Chapter 7 - Tree

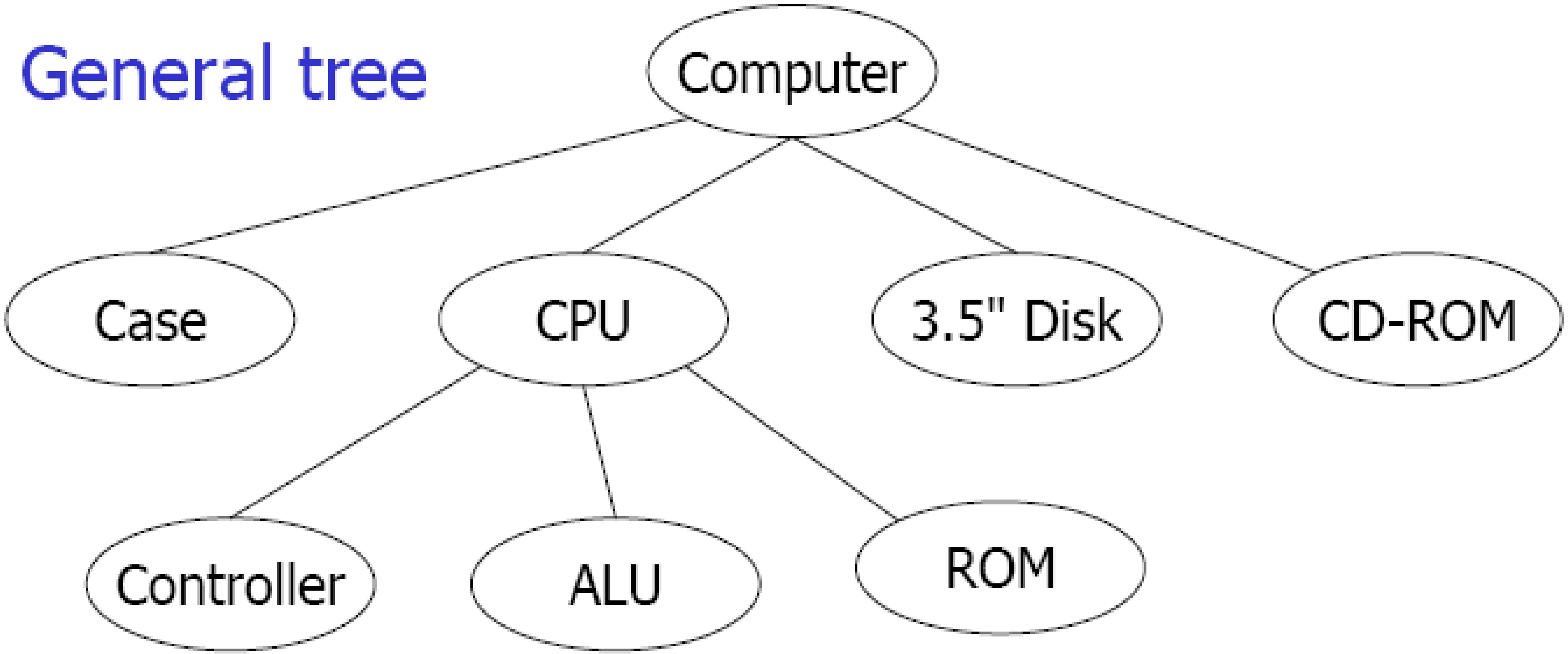
- Basic tree concepts
- Binary trees
- Binary Search Tree (BST)

Basic Tree Concepts

- A tree consists of:
  - nodes: finite set of elements
  - branches: directed lines connecting the nodes
- For a node:
  - degree: number of branches associated with the node
  - indegree: number of branches towards the node
  - outdegree: number of branches away from the node
- For a tree:
  - root: node with indegree 0
  - nodes different from the root must have indegree 1

Tree Representation

General tree



Terminology

- Leaf: node with outdegree 0
- Internal node: not a root or a leaf
- Parent: node with outdegree greater than 0
- Child: node with indegree greater than 0
- Siblings: nodes with the same parent
- Path: sequence of adjacent nodes

Terminology

- Ancestor: node in the path from the root to the node
- Descendent: node in a path from the node to a leaf
- Level: the node's distance from the root (at level 0)
- Height (Depth): the level of the leaf in the longest path from the root plus 1
- Sub-tree: connected structure below the root

Tree Representation

Indented list

Computer

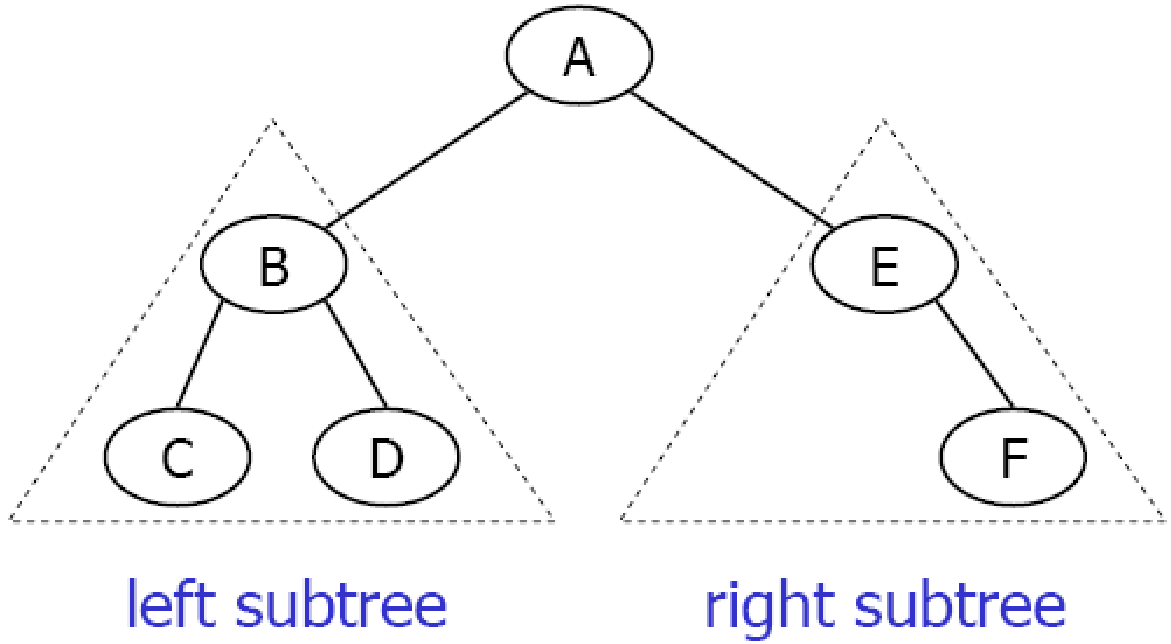
- Case
- CPU
  - Controller
  - ALU
  - ROM
  - ...
- 3.5" Disk
- CD-ROM

Parenthetical listing

Computer (Case CPU (Controller ALU ROM ...) 3.5" Disk CD-ROM)

Binary Trees

- A node cannot have more than two sub-trees:



Binary Tree Properties

- Height of binary trees:

$$H_{max} = N$$

$$H_{min} = \lfloor \log_2 N \rfloor + 1$$

$$N_{min} = H$$

$$N_{max} = 2^H - 1$$

Binary Tree Properties

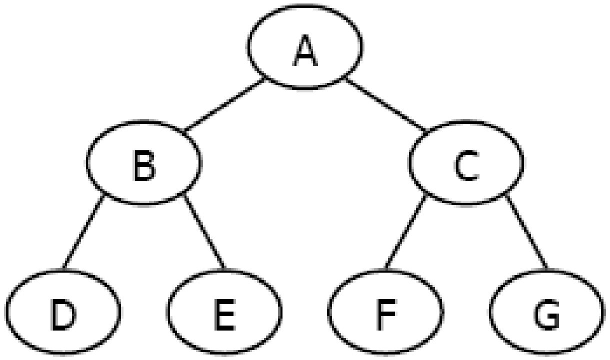
- Balance:
  - Balance factor: $B = H_L - H_R$
  - Balanced tree: balance factor is 0, -1, or 1
    sub-trees are balanced

Binary Tree Properties

- Completeness:
  - Complete tree:

    $N = N_{max} = 2^H - 1$

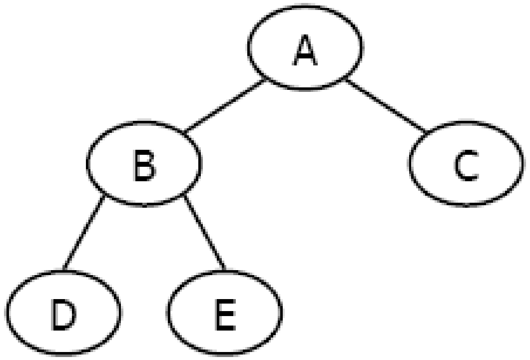
    (last level is full)



  - Nearly complete tree:

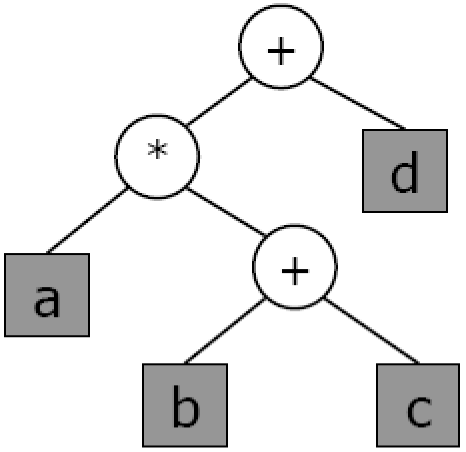
    $H = H_{min} = \lfloor \log_2 N \rfloor + 1$

    nodes in the last level are on the left



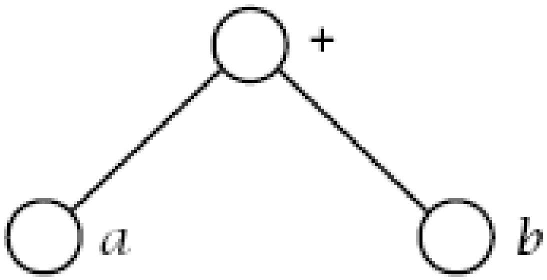
Expression Trees

- Each leaf is an operand
- The root and internal nodes are operators
- Sub-trees are sub-expressions

