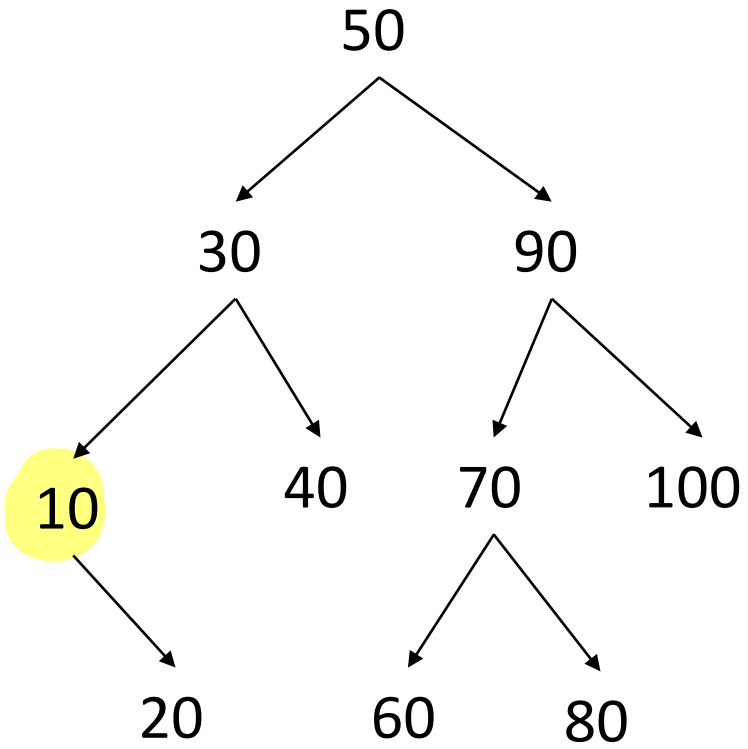

# Data Structure & Algorithms

Sunbeam Infotech

# Agenda

- BST – In-order traversal (non-recursive)
- BST – Post-order traversal (non-recursive)
- BST – BFS & DFS (non-recursive)
- BST – Delete node
- BST – Balance tree
- BST – Types
  - Skewed Binary Tree
  - AVL Tree
  - R & B Tree

# BST – Non-Recursive Algorithm – PreOrder

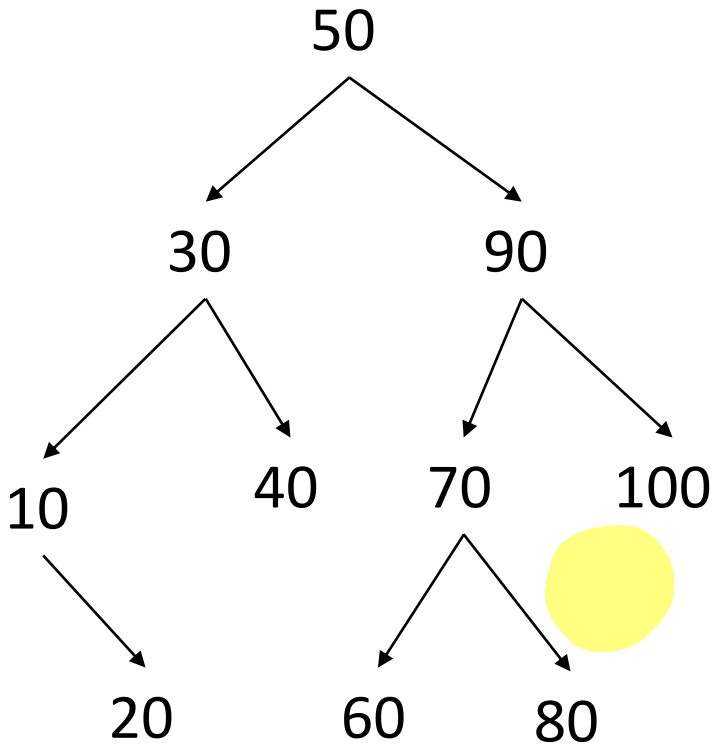



until trav became null & stack empty

until null is reached
visit trav
if trav has right,
push it on stack.
go to left

if stack is not empty
pop node from stack
into trav.

