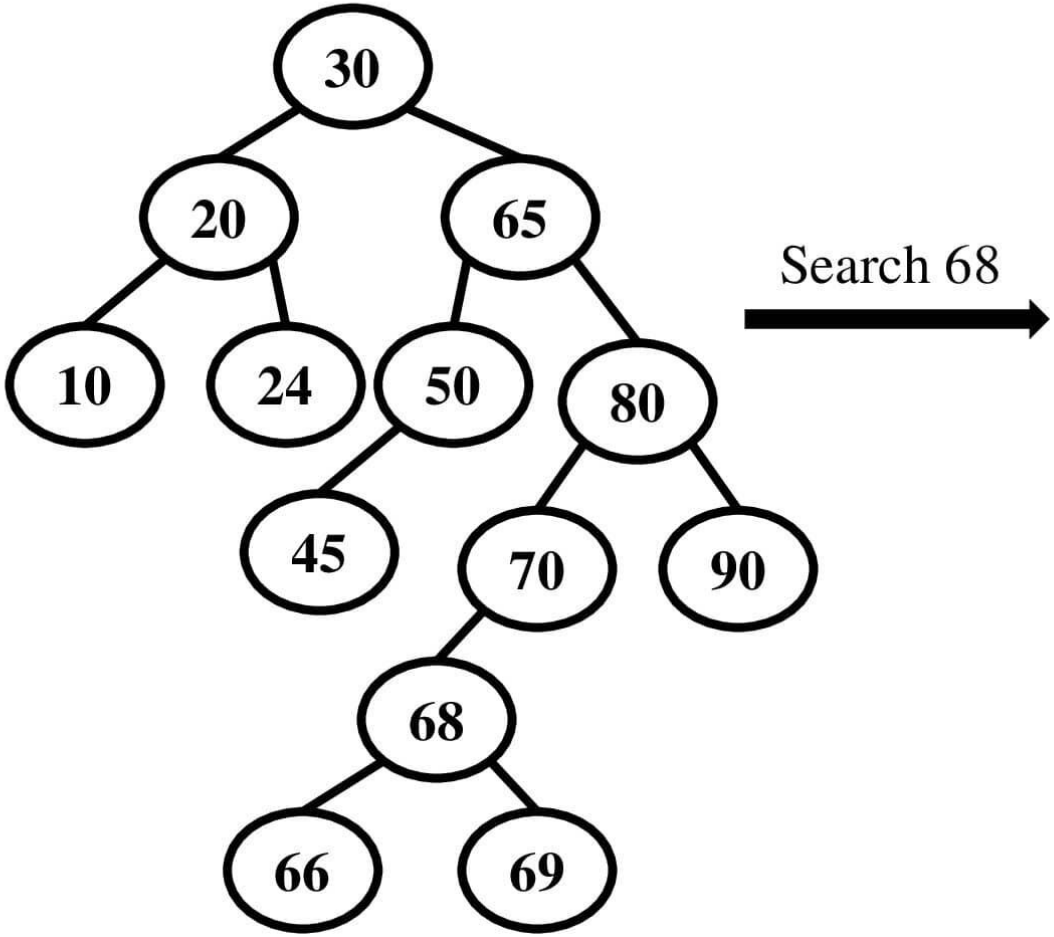
- The time complexity of searching, insertion and deletion = O(h)

  where h is the height of the tree.

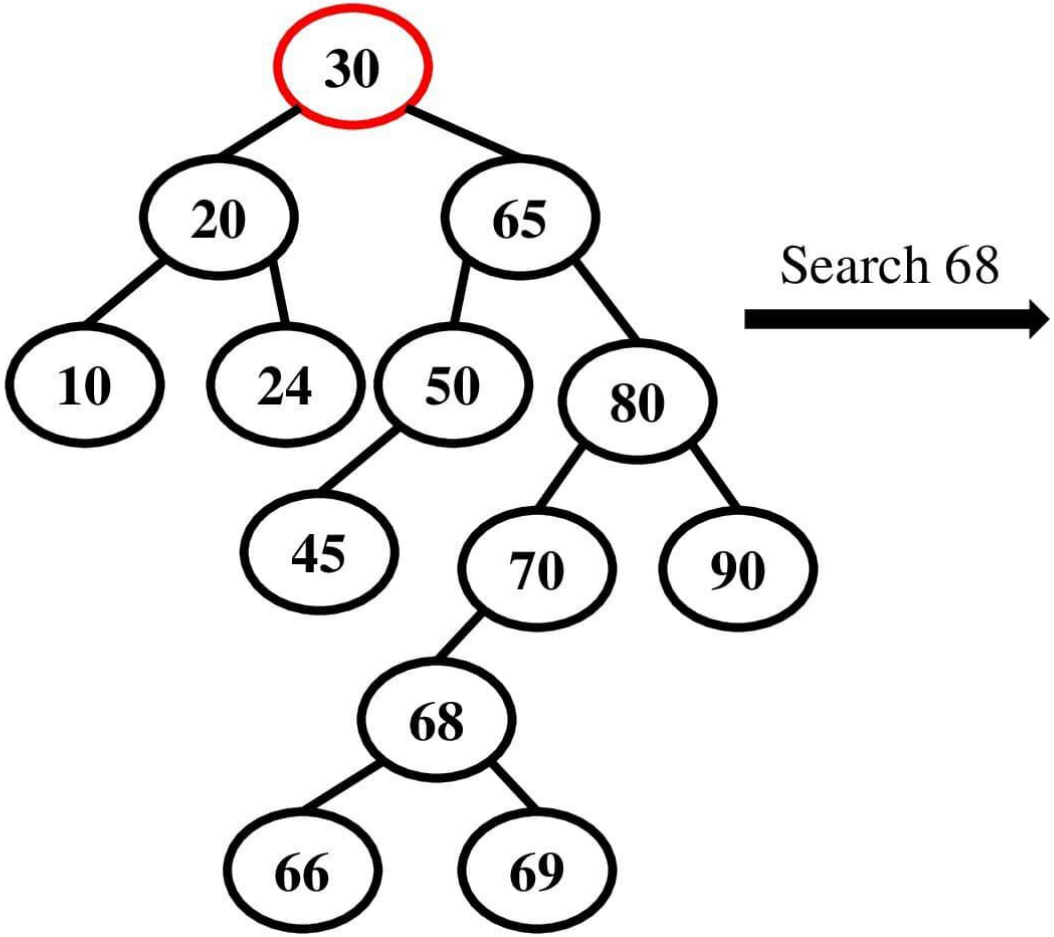
Binary Search Trees: Searching

- Let item is the value to be searched
- Search begins at root
- If root is NULL search unsuccessful
- Else compare item with root
- If item is less than root then only the left sub tree is to be searched. The sub tree may be searched recursively
- If item is greater than root then only the right sub tree is to be searched. The sub tree may be searched recursively