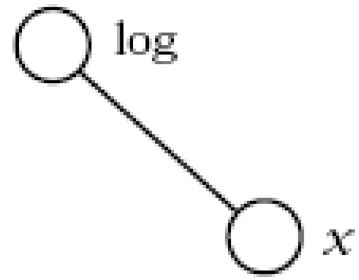


a * (b + c) + d

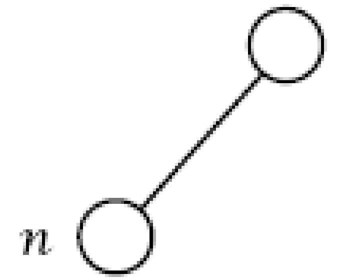
Expression Trees



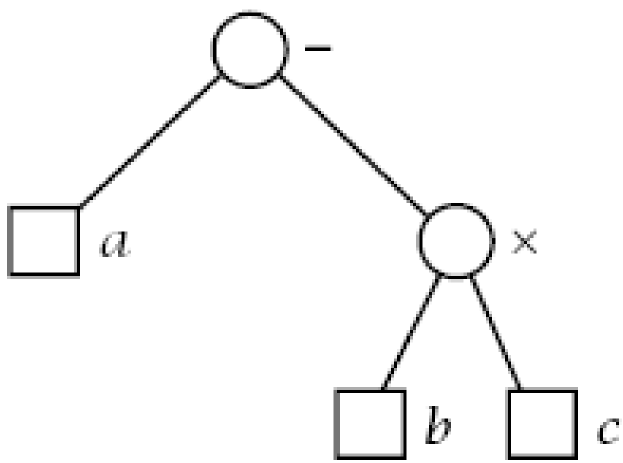
$a + b$



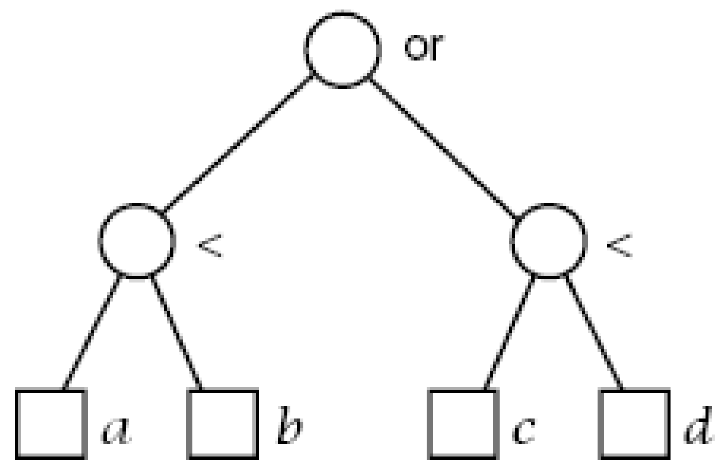
$\log x$



$n!$



$a - (b \times c)$



$(a < b)$ or $(c < d)$

Binary Tree ADT

DEFINITION: A binary tree ADT is either empty, or it consists of a node called root together with two binary trees called the left and the right subtree of the root.

Basic operations:

- *Construct* a tree, leaving it empty.
- *Insert* an element.
- *Remove* an element.
- *Search* an element.
- *Retrieve* an element.
- *Traverse* the tree, performing a given operation on each element.

Binary Tree ADT

Extended operations:

- Determine whether the tree is *empty* or not.
- Find the *size* of the tree.
- *Clear* the tree to make it empty.

Specifications for Binary Tree

<void> Create()

<boolean> isFull()

<boolean> isEmpty()

<integer> Size()

<void> Clear()

<ErrorCode> Search (ref DataOut <DataType>)

<ErrorCode> Insert (val DataIn <DataType>)

<ErrorCode> Remove (val key <KeyType>)

<ErrorCode> Retrieve (ref DataOut <DataType>)

Depend on various types of binary trees
(BST, AVL, 2d-tree)

Specifications for Binary Tree

- ***Binary Tree Traversal***: Each node is processed once and only once in a predetermined sequence.

- ***Depth-First Traverse***:

<void> preOrderTraverse (ref<void>Operation(ref Data <DataType>))

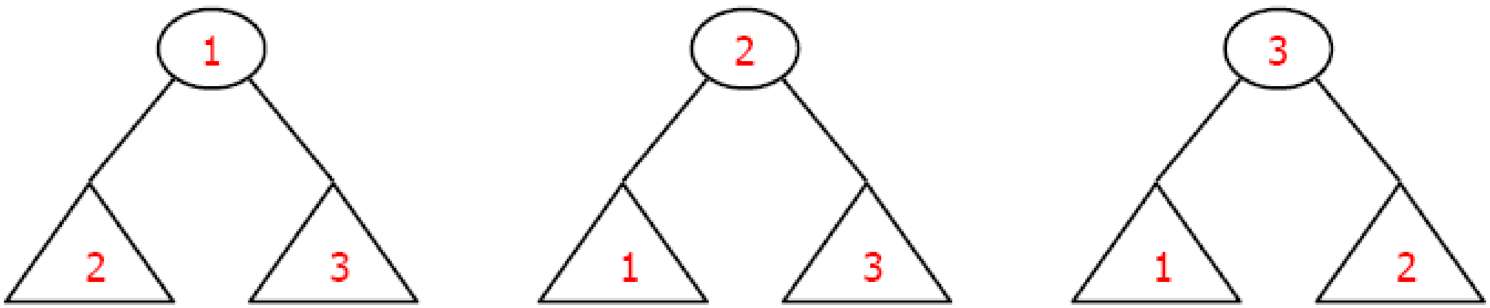
<void> inOrderTraverse (ref<void>Operation(ref Data <DataType>))

<void> postOrderTraverse (ref<void>Operation(ref Data <DataType>))

- ***Breadth-First Traverse***:

<void> BreadthFirstTraverse (ref<void>Operation(ref Data <DataType>))