# BST – Non-Recursive Algorithm – PreOrder



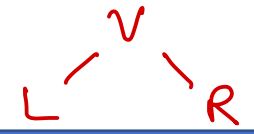

50 30 10 20 40

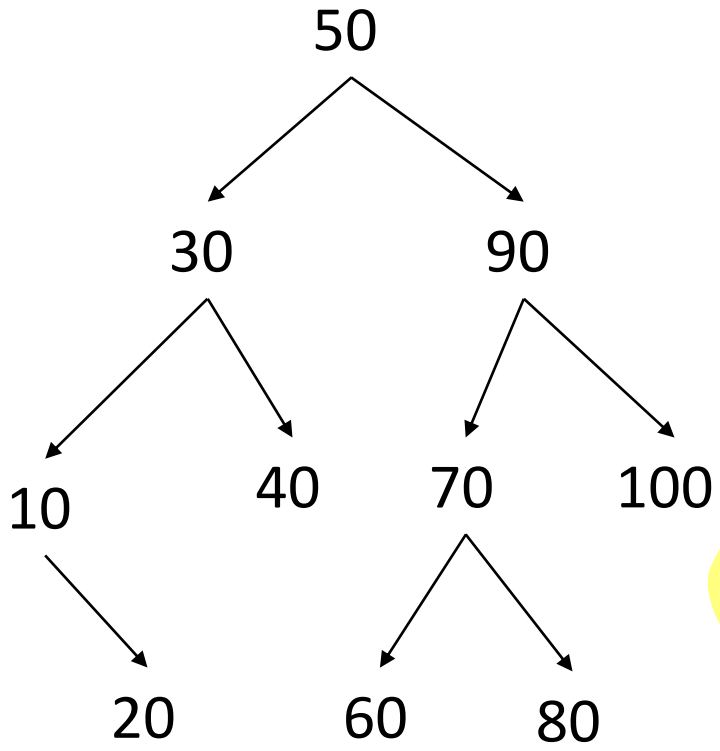
90 70 60 80 100

```
node* trav = root;
while (trav != NULL || !s.empty()) {
    while (trav != NULL) {
        cout << trav->data << ", "; //visit
        if (trav->right != NULL) // push right child
            s.push(trav->right);
        trav = trav->left; // go to left
    }
    if (!s.empty()) {
        trav = s.top(); // pop from stack
        s.pop();
    }
}
```

# BST – Non-Recursive Algorithm – InOrder

L V R




10 20 30 40 50

60 70 80 90 100

trav = root;

until trav is null or stack is empty:

until null is reached:

push trav on stack

go to the left

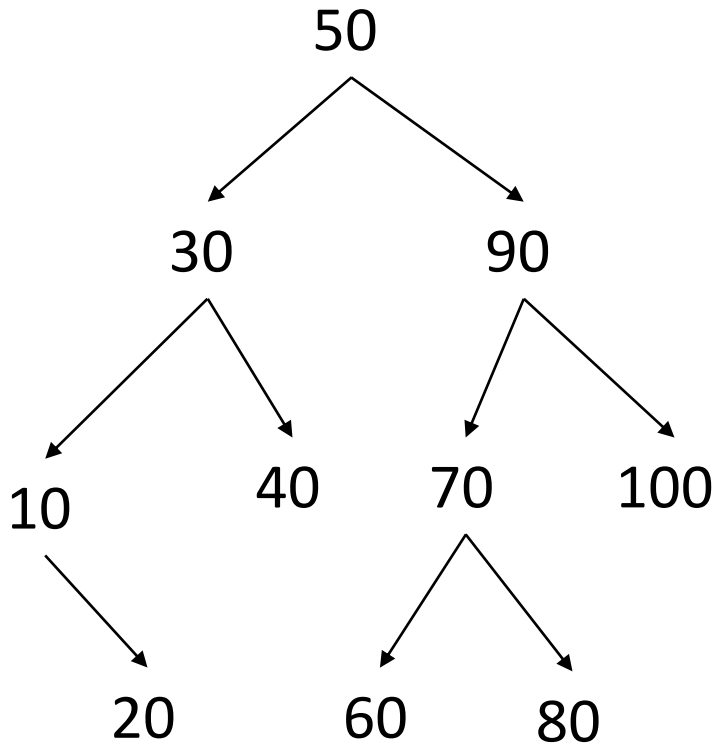
if stack is not empty:

pop trav from stack

visit trav

go to the right

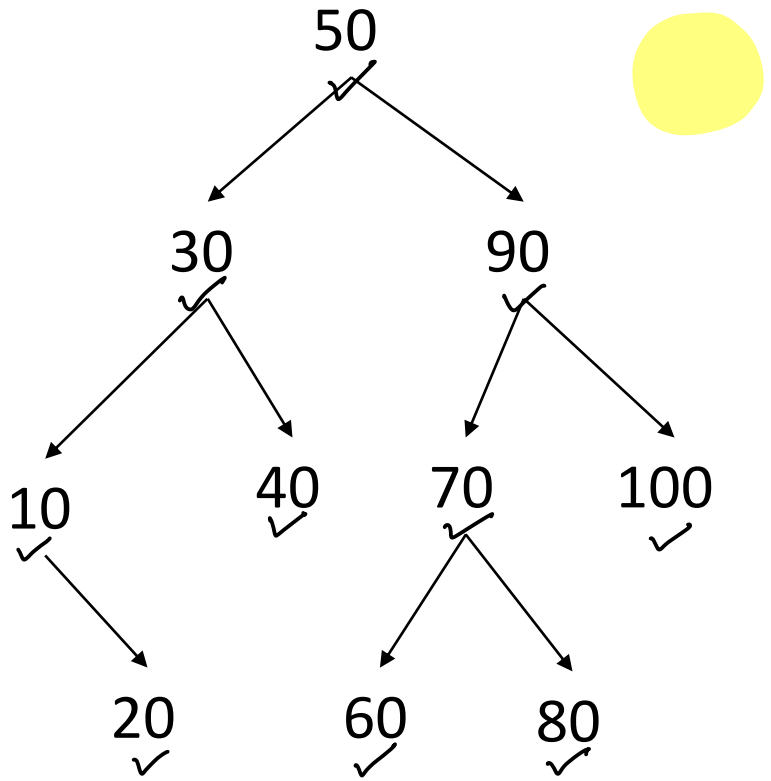
# BST – Non-Recursive Algorithm – InOrder




```
node* trav = root;
while (trav != NULL || !s.empty()) {
    while (trav != NULL) {
        s.push(trav);
        trav = trav->left;
    }
    if (!s.empty()) {
        trav = s.top();
        s.pop();
        cout << trav->data << ", ";
        trav = trav->right;
    }
}
```