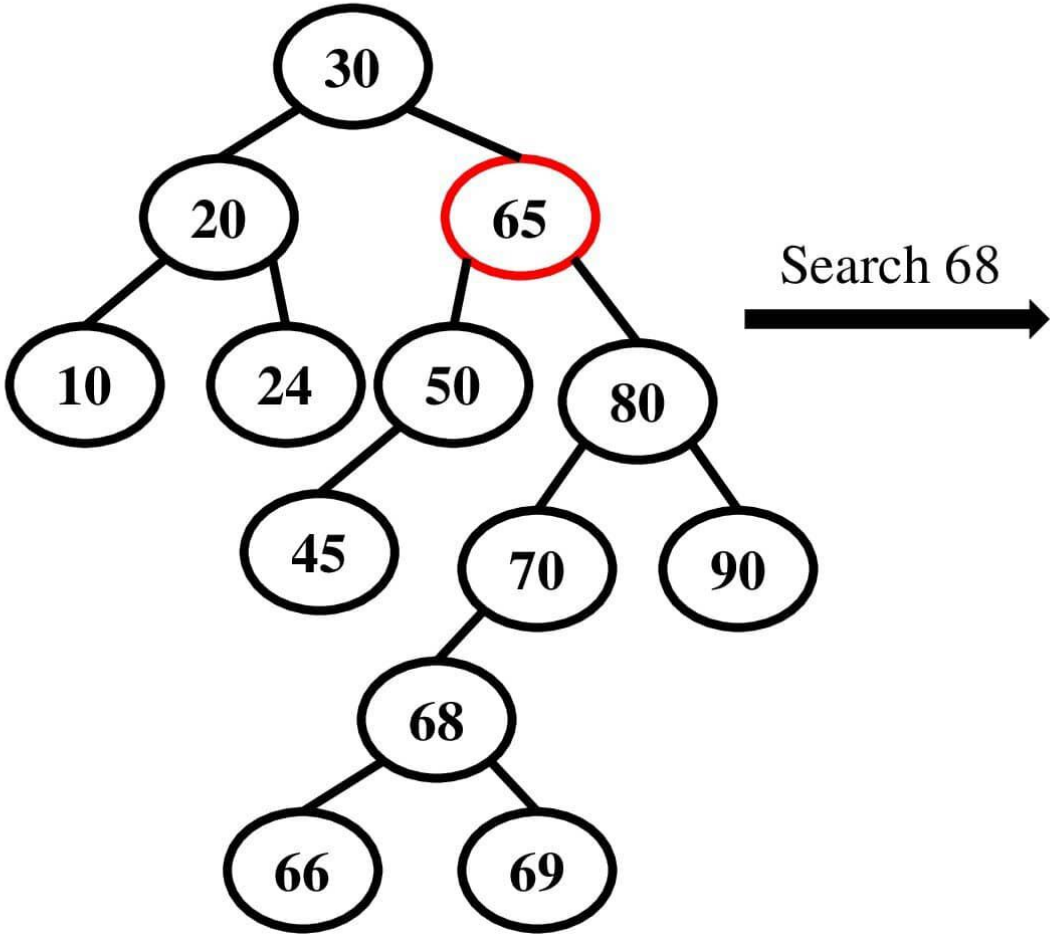
Binary Search Trees: Searching



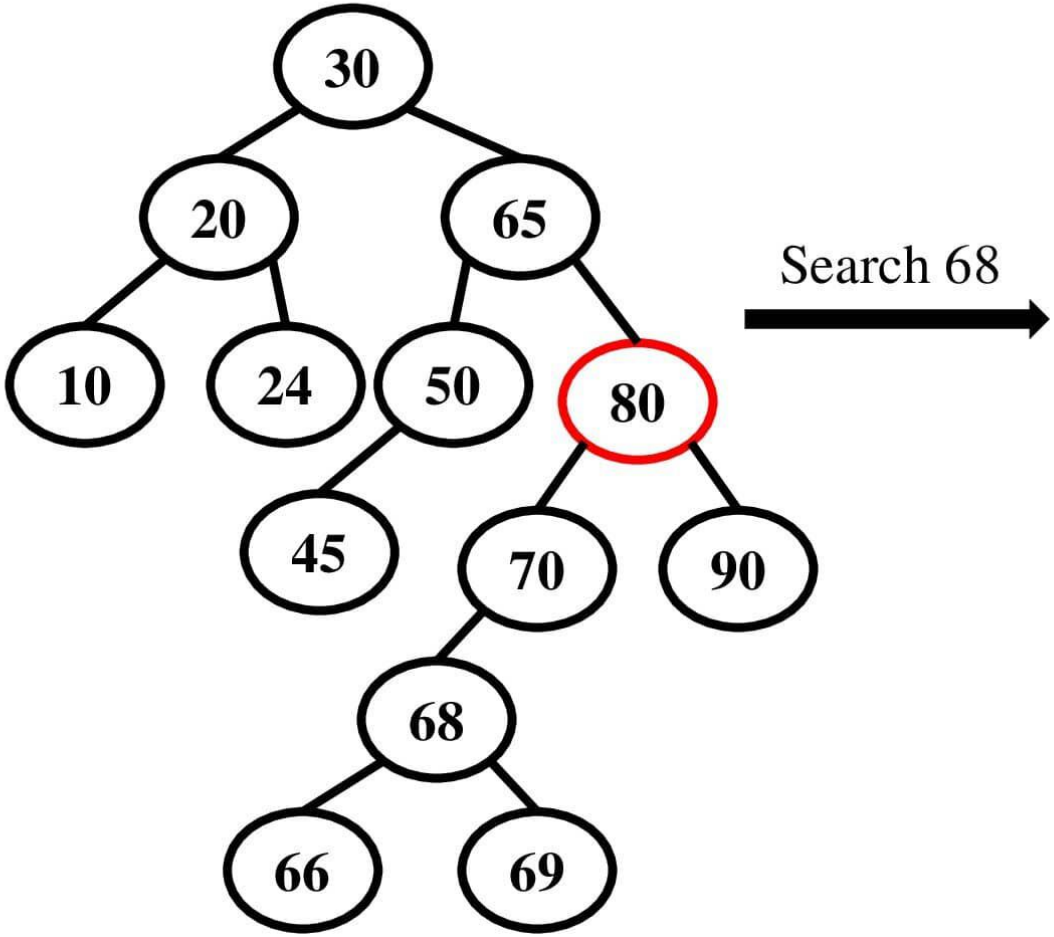
Binary Search Trees: Searching



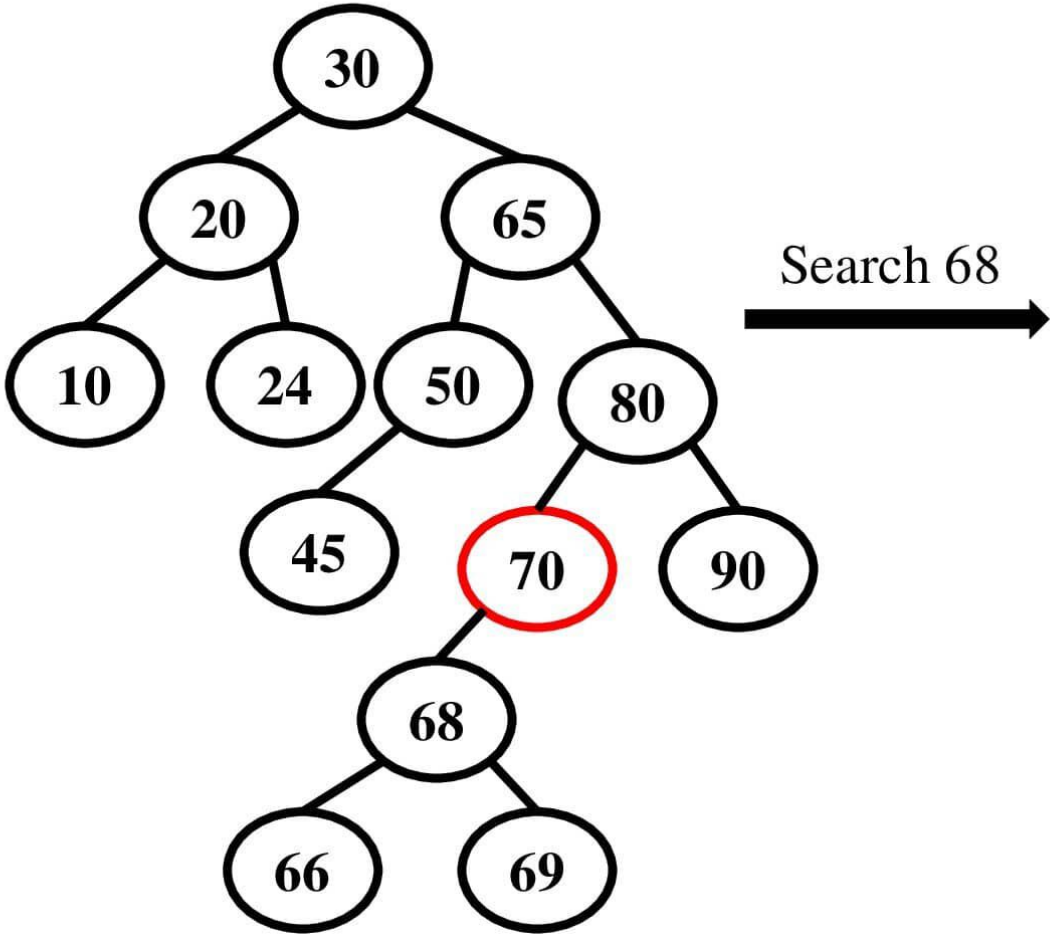
Binary Search Trees: Searching



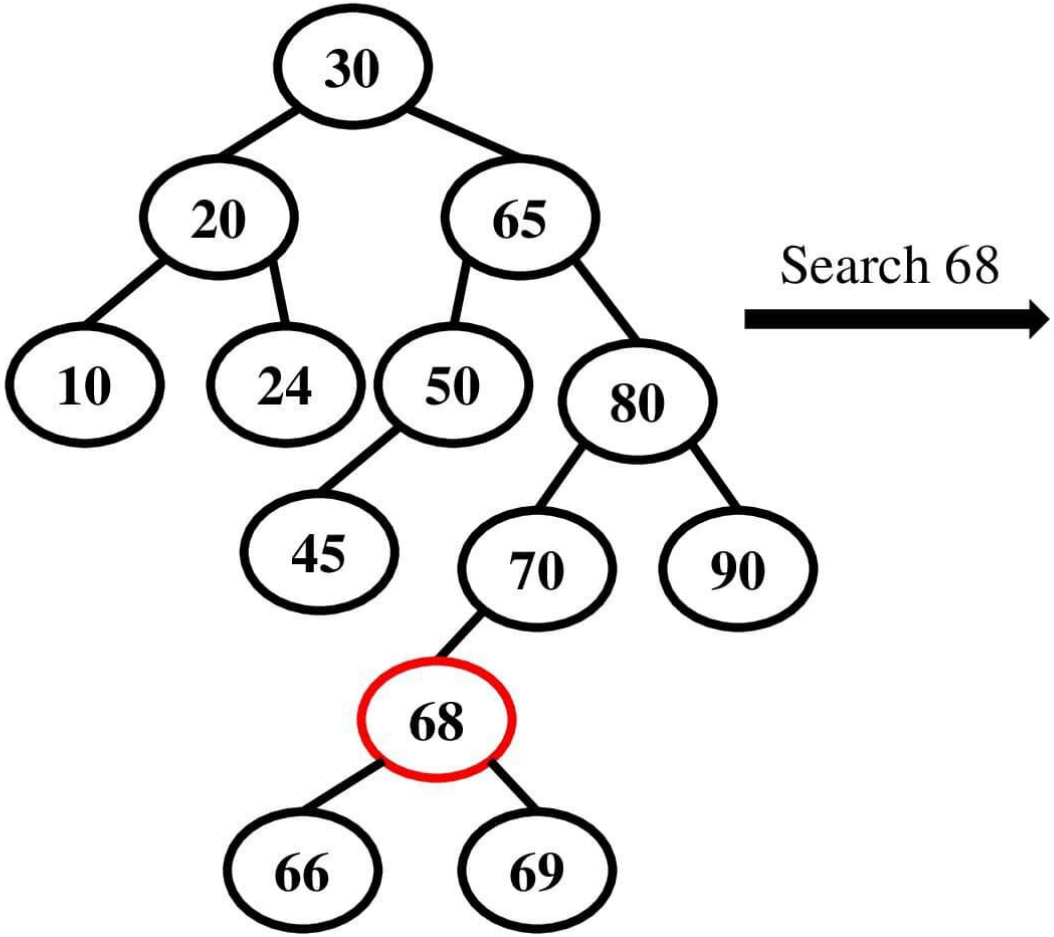
Binary Search Trees: Searching



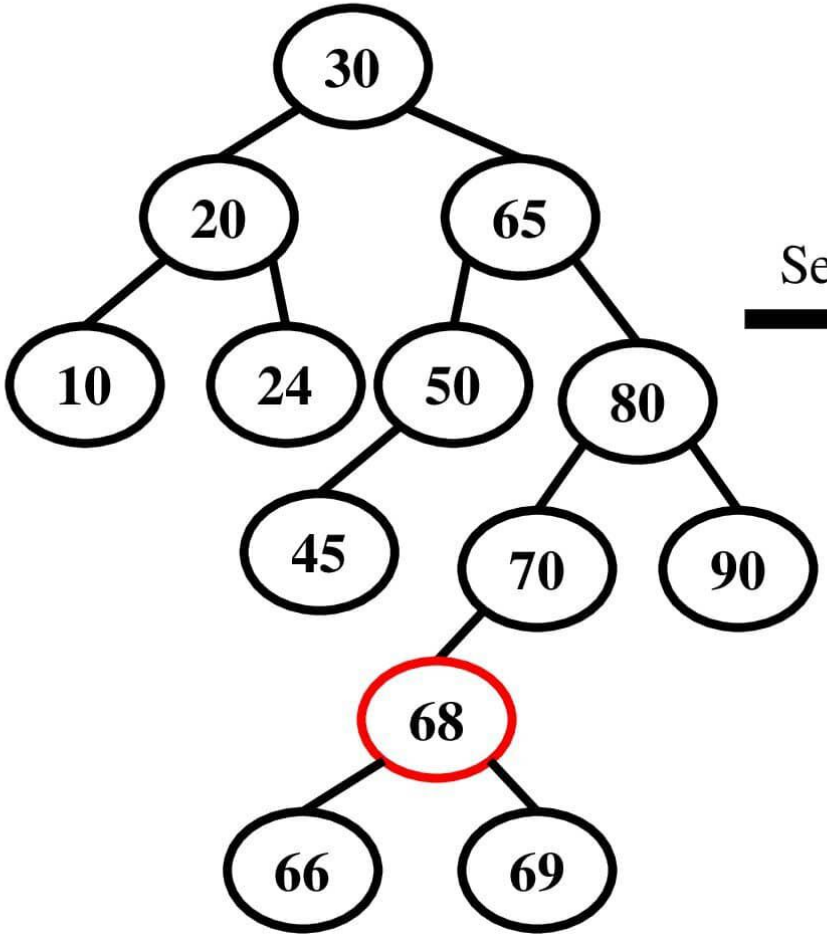
Binary Search Trees: Searching



Binary Search Trees: Searching



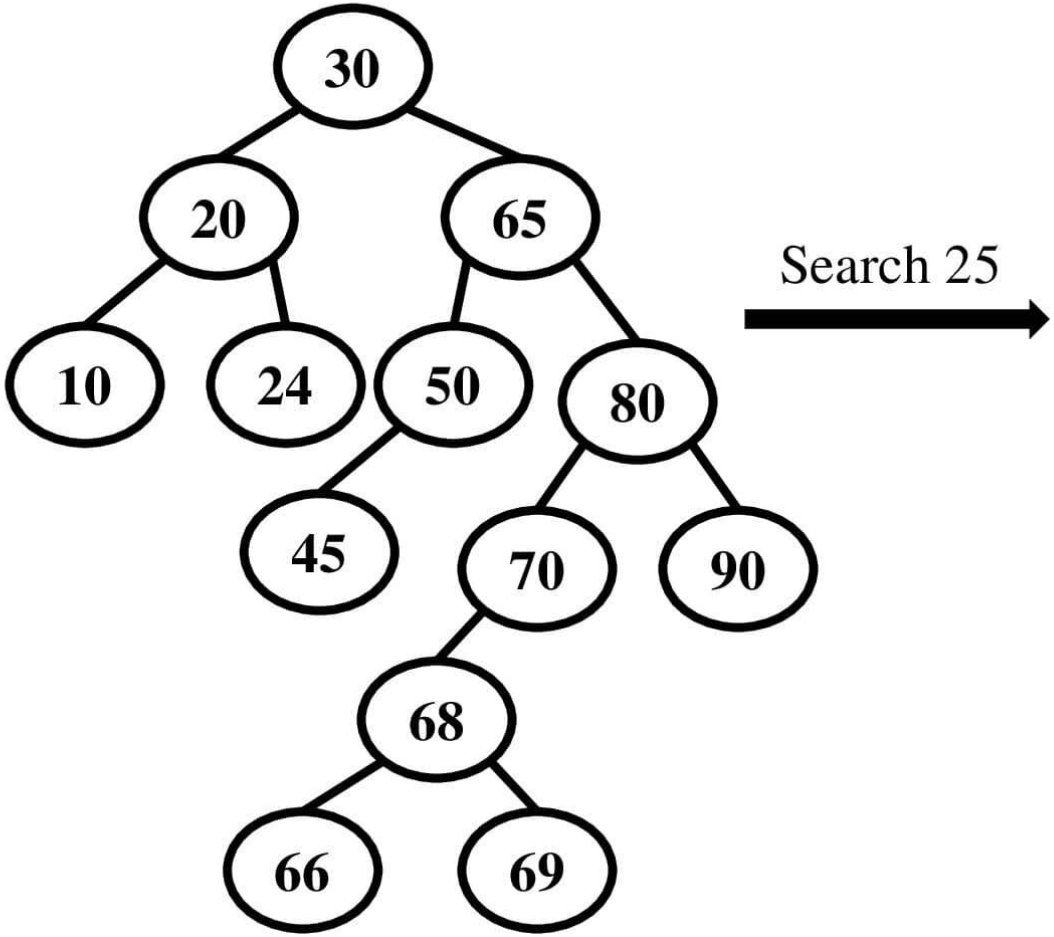
Binary Search Trees: Searching