Depth-First Traversal

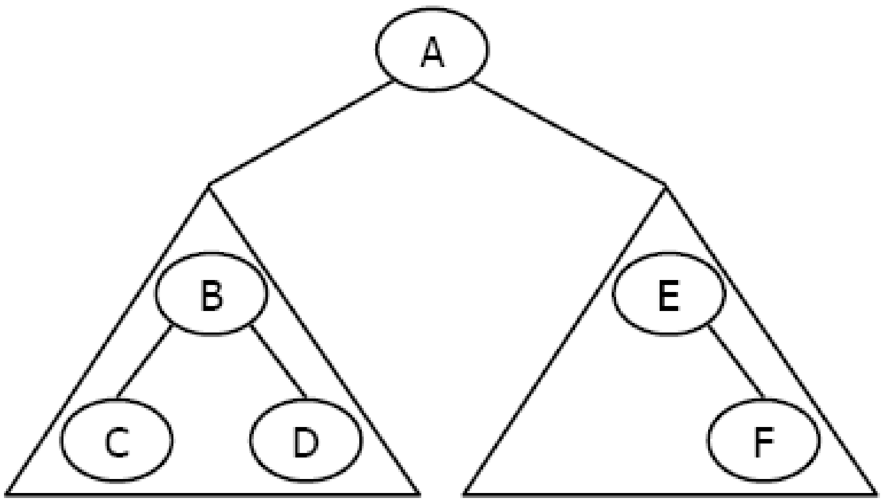


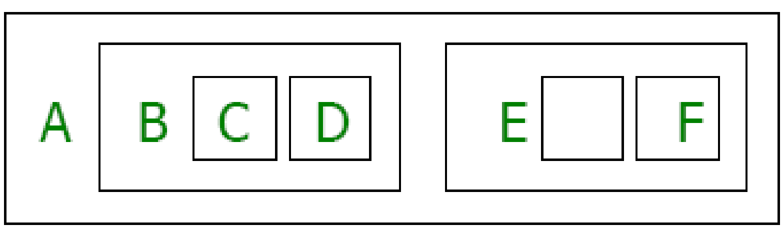
PreOrder
NLR

InOrder
LNR

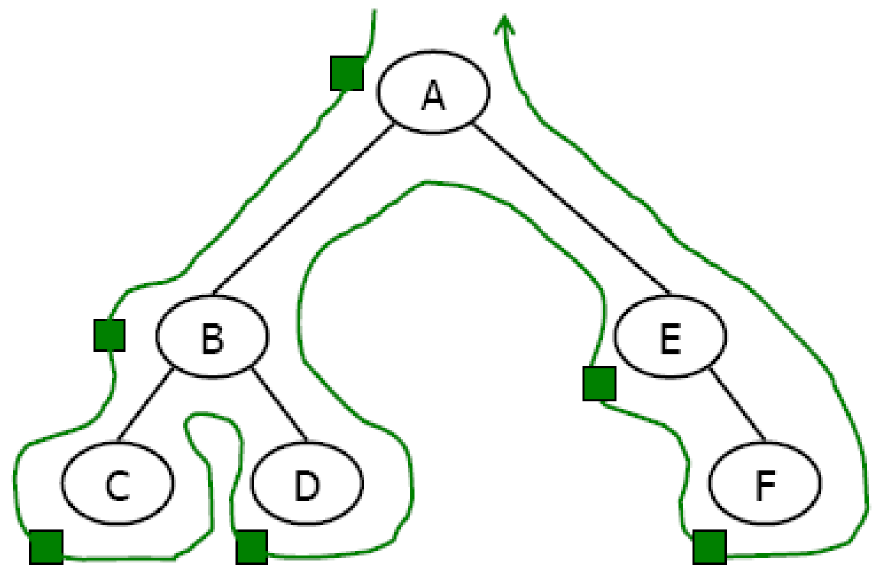
PostOrder
LRN

PreOrder Traversal



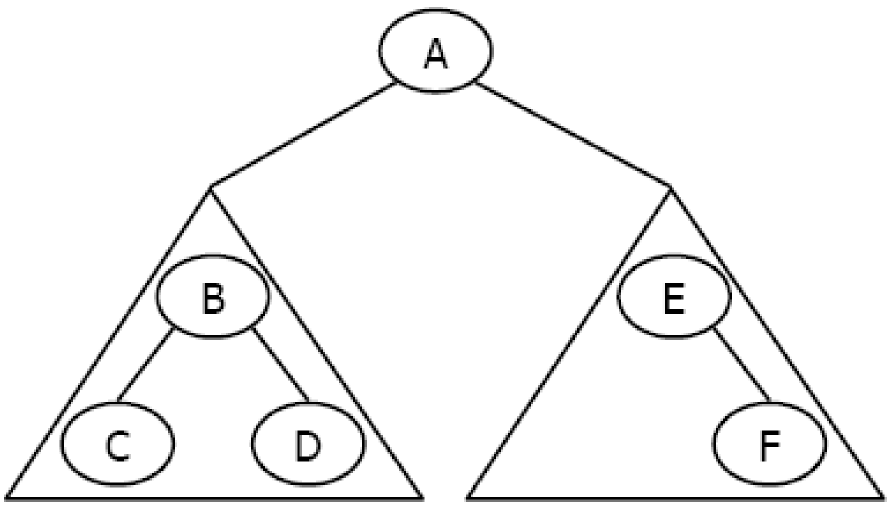


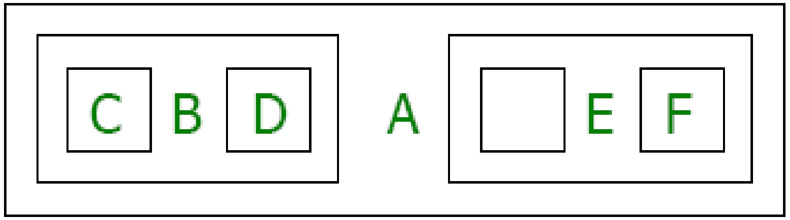
Processing order



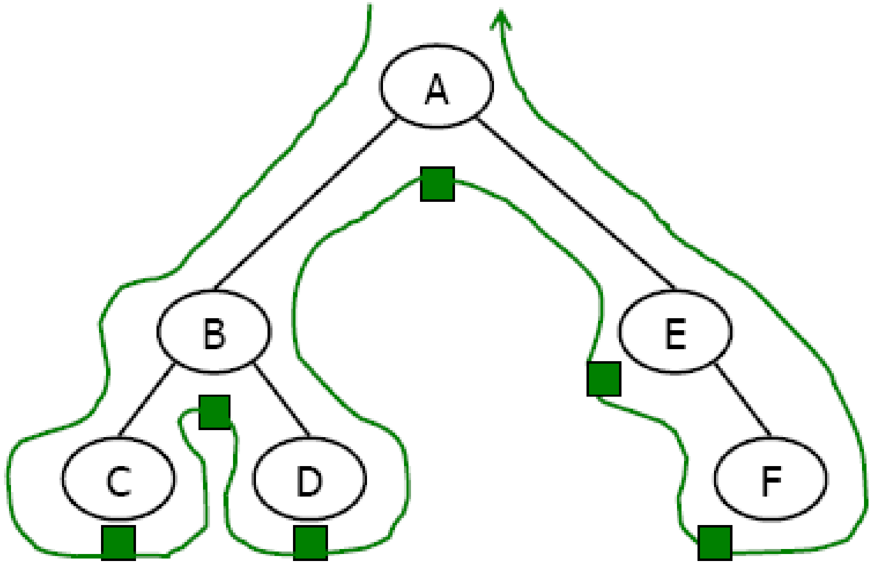
Walking order

InOrder Traversal



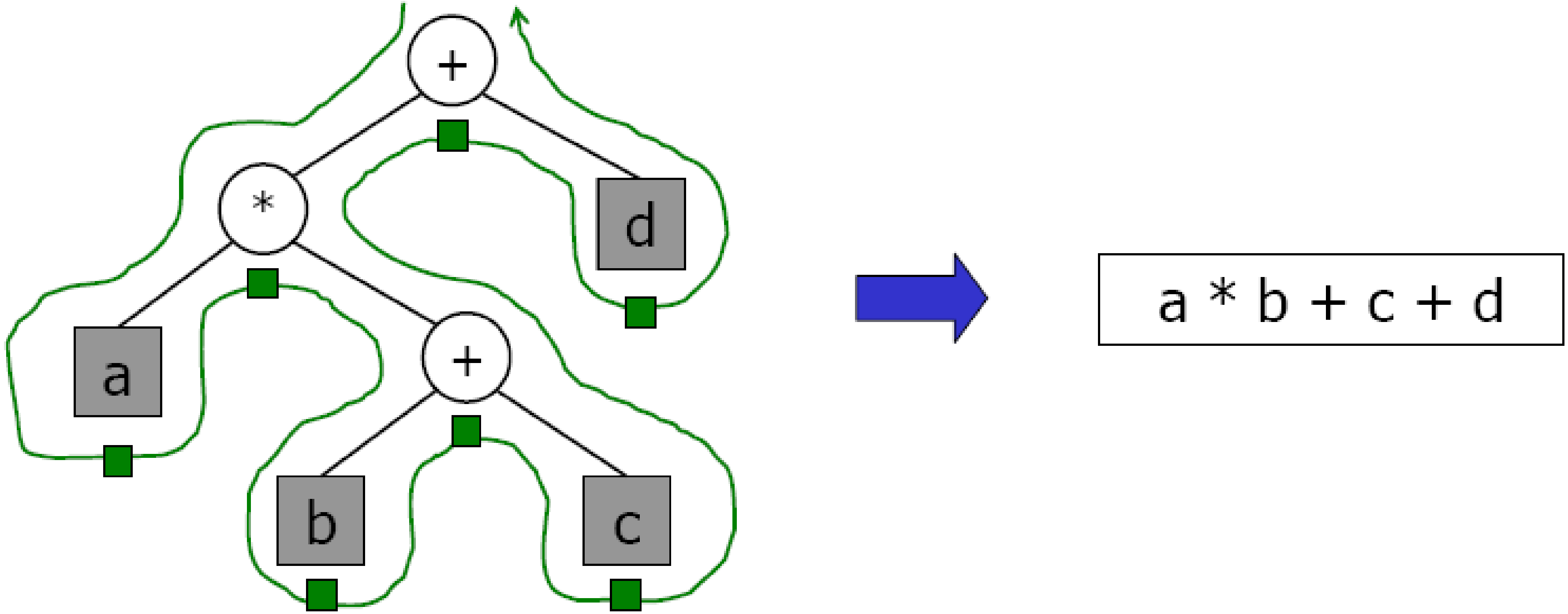


Processing order

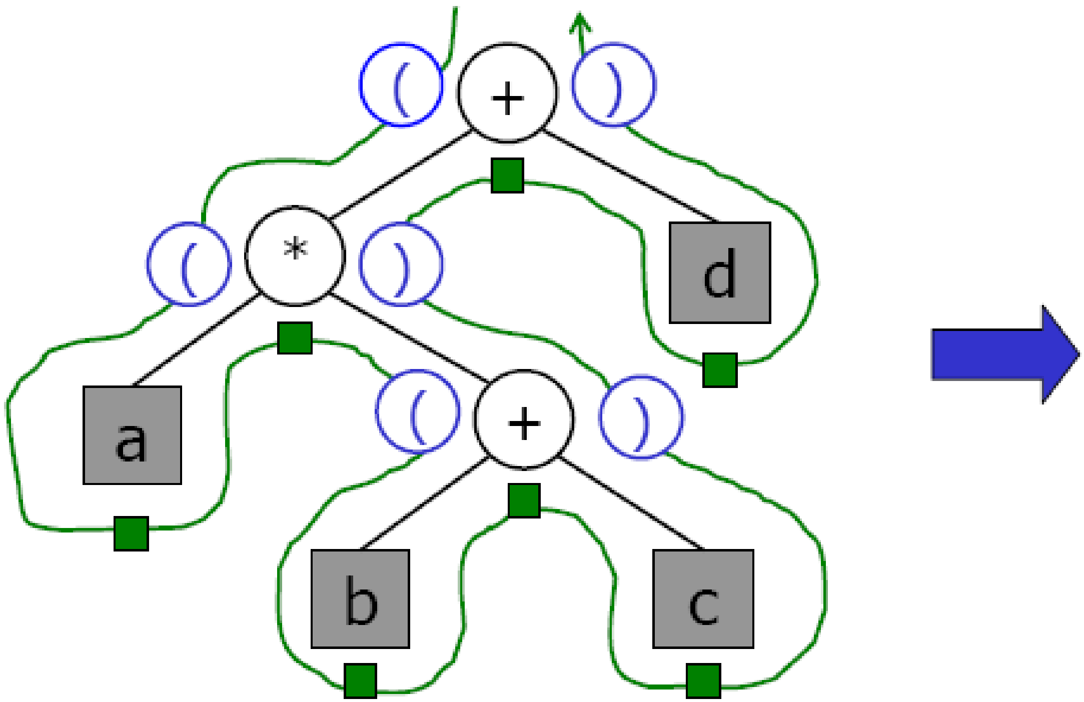


Walking order

InOrder Traversal

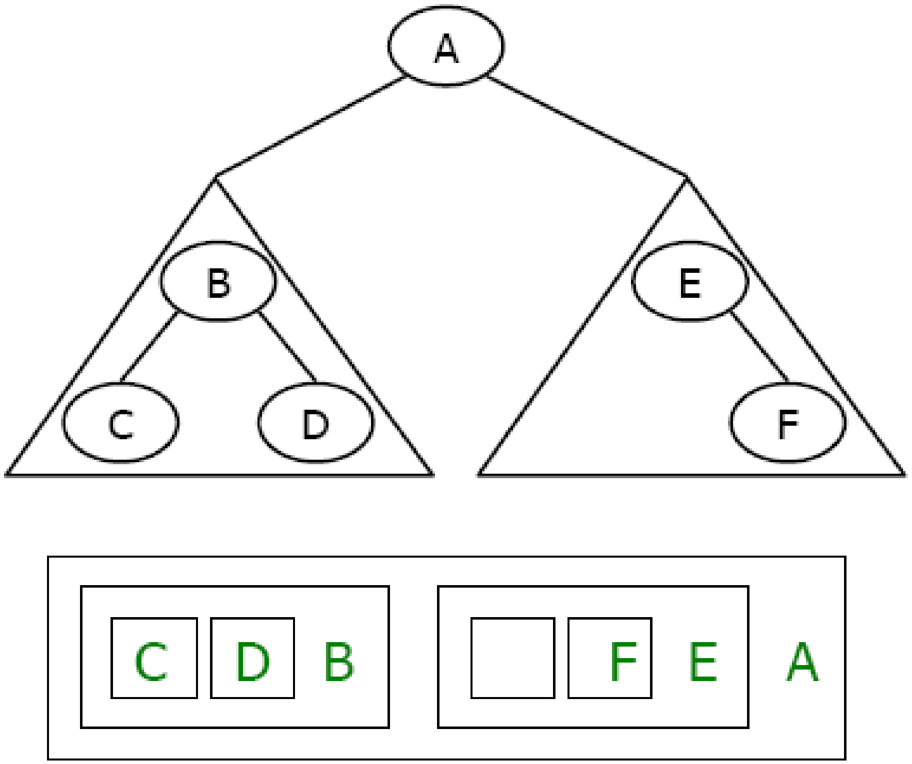


InOrder Traversal