# BST – Non-Recursive Algorithm – PostOrder

L R V

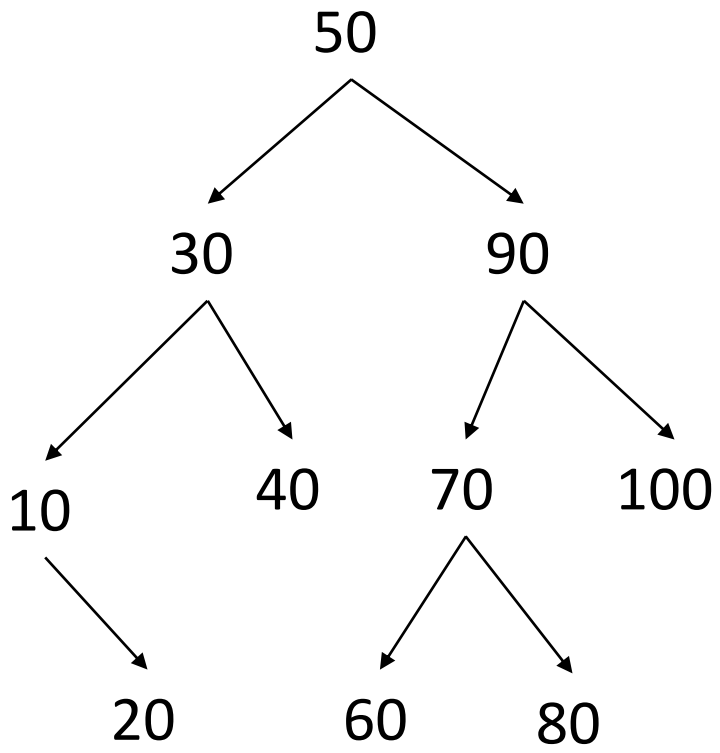




20 10 40 30 60

80 70 100 90 50

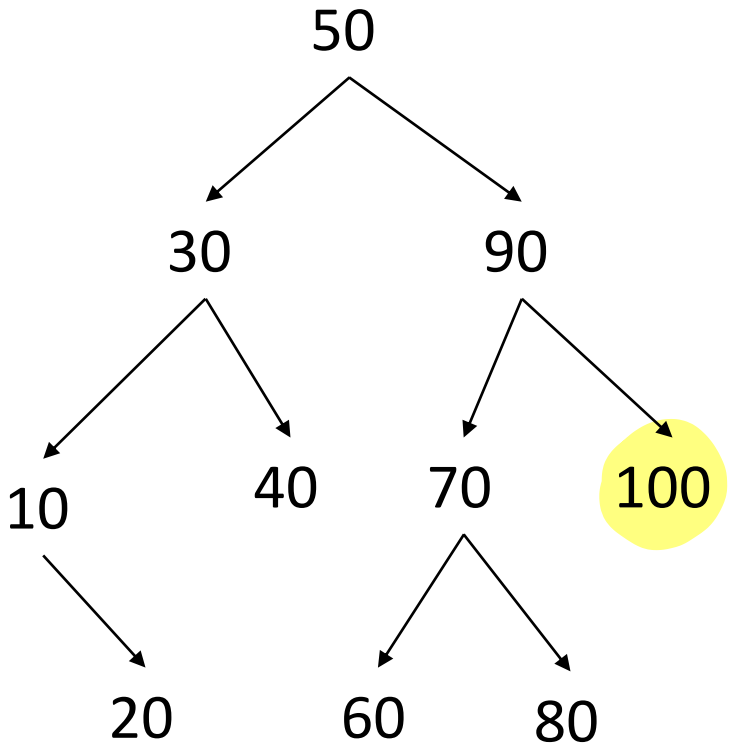
```
trav = root;
while trav not null or stack not empty:
    until null is reached:
        push trav on stack
        goto left of trav
    if stack is not empty:
        pop node into trav
        if right node is present & visited:
            print node
            mark node as visited
            make trav null.
        else:
            push node on stack
            go to the right
```

# BST – Non-Recursive Algorithm – PostOrder




```
node* trav = root;
while (trav != NULL || !s.empty()) {
    while (trav != NULL) {
        s.push(trav);
        trav = trav->left;
    }
    if (!s.empty()) {
        trav = s.top();    s.pop();
        if(trav->right != NULL && trav->right->visited) {
            cout << trav->data << ", ";
            trav->visited = true;
            trav = NULL;
        } else {
            s.push(trav);
            trav = trav->right;
        }
    }
}
```