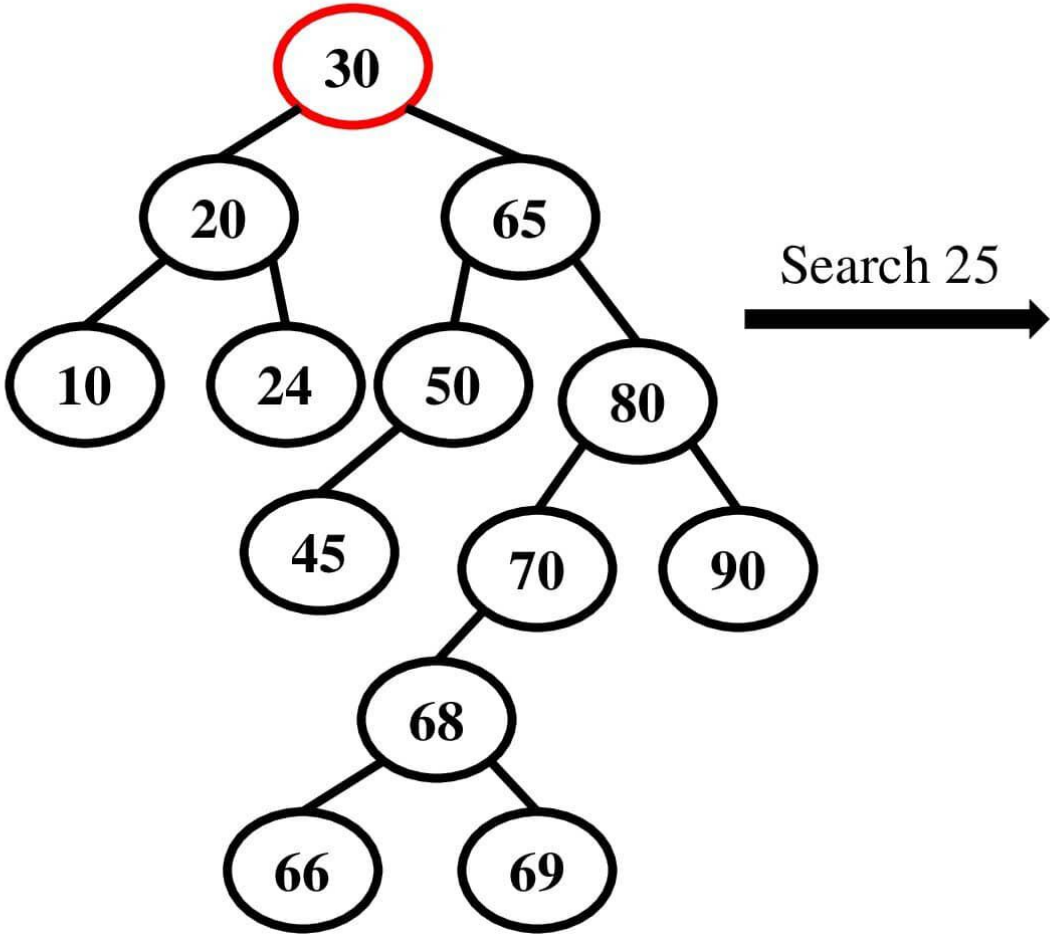
Search 68

Search Data Found

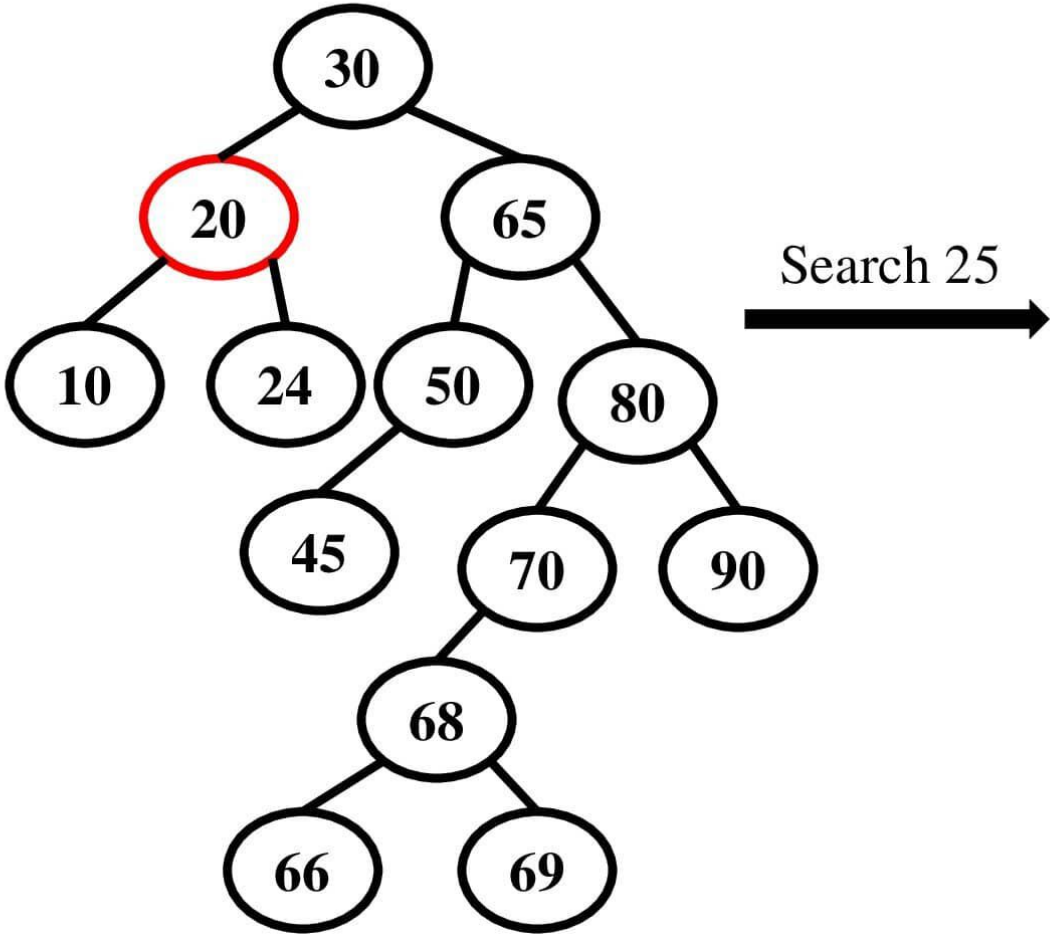
Binary Search Trees: Searching



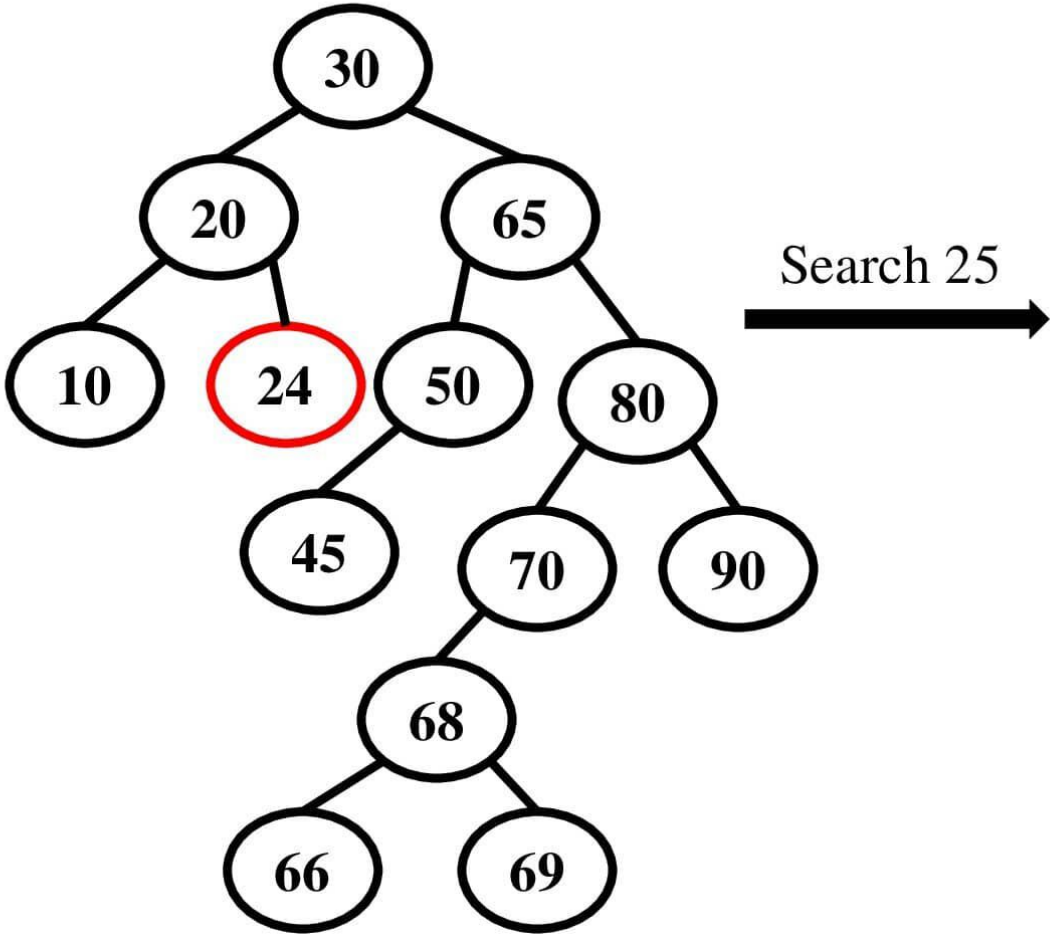
Binary Search Trees: Searching



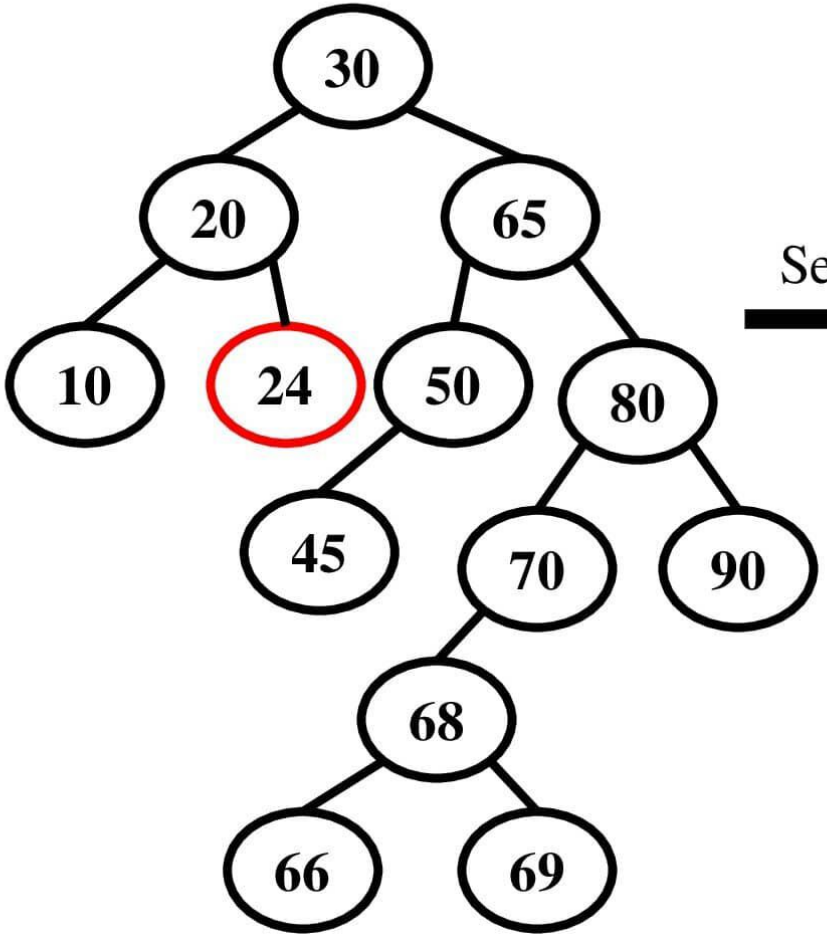
Binary Search Trees: Searching



Binary Search Trees: Searching



Binary Search Trees: Searching



Search 25 → Search Data Not Found

Algorithm BST_Search(item)

1. ptr=root
2. flag=0
3. While ptr!=NULL and flag=0 do
    1. If ptr→data=item then
        1. flag=1