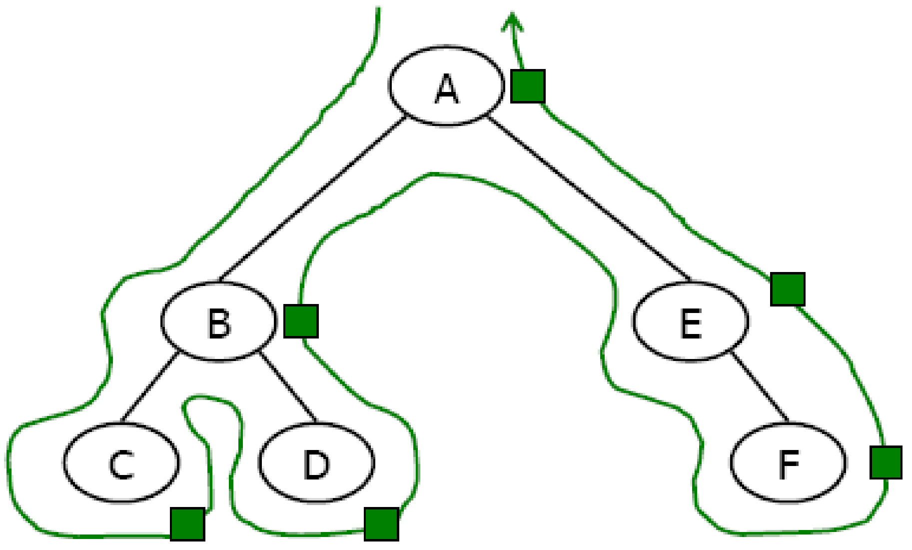


((a * (b + c)) + d)

PostOrder Traversal

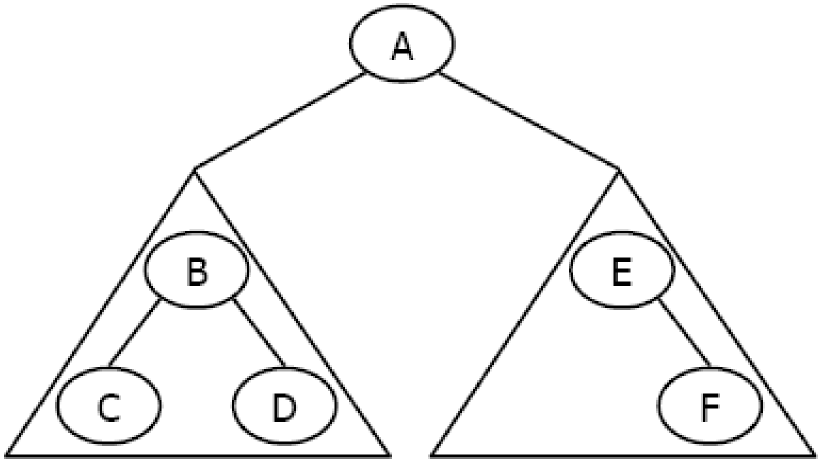


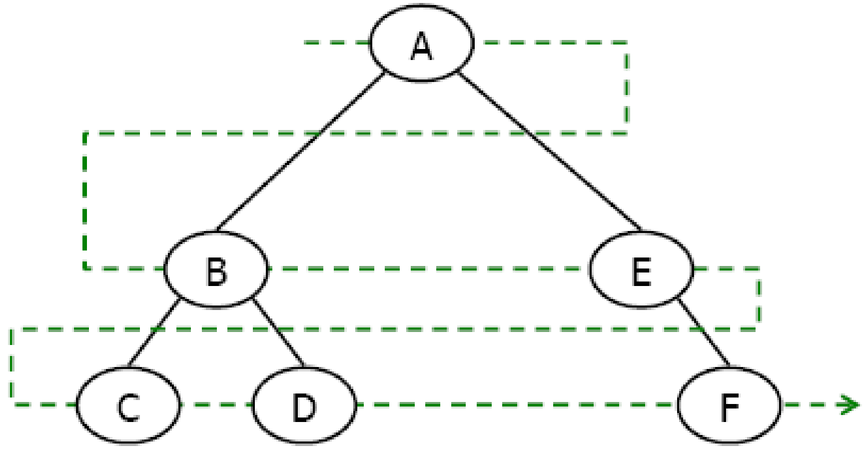
Processing order

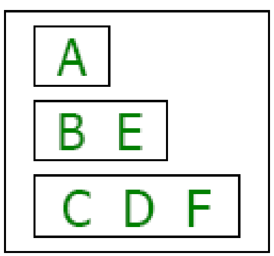


Walking order

Breadth-First Traversal







Processing order

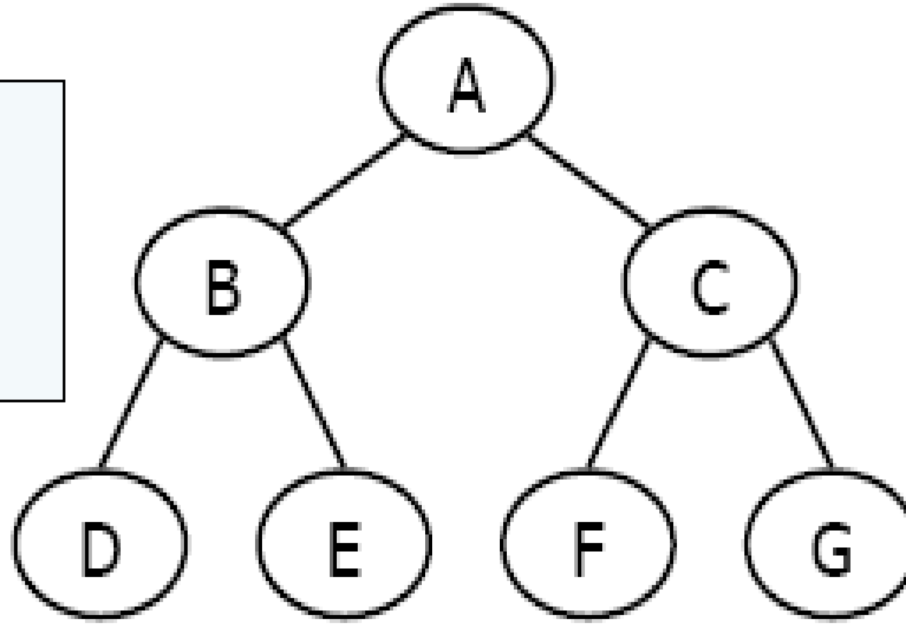
Walking order

Contiguous Implementation of Binary Tree

```
BinaryTree
        Data <Array of <DataType> >
End BinaryTree
```

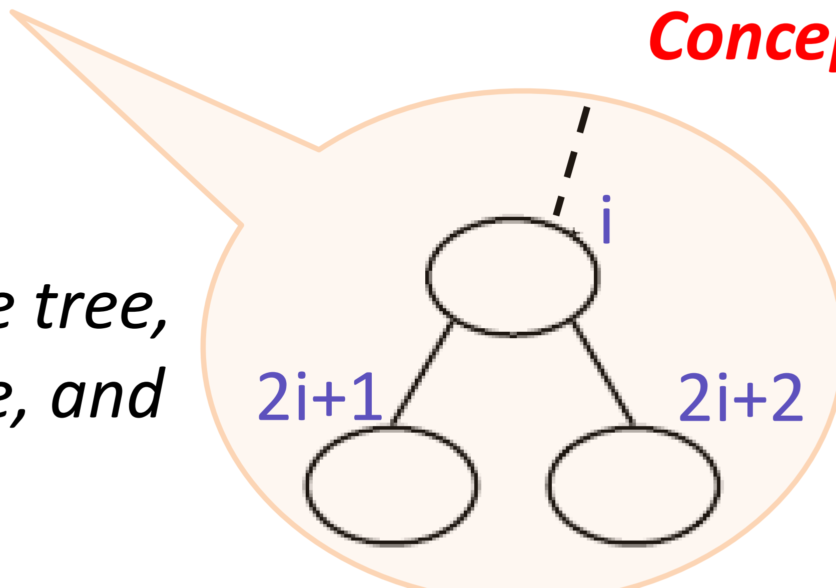
0	1	2	3	4	5	6	
A	B	C	D	E	F	G	…

Physical



Conceptual

(suitable for complete tree, nearly complete tree, and bushy tree)