# BST – Non-Recursive Algorithm – DFS - any binary tree (non-sorted)




50 30 10 20 40

90 70 60 80 100

push root on stack

while stack is not empty:

pop node into tmp

visit it

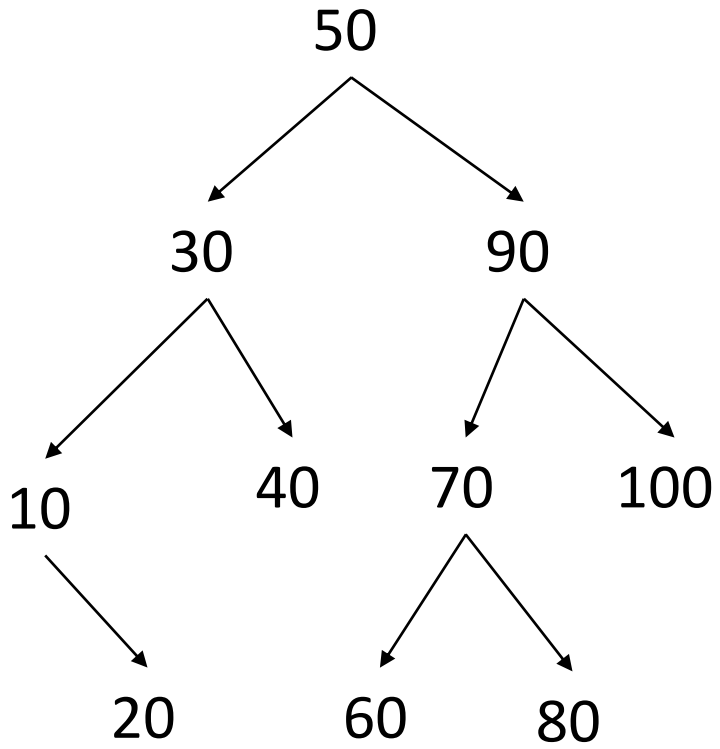
if tmp has right,

push it on stack

if tmp has left,

push it on stack.

# BST – Non-Recursive Algorithm – DFS

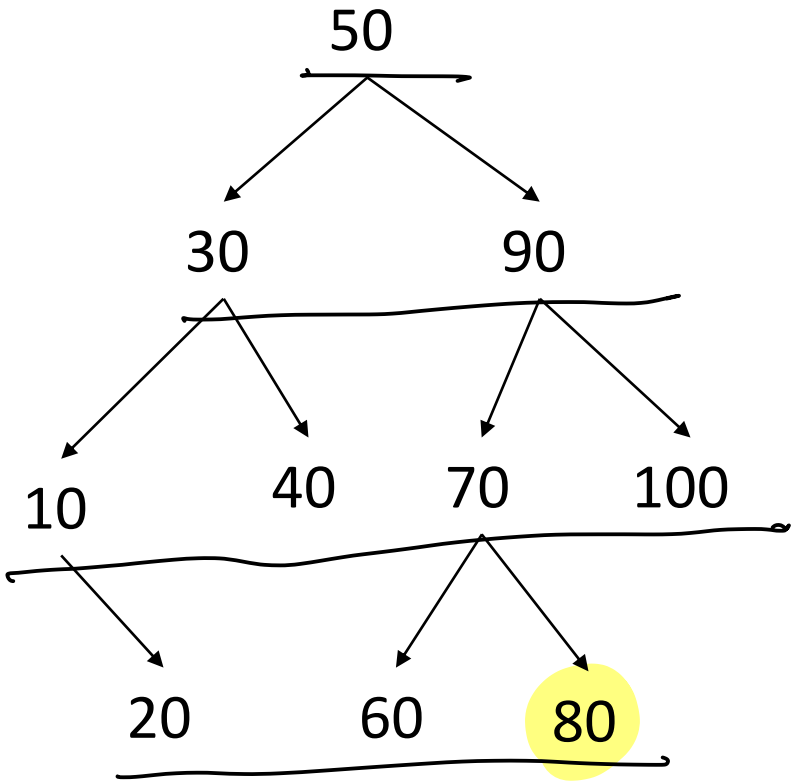



```
s.push(root);
while (!s.empty()) {
        trav = s.top();
        s.pop();
        cout << trav->data << ", ";
        if (trav->right != NULL)
            s.push(trav->right);
        if (trav->left != NULL)
            s.push(trav->left);
}
```