    2. Else if ptr→data<item then
        1. ptr=ptr→rChild
    3. Else
        1. ptr=ptr→lChild
4. If flag=1 then
    1. Print “Search data found”
5. Else
    1. Print “Search data not found”

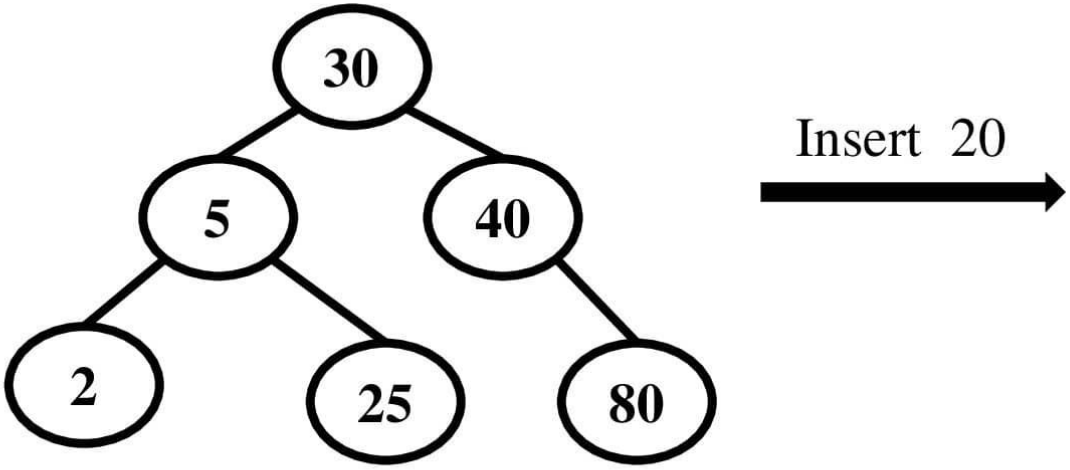
Binary Search Trees: Insertion

- Suppose 'item' is to be inserted in the BST
- Search for item
- If item is found then do nothing
- Else the item is inserted as left or right child where the search halts

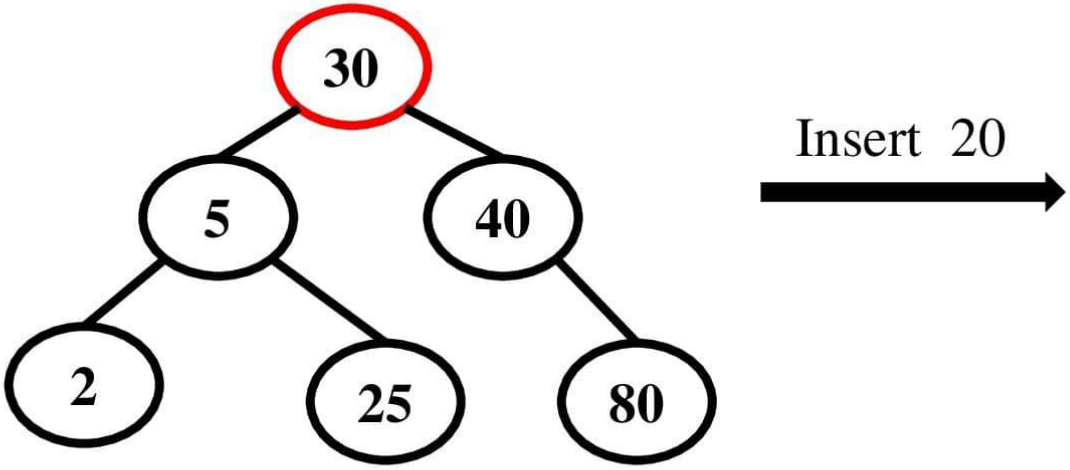
Algorithm BST_Insertion(item)

1. If ROOT=NULL then
    1. Create a node new and insert item in to it. Set it as ROOT node
2. Else
    1. Search item in the tree
    2. If search data found, insertion is not possible
    3. Else
        1. Find the parent node in which the item is to be inserted
        2. Create a node new and insert item in to it.
        3. If item<parent→data then
            1. Insert new as the left child of parent
        4. Else
            1. Insert new as the right child of parent