Contiguous Implementation of Binary Tree

Record

Data <DataType>

Parent <DataType>

Flag <ChildType>

End Record

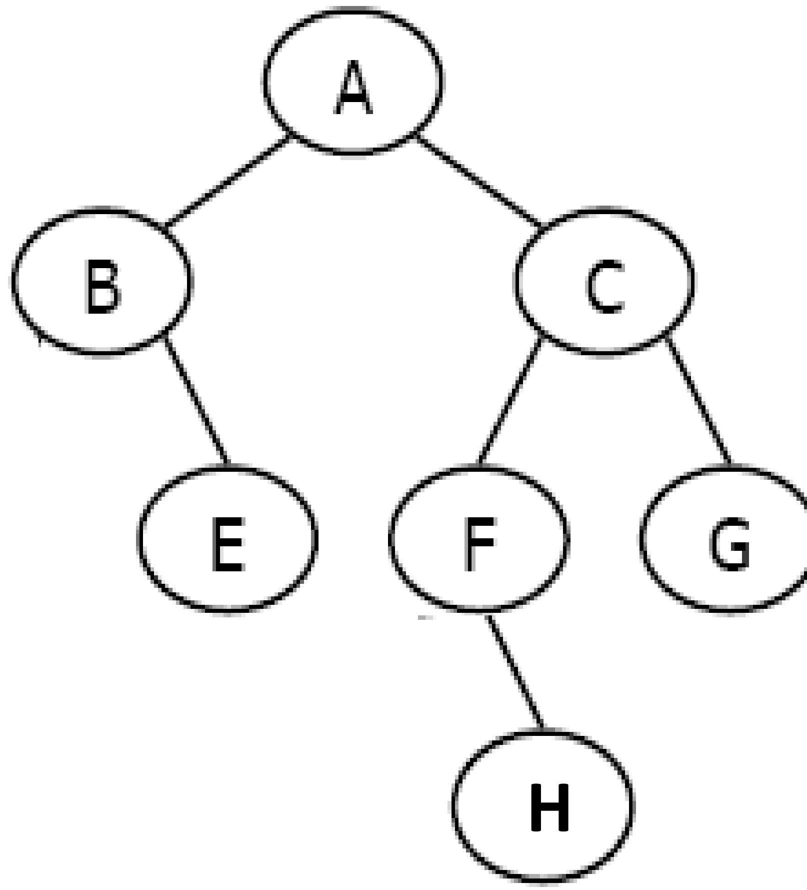
BinaryTree

Data <Array of <Record> >

End BinaryTree

	0	**1**	**2**	**3**	**4**	**5**	**6**	
Data	A	B	E	C	F	G	H	…
Parent	-	A	B	A	C	C	F	…
Flag	-	L	R	R	L	R	R	…

Physical *(suitable for sparse tree)*



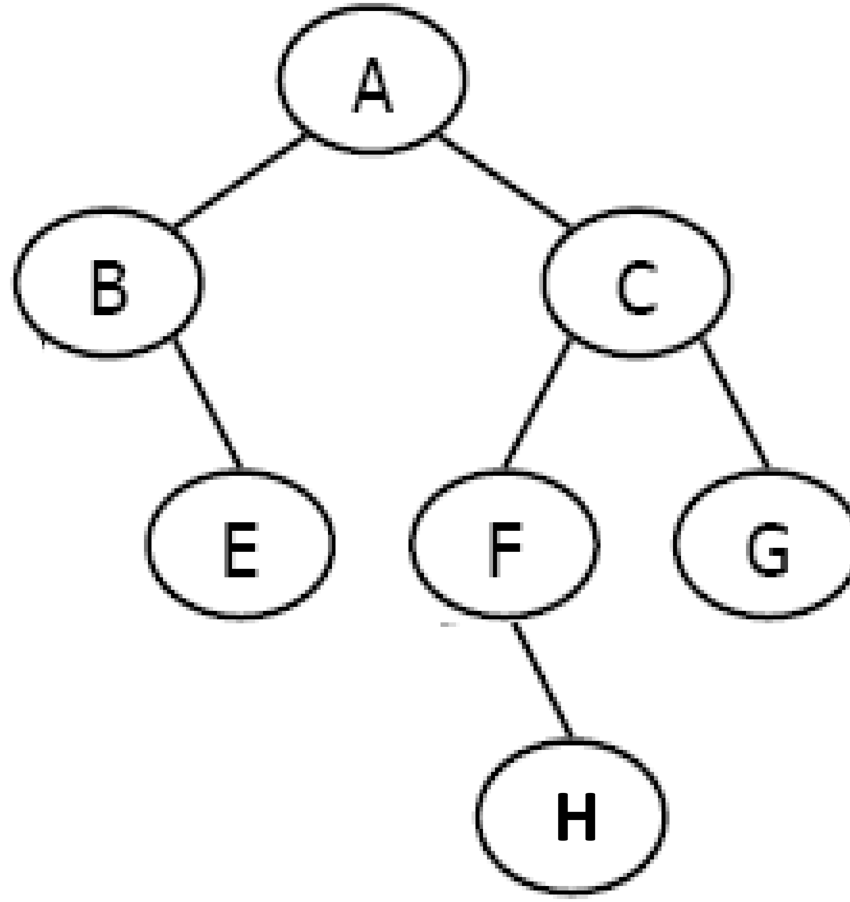
Conceptual

Contiguous Implementation of Binary Tree

BinaryTree
NLR <Array of <DataType> >
LNR <Array of <DataType> >
End BinaryTree

	0	1	2	3	4	5	6
NLR	A	B	E	C	F	H	G
LNR	B	E	A	F	H	C	G

Physical



Conceptual

(A binary tree without identical data can be restored from two array of LNR and NLR traverse)

Linked Implementation of Binary Tree

BinaryNode

data <DataType>

left <pointer>

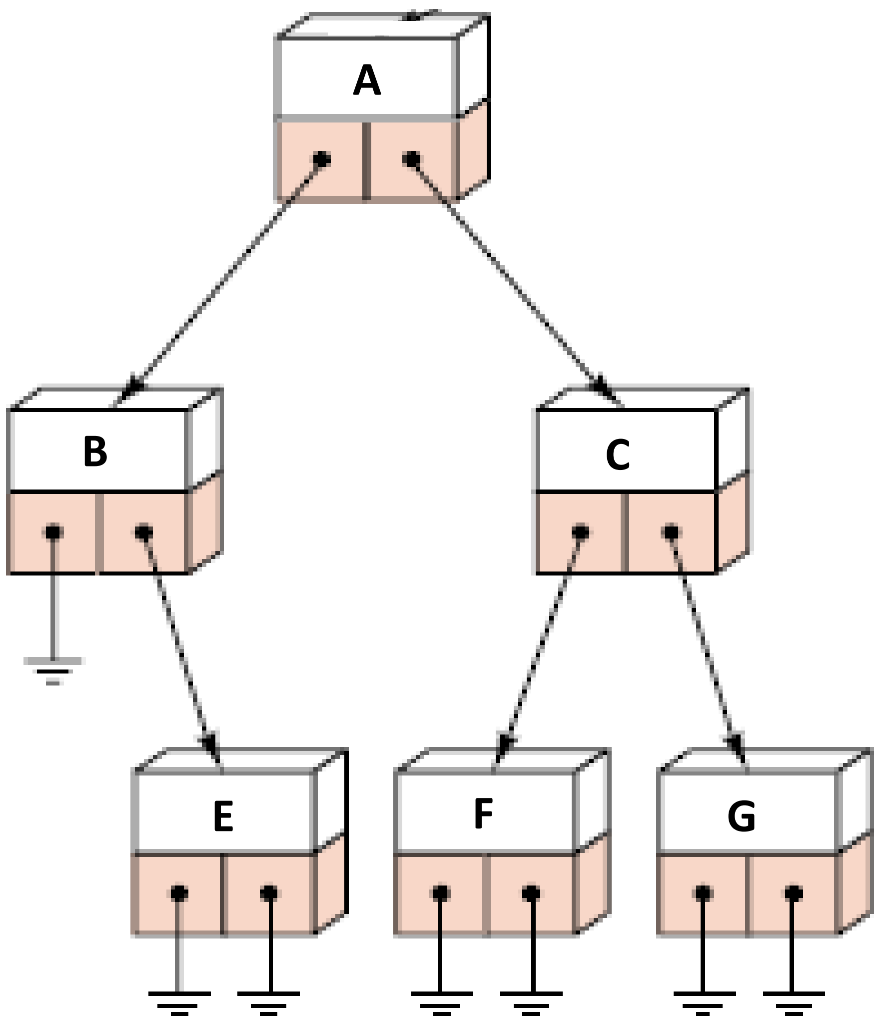
right <pointer>

End BinaryNode

BinaryTree

root <pointer>

End BinaryTree



Physical

Depth-First Traversal

Auxiliary functions for Depth_First Traversal:

recursive_preOrder

recursive_inOrder

recursive_postOrder

PreOrder Traversal

Algorithm **recursive_preOrder** (val subroot <pointer>,
ref<void>Operation(ref Data <DataType>))

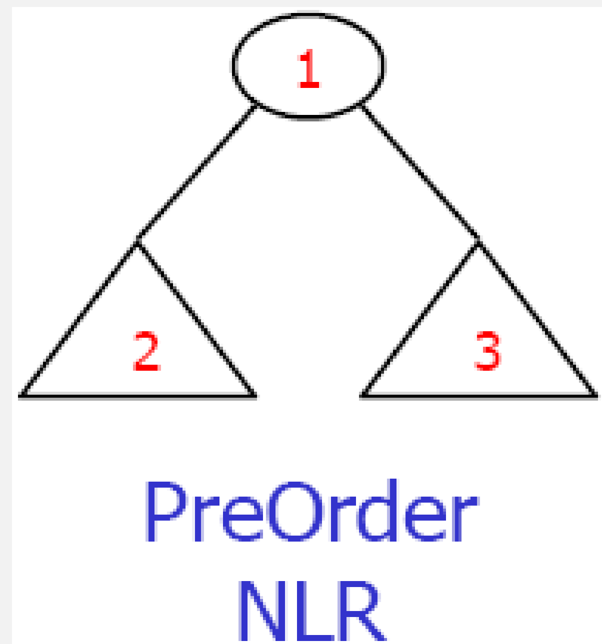
Traverses a binary tree in *node-left-right* sequence.