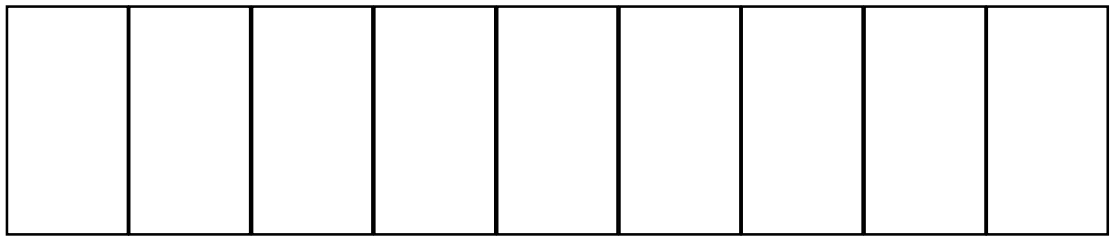
# BST – Non-Recursive Algorithm – BFS

- level wise search

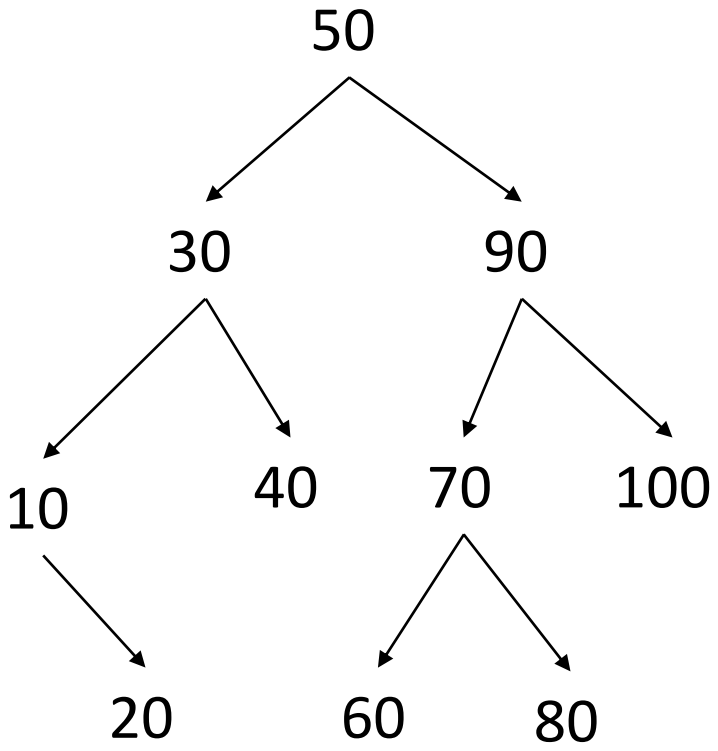
Btree

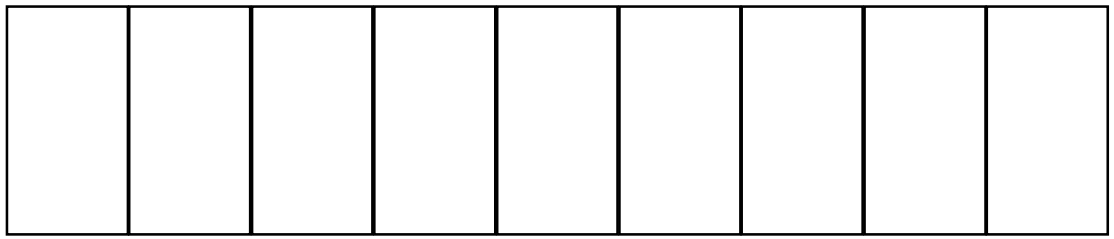



50 30 90 10 40 70 100 20 60 80



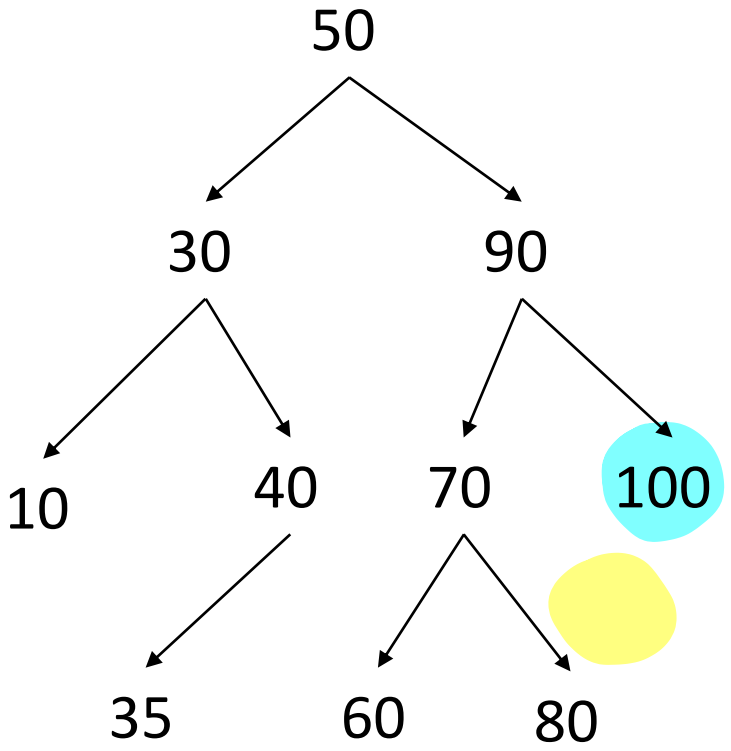
# BST – Non-Recursive Algorithm – BFS



```
q.push(root);
while (!s.empty()) {
    trav = q.front();
    q.pop();
    cout << trav->data << ", ";
    if (trav->left != NULL)
        q.push(trav->left);
    if (trav->right != NULL)
        q.push(trav->right);
}
```