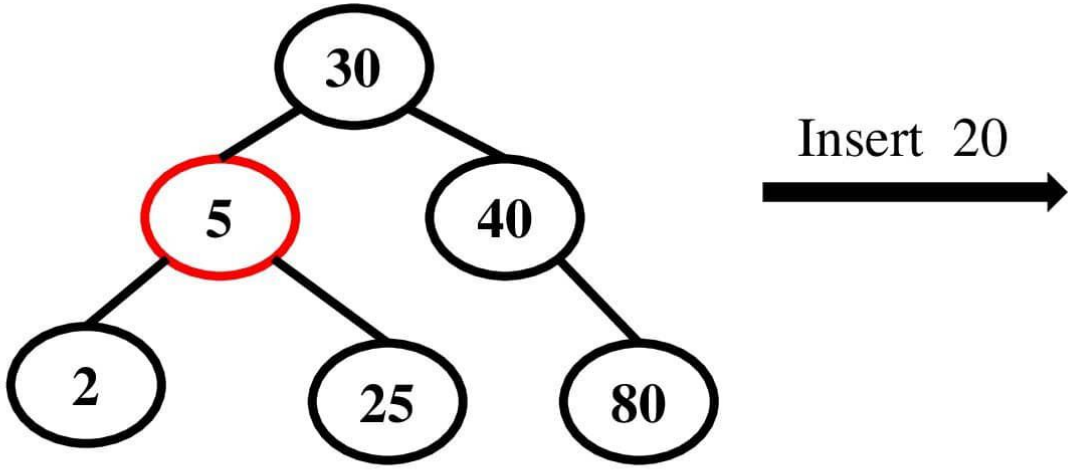
Binary Search Trees: Insertion



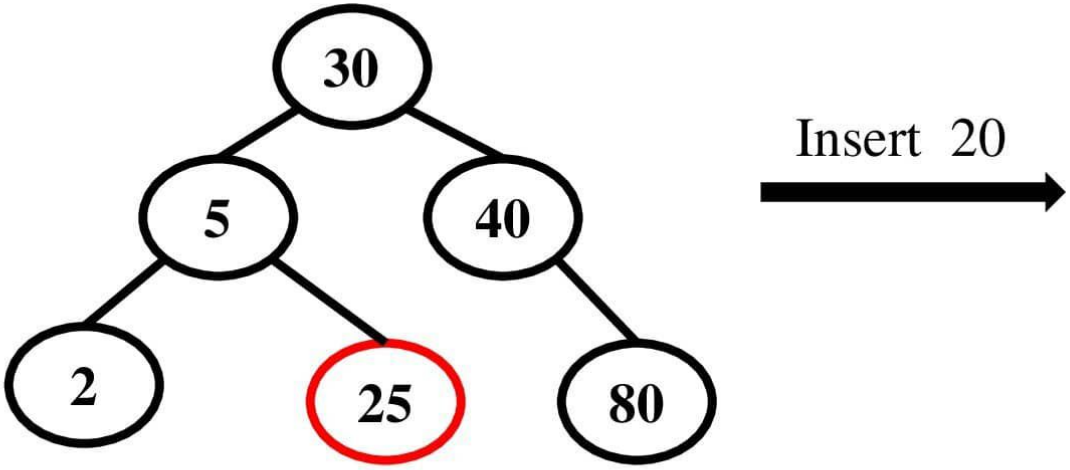
Binary Search Trees: Insertion



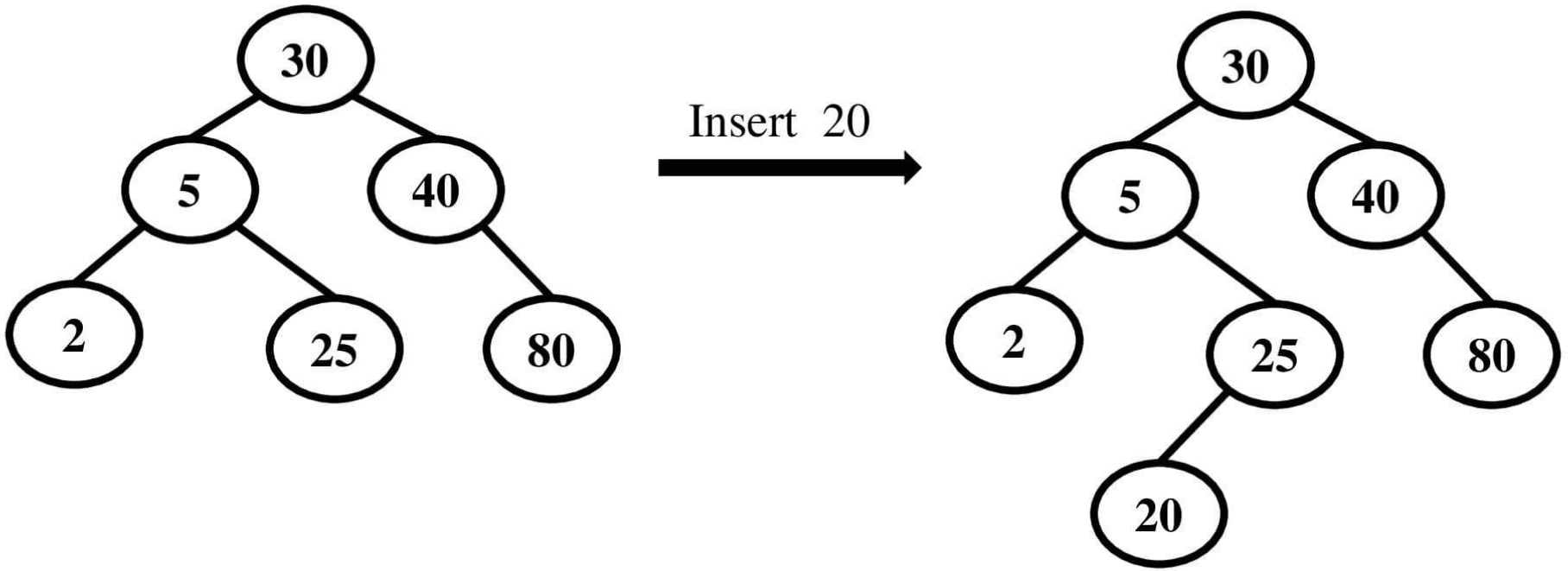
Binary Search Trees: Insertion



Binary Search Trees: Insertion



Binary Search Trees: Insertion



Algorithm BST_Insertion(item)

1. If ROOT=NULL then
    1. Create a node new
    2. new→data=item
    3. new→lChild=new→rChild=NULL
    4. ROOT=new
2. Else
    1. ptr=ROOT
    2. flag=0
    3. While ptr!=NULL and flag=0 do
        1. If item = ptr→data then
            1. flag=1
            2. Print “item al ready exist”
            3. Exit

2. Else If item < ptr→data then
    1. parent=ptr
    2. ptr=ptr→lChild
3. Else if item > ptr→data
    1. parent=ptr
    2. ptr=ptr→rChild
4. If ptr=NULL then
    1. Create a node new
    2. new→data=item
    3. new→lChild=new→rChild=NULL
    4. If parent→data < item then
        1. parent→rChild=new
    5. Else
        1. parent→lChild=new

Binary Search Trees: Deletion

Binary Search Trees: Deletion

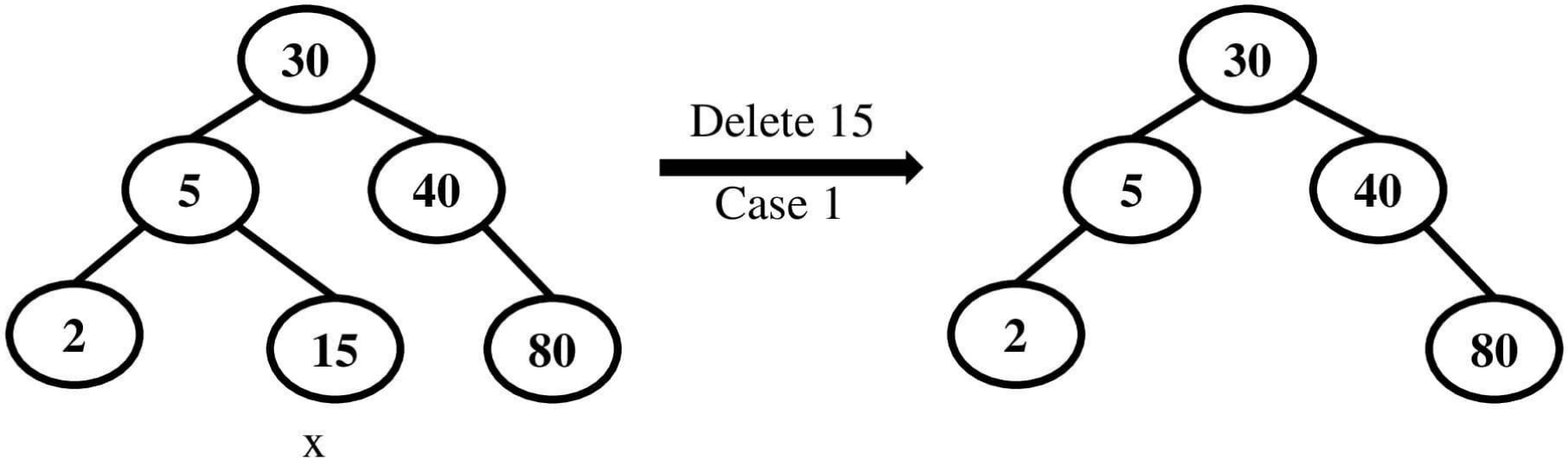
Algorithm BST_Deletion()