Pre subroot points to the root of a tree/ subtree.

Post each node has been processed in order.

1. **if** (subroot is not NULL)
    1. Operation(subroot->data)
    2. recursive_preOrder(subroot->left)
    3. recursive_preOrder(subroot->right)

End recursive_preOrder



InOrder Traversal

Algorithm **recursive_inOrder** (val subroot <pointer>,

ref<void>Operation(ref Data <DataType>))

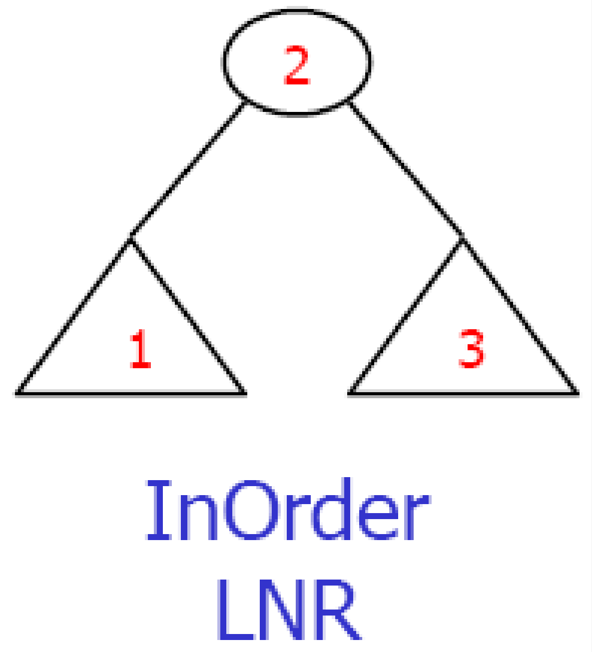
Traverses a binary tree in *left-node-right* sequence

Pre subroot points to the root of a tree/ subtree

Post each node has been processed in order

1. **if** (subroot is not NULL)
    1. recursive_inOrder(subroot->left)
    2. Operation(subroot->data)
    3. recursive_inOrder(subroot->right)

End recursive_inOrder



PostOrder Traversal

Algorithm **recursive_postOrder** (val subroot <pointer>,
ref<void>Operation(ref Data <DataType>))

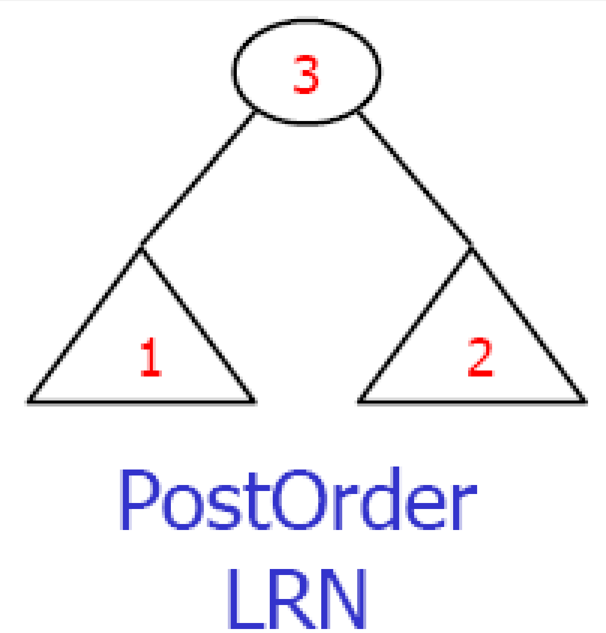
Traverses a binary tree in *left-right-node* sequence

Pre subroot points to the root of a tree/ subtree

Post each node has been processed in order

1. **if** (subroot is not NULL)
    1. recursive_postOrder(subroot->left)
    2. recursive_postOrder(subroot->right)
    3. Operation(subroot->data)

End recursive_postOrder



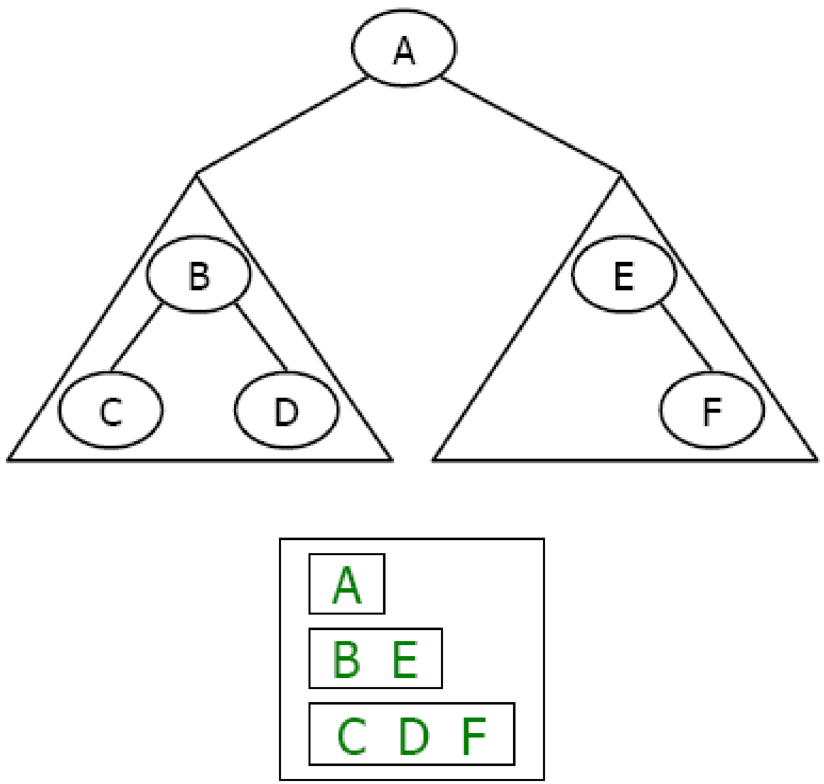
Depth-First Traversal

```
<void> preOrderTraverse (ref<void>Operation(ref Data <DataType>))
1.        recursive_preOrder(root, Operation)
End preOrderTraverse
```

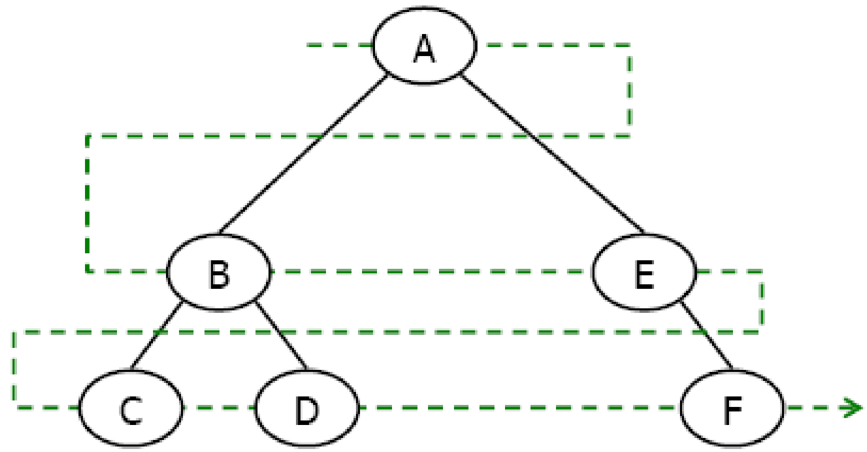
```
<void> inOrderTraverse (ref<void>Operation(ref Data <DataType>))
1.        recursive_inOrder(root, Operation)
End inOrderTraverse
```

```
<void> postOrderTraverse (ref<void>Operation(ref Data <DataType>))
1.        recursive_postOrder(root, Operation)
End postOrderTraverse
```

Breadth-First Traversal



Processing order



Walking order

Breadth-First Traversal

Algorithm **BreadthFirstTraverse**

(ref<void>Operation(ref Data <DataType>))

Traverses a binary tree in sequence from lowest level to highest level, in each level traverses from left to right.

Post each node has been processed in order

Uses Queue ADT

Breadth-First Traversal

Algorithm BreadthFirstTraverse

(ref<void>Operation(ref Data <DataType>))

1. **if** (root is not NULL)
    1. queueObj <Queue>
    2. queueObj.EnQueue(root)
    3. **loop** (not queueObj.isEmpty())
        1. queueObj.QueueFront(pNode)
        2. queueObj.DeQueue()
        3. Operation(pNode->data)
        4. **if** (pNode->left is not NULL)
            1. queueObj.EnQueue(pNode->left)
        5. **if** (pNode->right is not NULL)
            1. queueObj.EnQueue(pNode->right)

End BreadthFirstTraverse

Binary Search Tree (BST)

- All items in the left subtree < the root.
- All items in the right subtree > the root.
- Each subtree is itself a binary search tree.