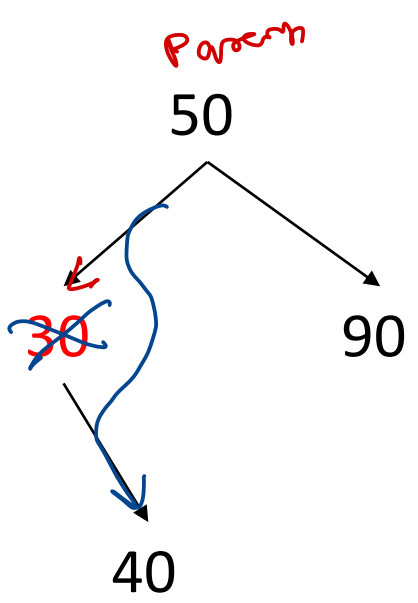
# BST – Find node with its Parent

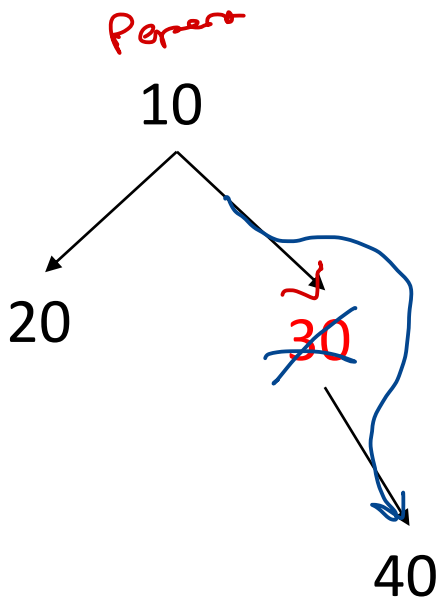


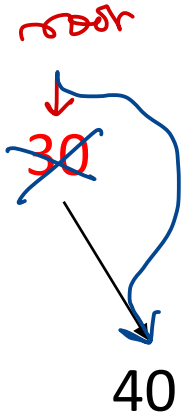

```
node* binsearch(int key , node **pparent ) {
    node* trav = root;
    *pparent = null;
    while (trav != NULL) {
        if (key == trav->data)
            return trav;
        *pparent = trav;
        if (key < trav->data)
            trav = trav->left;
        else
            trav = trav->right;
    }
    *pparent = NULL;
    return NULL;
}
```

```
node *temp, *parent;
temp = t.binsearch(val, &parent);
if (temp != null)
    cout << temp->data << parent->data;
```

# BST – Delete Node – whose left child is null. (temp)



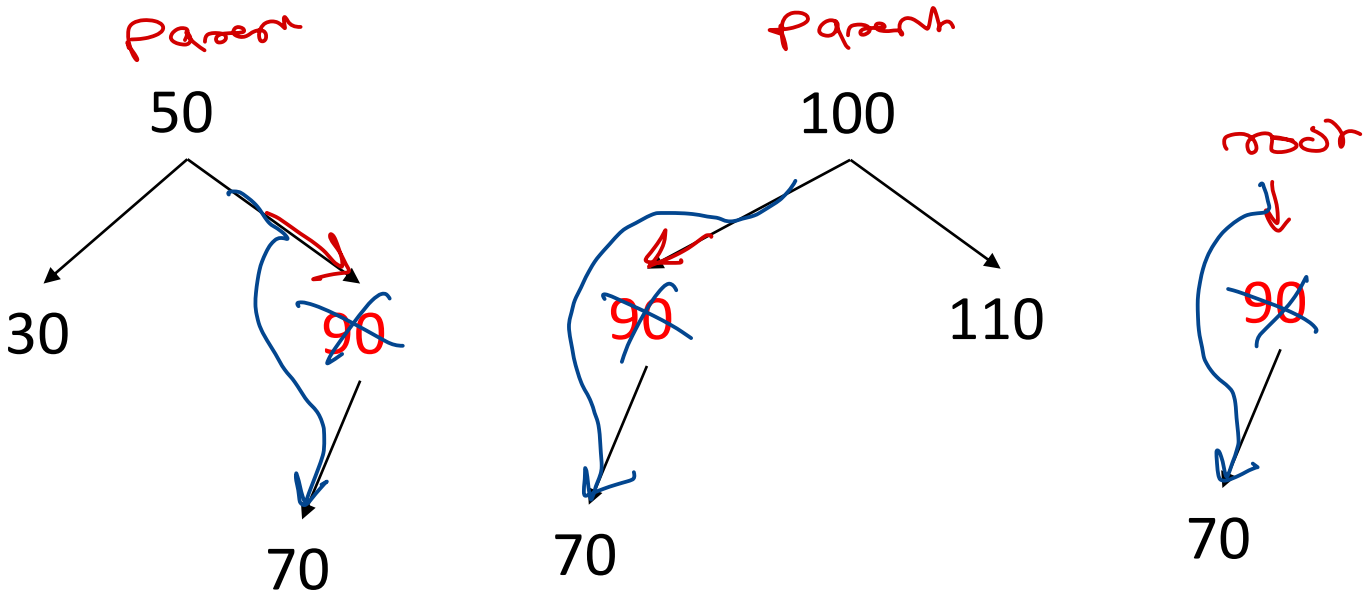

```
if (temp == root)
    root = temp->right;
else if (temp == parent->left)
    parent->left = temp->right;
else
    parent->right = temp->right;
delete temp;
```

```
parent->left = temp->right;
```

```
root = temp->right;
```