1. Let x be a node to be deleted
2. If x is a leaf node then
    1. Delete x
3. Else if x is a node with only one child then
    1. Replace x with the child node of x
    2. Delete x
4. Else if x having two children then
    1. Find the inorder successor/predecessor of x, say y
    2. Find the right/left child of y, say z
    3. Copy the data from y to x
    4. Replace y by z

Binary Search Trees: Deletion

Case 1: x is a leaf node

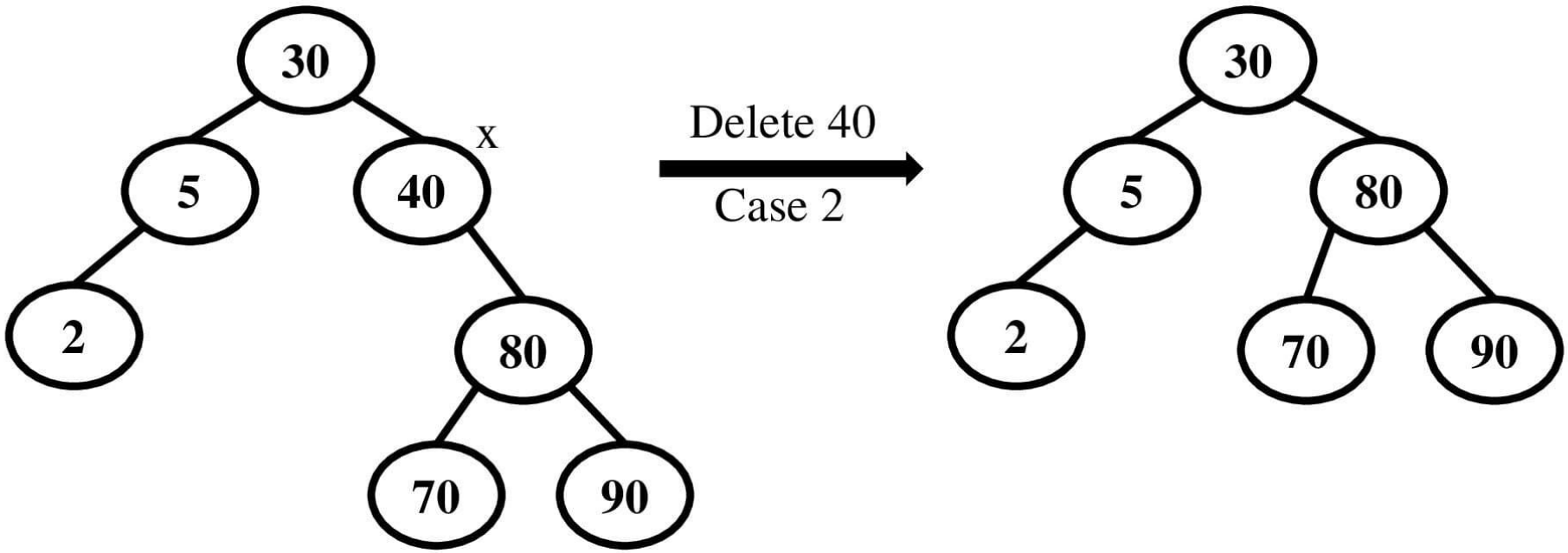
- Delete x



Binary Search Trees: Deletion

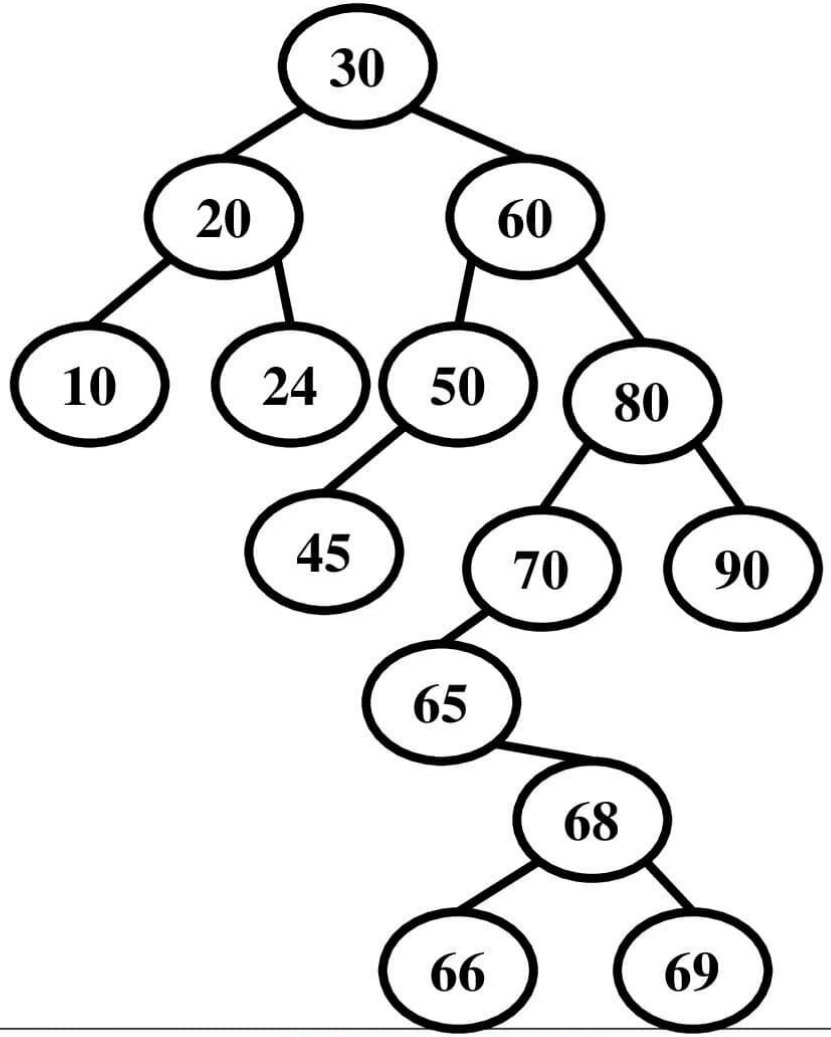
Case 2: x is a node with only one child

- Replace x with the child node of x
- Delete x



Case 3: x having two children

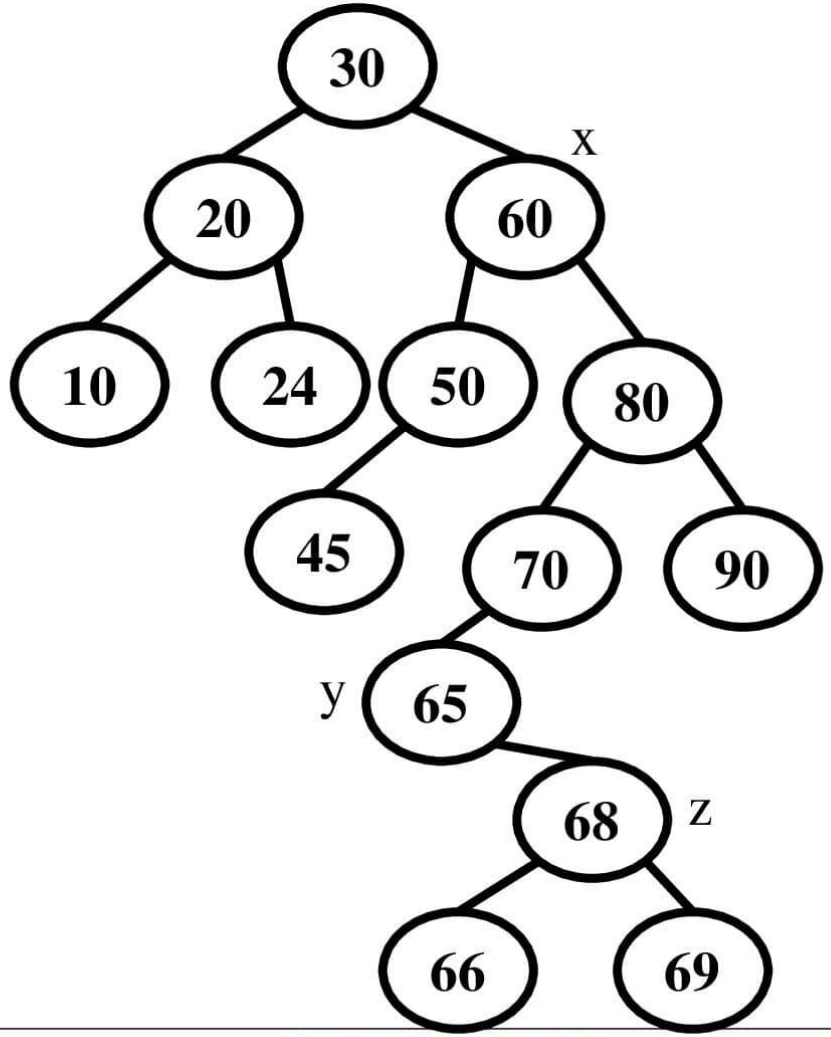
- Find the inorder successor, say y
- Find the right, say z
- Copy the data from y to x
- Replace y by z