Binary Search Tree (BST)

- BST is one of implementations for ordered list.

- In BST we can search quickly (as with binary search on a *contiguous list*).

- In BST we can make insertions and deletions quickly (as with a *linked list*).

- When a BST is traversed in *inorder*, the keys will come out in sorted order.

37

Binary Search Tree (BST)

Auxiliary functions for Search:

recursive_Search

iterative_Search

Search node in BST (recursive version)

<pointer> **recursive_Search** (val subroot <pointer>,
val target <KeyType>)

Searches target in the subtree.

Pre subroot points to the root of a tree/ subtree.

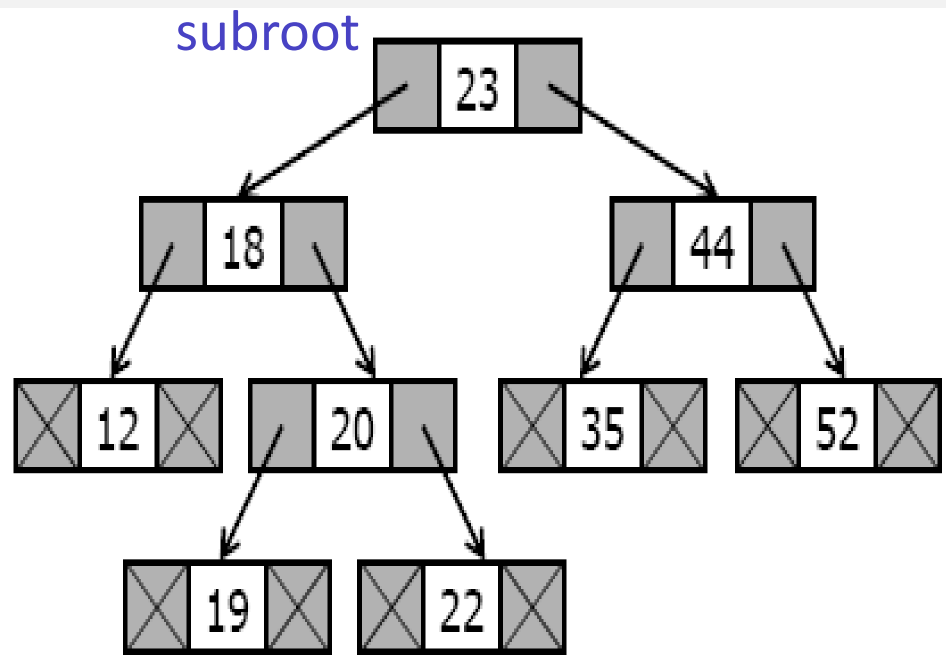
Post If target is not in the subtree, NULL is returned. Otherwise, a pointer to the node containing the target is returned.

Search node in BST (recursive version)

1. **if** (subroot is NULL) OR (subroot->data = target)
    1. return subroot
2. **else if** (target < subroot->data)
    1. return recursive_Search(subroot->left, target)
3. **else**
    1. return recursive_Search(subroot->right, target)

End recursive_Search

Target = 22

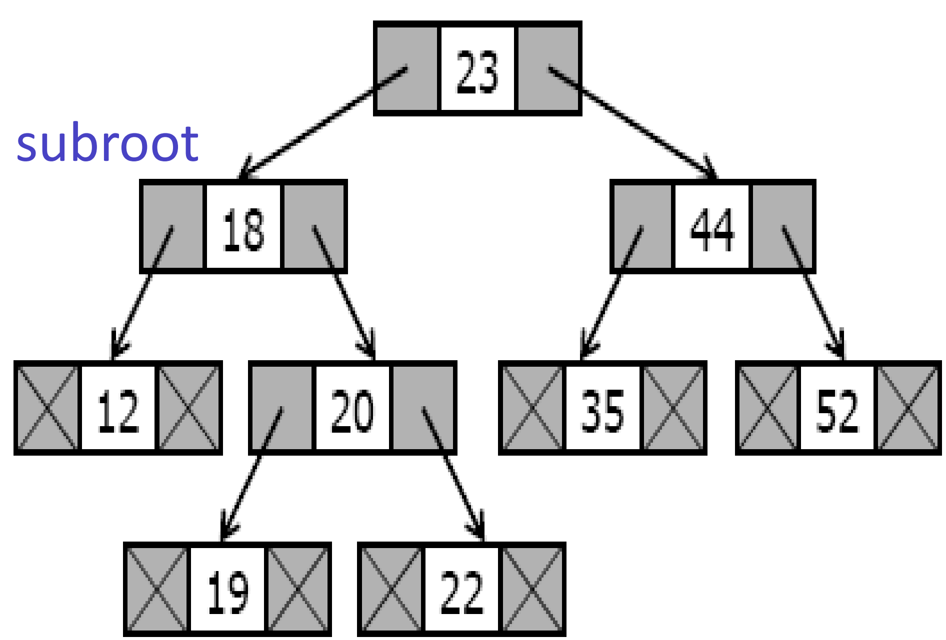


Search node in BST (recursive version)

1. **if** (subroot is NULL) OR (subroot->data = target)
    1. return subroot
2. **else if** (target < subroot->data)
    1. return recursive_Search(subroot->left, target)
3. **else**
    1. return recursive_Search(subroot->right, target)

End recursive_Search

Target = 22

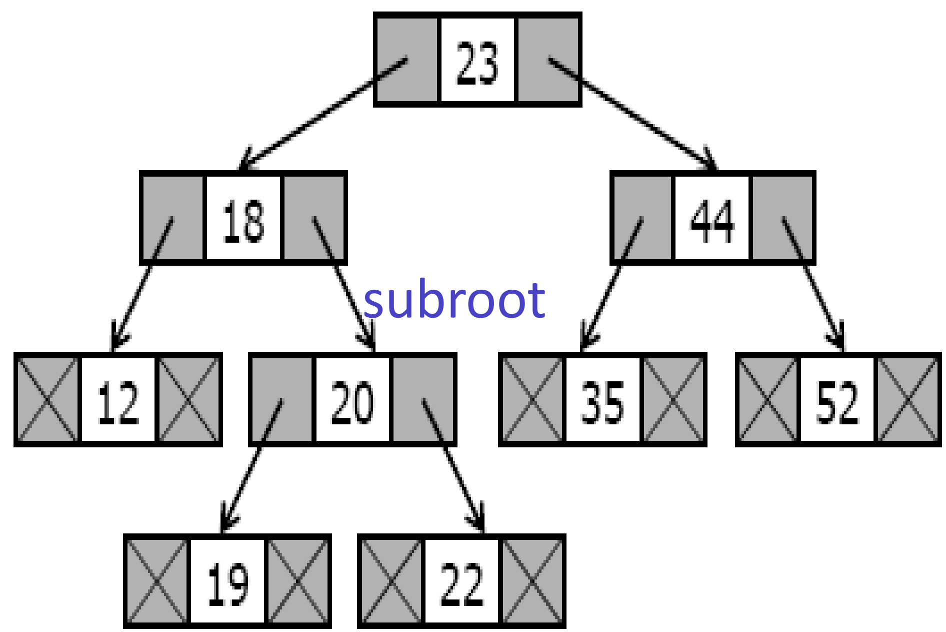


Search node in BST (recursive version)

1. **if** (subroot is NULL) OR (subroot->data = target)
   1. return subroot
2. **else if** (target < subroot->data)
   1. return recursive_Search(subroot->left, target)
3. **else**
   1. return recursive_Search(subroot->right, target)

End recursive_Search

Target = 22

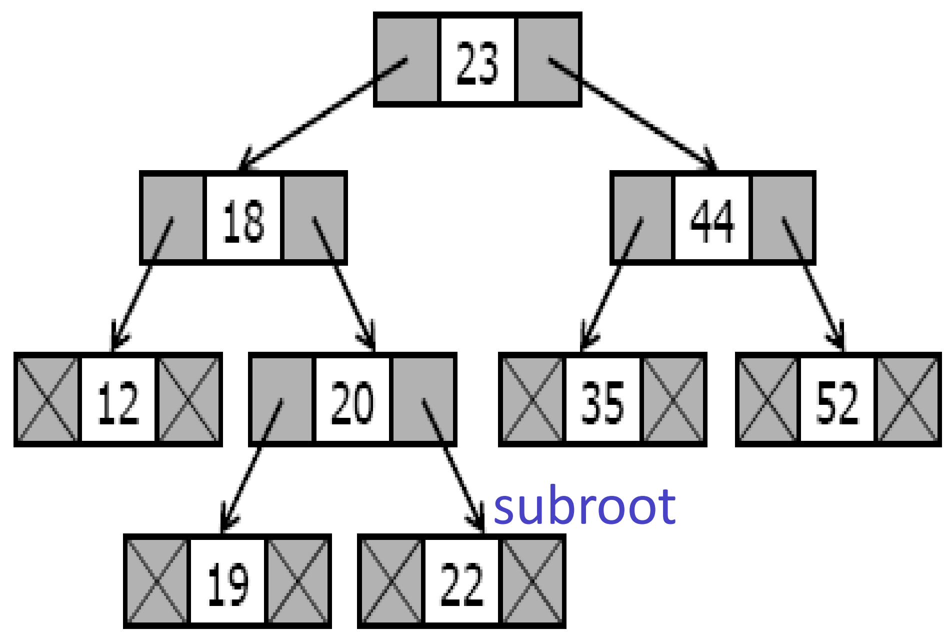


Search node in BST (recursive version)

1. **if** (subroot is NULL) OR (subroot->data = target)
    1. return subroot
2. **else if** (target < subroot->data)
    1. return recursive_Search(subroot->left, target)
3. **else**
    1. return recursive_Search(subroot->right, target)

End recursive_Search

Target = 22



Search Node in BST (nonrecursive version)

<pointer> **iterative_Search** (val subroot <pointer>,
val target <KeyType>)

Searches target in the subtree.