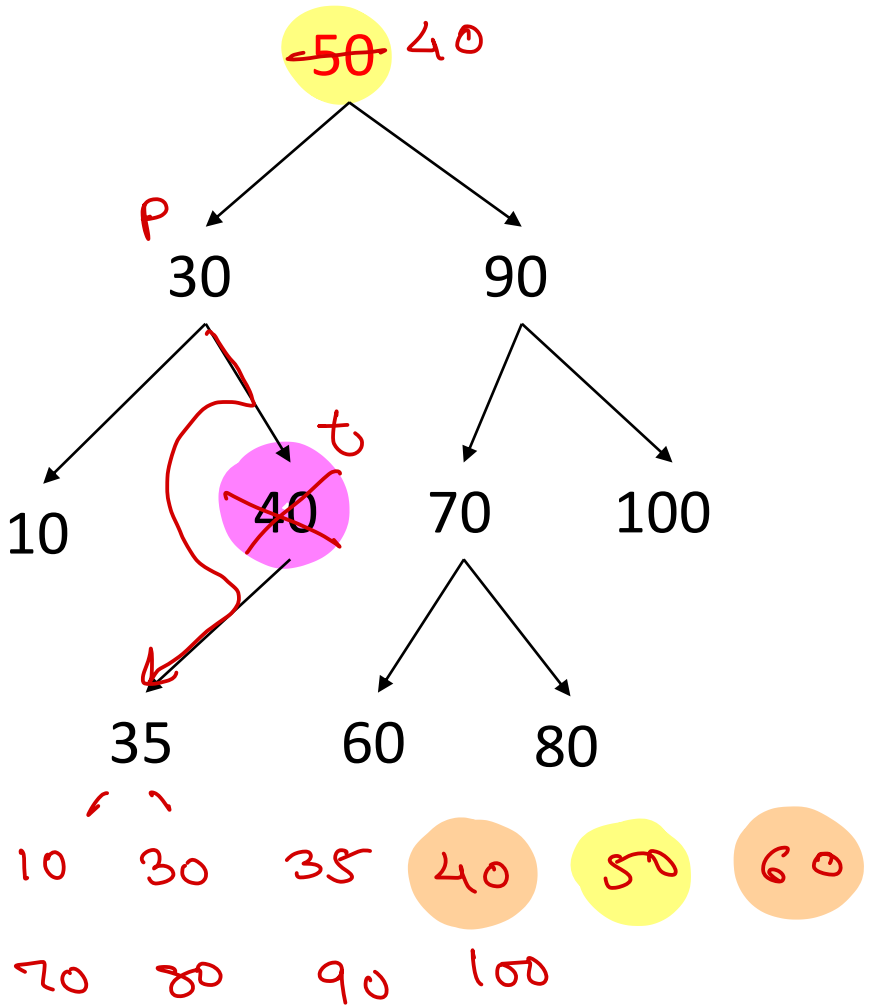
```
parent->right = temp->right;
```

# BST – Delete Node - whose right is null (temp)



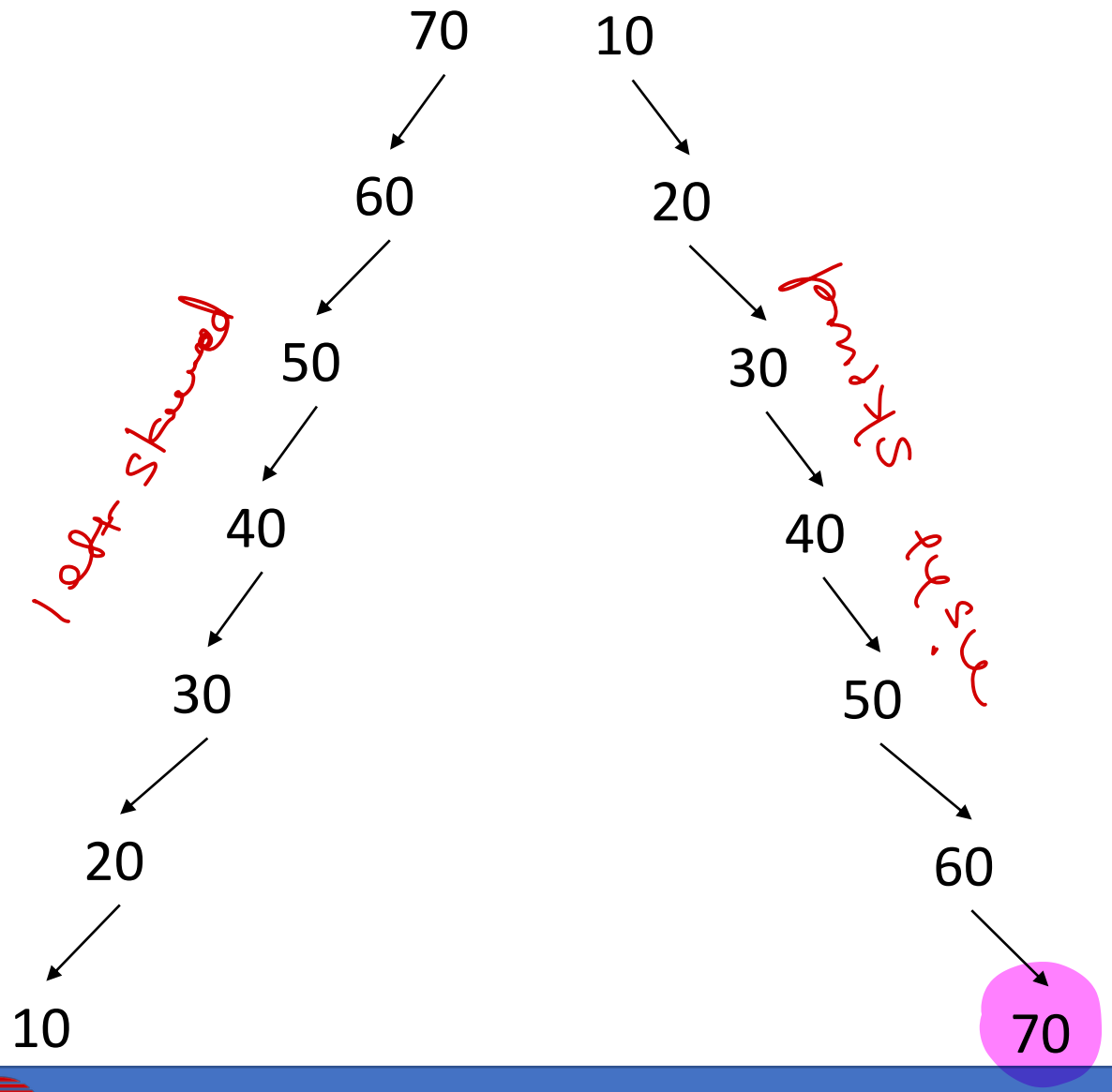

```
if (temp == root)
    root = temp->left;
else if (temp == parent->left)
    parent->left = temp->left;
else
    parent->right = temp->left;

delete temp;
```