Delete 60

Case 3: x having two children

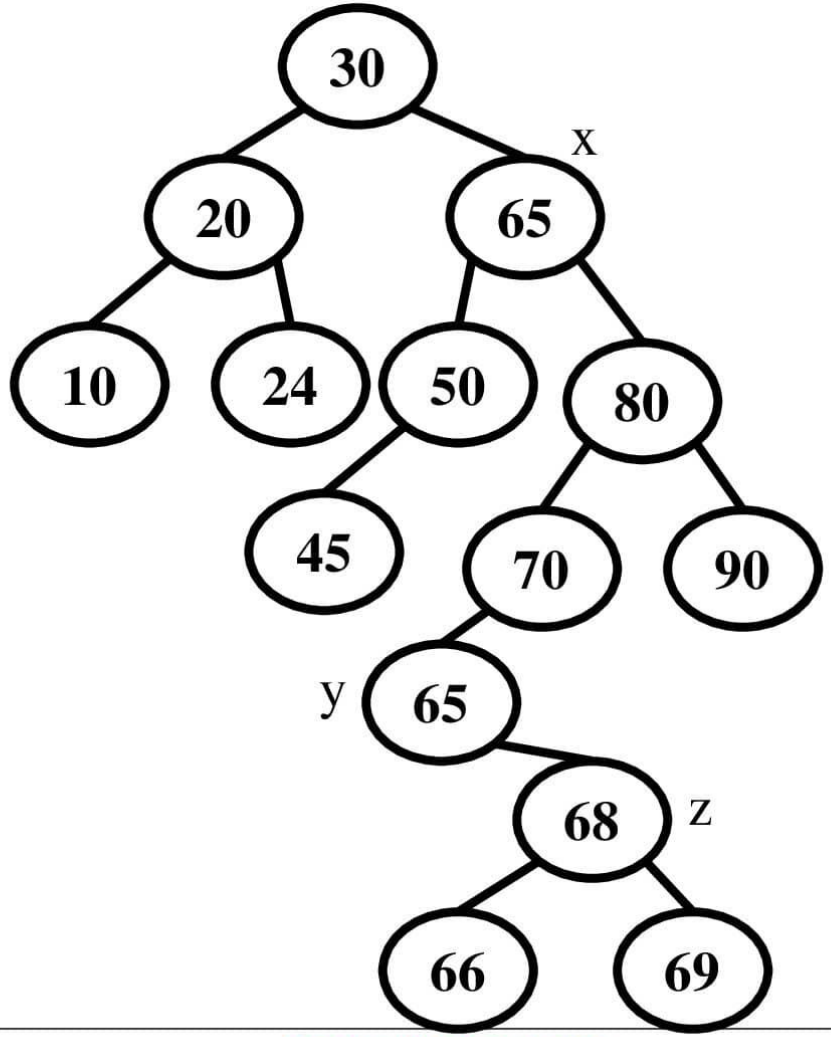
- Find the inorder successor, say y
- Find the right, say z
- Copy the data from y to x
- Replace y by z



Delete 60

Case 3: x having two children

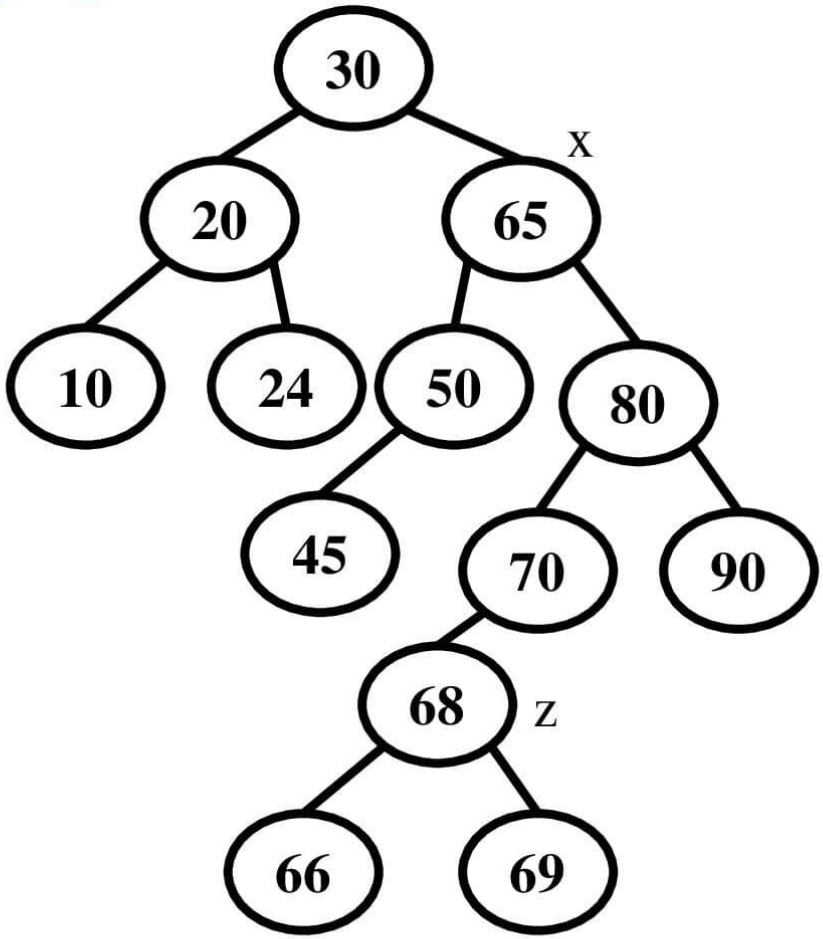
- Find the inorder successor, say y
- Find the right, say z
- Copy the data from y to x
- Replace y by z



Delete 60

Case 3: x having two children

- Find the inorder successor, say y
- Find the right, say z
- Copy the data from y to x
- Replace y by z



Delete 60

Algorithm BST_Deletion(item)

1. ptr=ROOT, flag=0
2. While ptr!=NULL and flag=0 do
    1. If item < ptr→data then
        1. parent=ptr
        2. ptr=ptr→lChild
    2. Else if item > ptr→data then
        1. parent=ptr
        2. ptr=ptr→rc
    3. Else
        1. flag=1
3. If (flag==0)
    1. Print "ITEM does not exist"
    2. Exit

4. Else
    1. If ptr→rChild=NULL and ptr→lChild=NULL then
        1. If parent→lChild=ptr then parent→lChild=NULL
        2. Else parent→rChild=NULL
    2. Else if ptr→lChild!=NULL and ptr→rChild!=NULL then
        1. y is inorder successor of ptr
        2. p is the parent of y
        3. z is the right child of y
        4. Copy the data from y to ptr
        5. p→lChild=z
        6. Dispose(y)

3. Else
    1. If parent→lChild=ptr then
        1. If ptr→lChild!=NULL then
            1. parent→lChild=ptr→lChild
        2. Else
            1. parent→lChild=ptr→rChild
    2. Else
        1. If ptr→lChild!=NULL then
            1. parent→rChild=ptr→lChild
        2. Else
            1. parent→rChild=ptr→rChild

Binary Search Trees: Applications

- **Remove duplicate values**
  - Consider a collection of n data items A_1,A_2, …, A_N. Suppose we want to find and delete all duplicates in the collection. Then, we can represent them as BST and remove the duplicate values, while it occurs.
- **Find the smallest data**- Traverse the left subtree of BST
- **Find the largest data**- Traverse the right subtree of BST
- **Find a particular data**- Traverse the whole BST