# BST – Delete Node – whose left & right child present (temp)



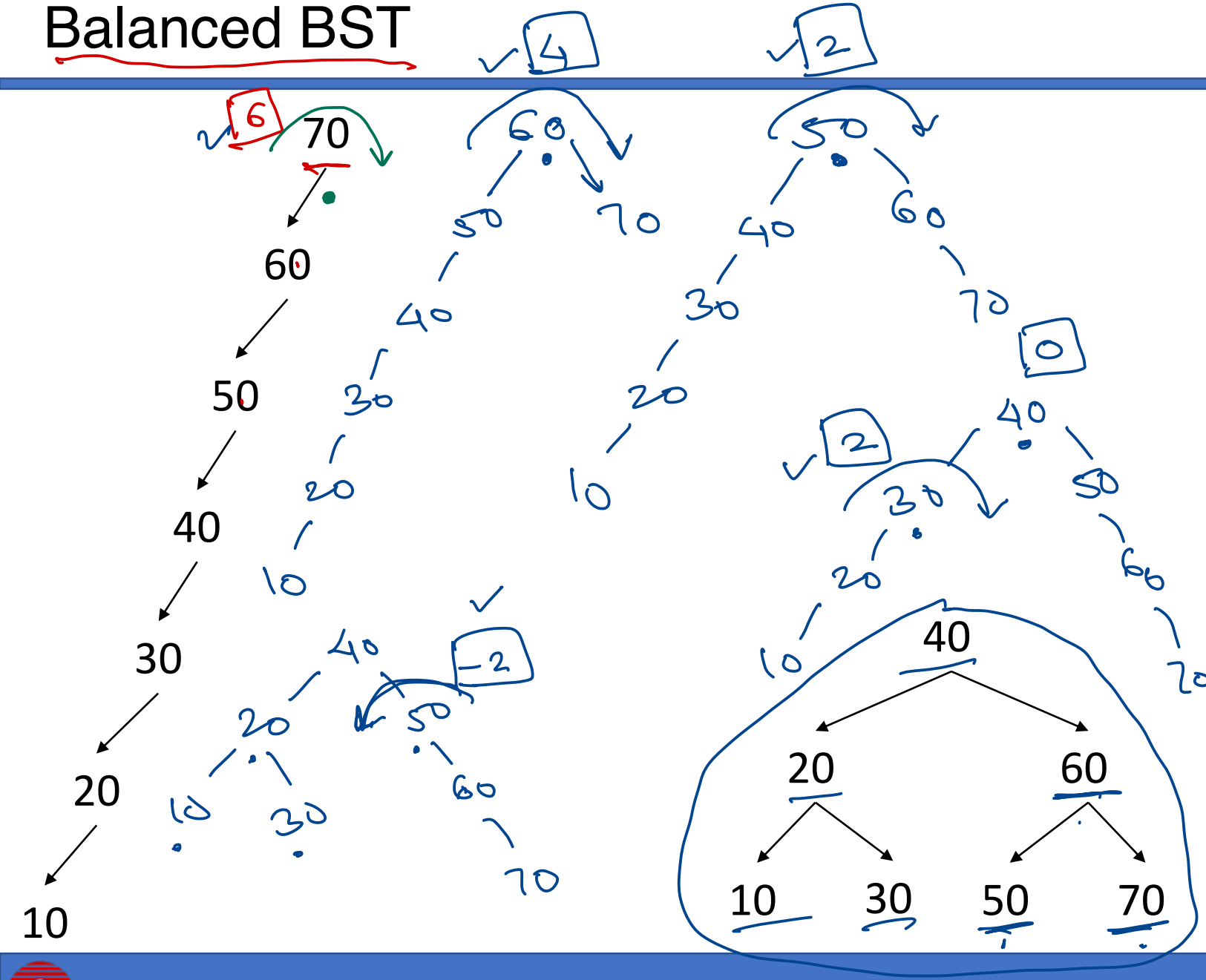

```
parent = temp;
pred = temp->left;

while (pred->right != null) {
    parent = pred;
    pred = pred->right;
}

temp->data = pred->data;
temp = pred;
```

# Skewed Binary Tree




- In Binary tree if only left or only right links are used, tree grows only on one side. Such tree is called as skewed binary tree.
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h).
- Such tree have maximum height i.e. same as number of elements.
- Time complexity of searching in skewed BST is O(n).

# Balanced BST




- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node is -1, 0 or +1.
- A tree can be balanced by applying series of left or right rotations on unbalanced nodes.


# Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>