Pre subroot points to the root of a tree/ subtree.

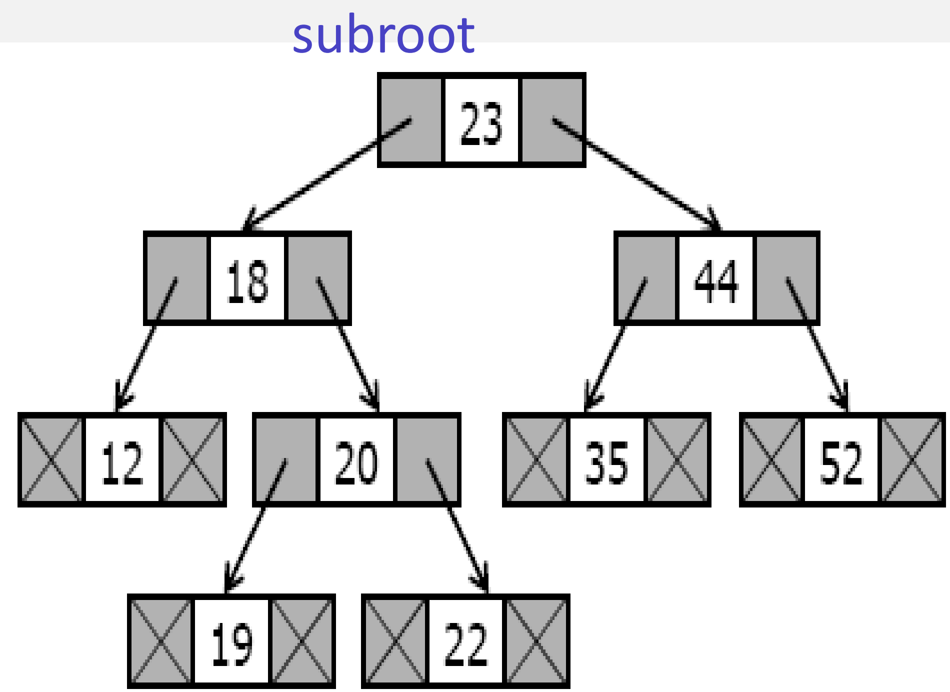
Post If target is not in the subtree, NULL is returned. Otherwise, a pointer to the node containing the target is returned.

Search Node in BST (nonrecursive version)

1. **while** (subroot is not NULL) AND (subroot->data.key <> target)
    1. **if** (target < subroot->data.key)
        1. subroot = subroot->left
    2. **else**
        1. subroot = subroot->right
2. return subroot

End iterative_Search

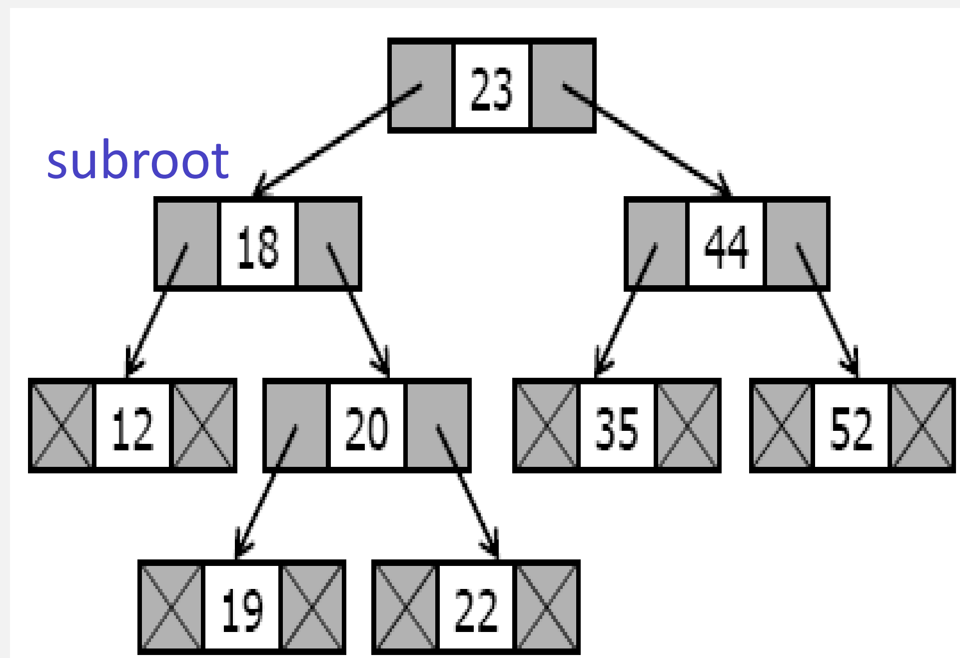
Target = 22



Search Node in BST (nonrecursive version)

1. **while** (subroot is not NULL) AND (subroot->data.key <> target)
    1. **if** (target < subroot->data.key)
        1. subroot = subroot->left
    2. **else**
        1. subroot = subroot->right
2. return subroot

End iterative_Search



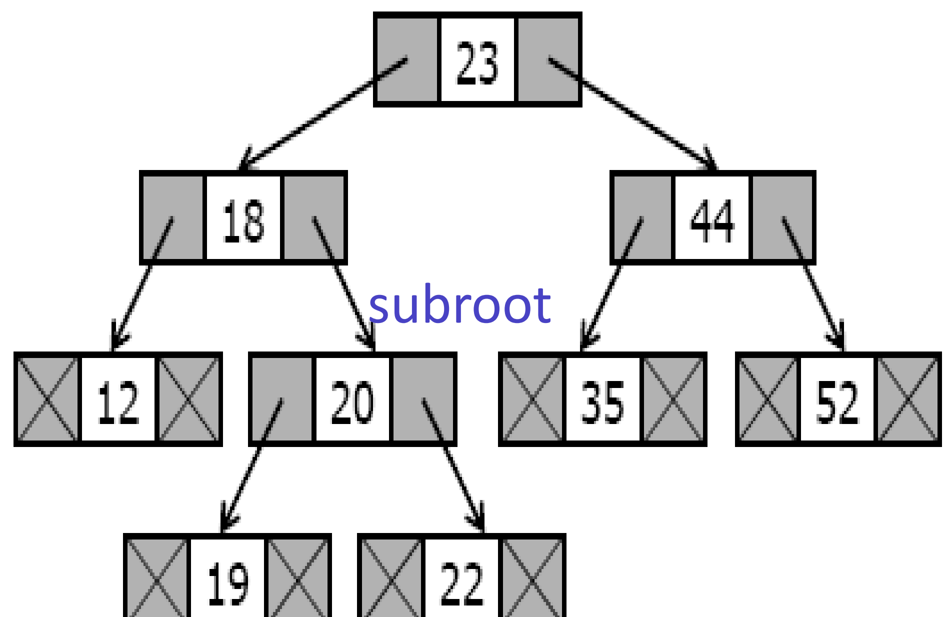
Target = 22

Search Node in BST (nonrecursive version)

```
1. while (subroot is not NULL) AND (subroot->data.key <> target)
   1. if (target < subroot->data.key)
      1. subroot = subroot->left
   2. else
      1. subroot = subroot->right
2. return subroot
```

End iterative_Search

Target = 22

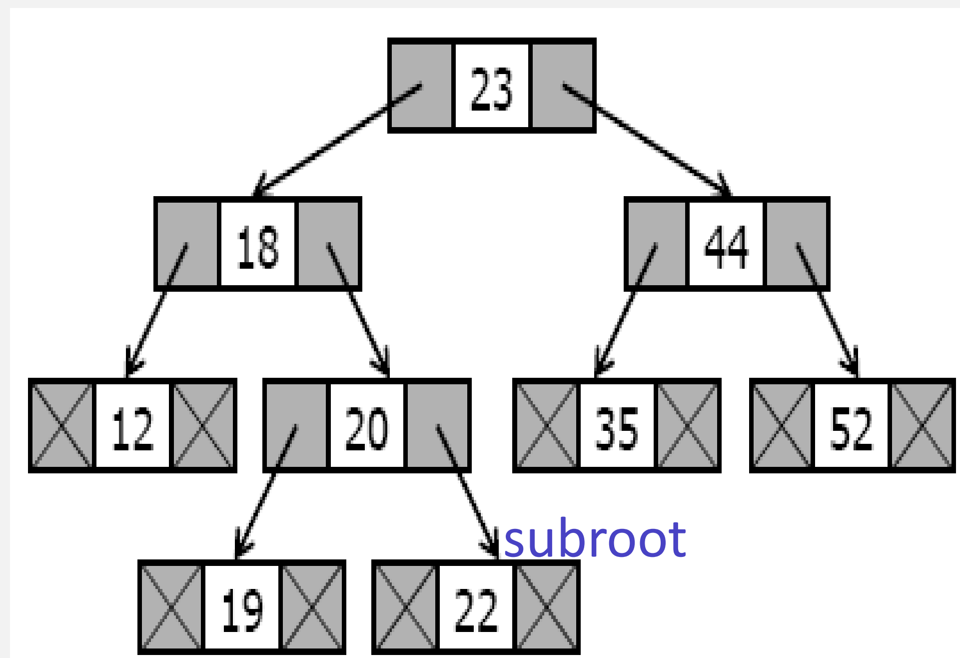


Search Node in BST (nonrecursive version)

1. **while** (subroot is not NULL) AND (subroot->data.key <> target)
   1. **if** (target < subroot->data.key)
      1. subroot = subroot->left
   2. **else**
      1. subroot = subroot->right
2. return subroot

End iterative_Search

Target = 22



Search node in BST

<ErrorCode> **Search** (ref DataOut <DataType>)

Searches target in the subtree.

Pre DataOut contains value needs to be found in key field.

Post DataOut will reveive all other values in other fields if that key is found.

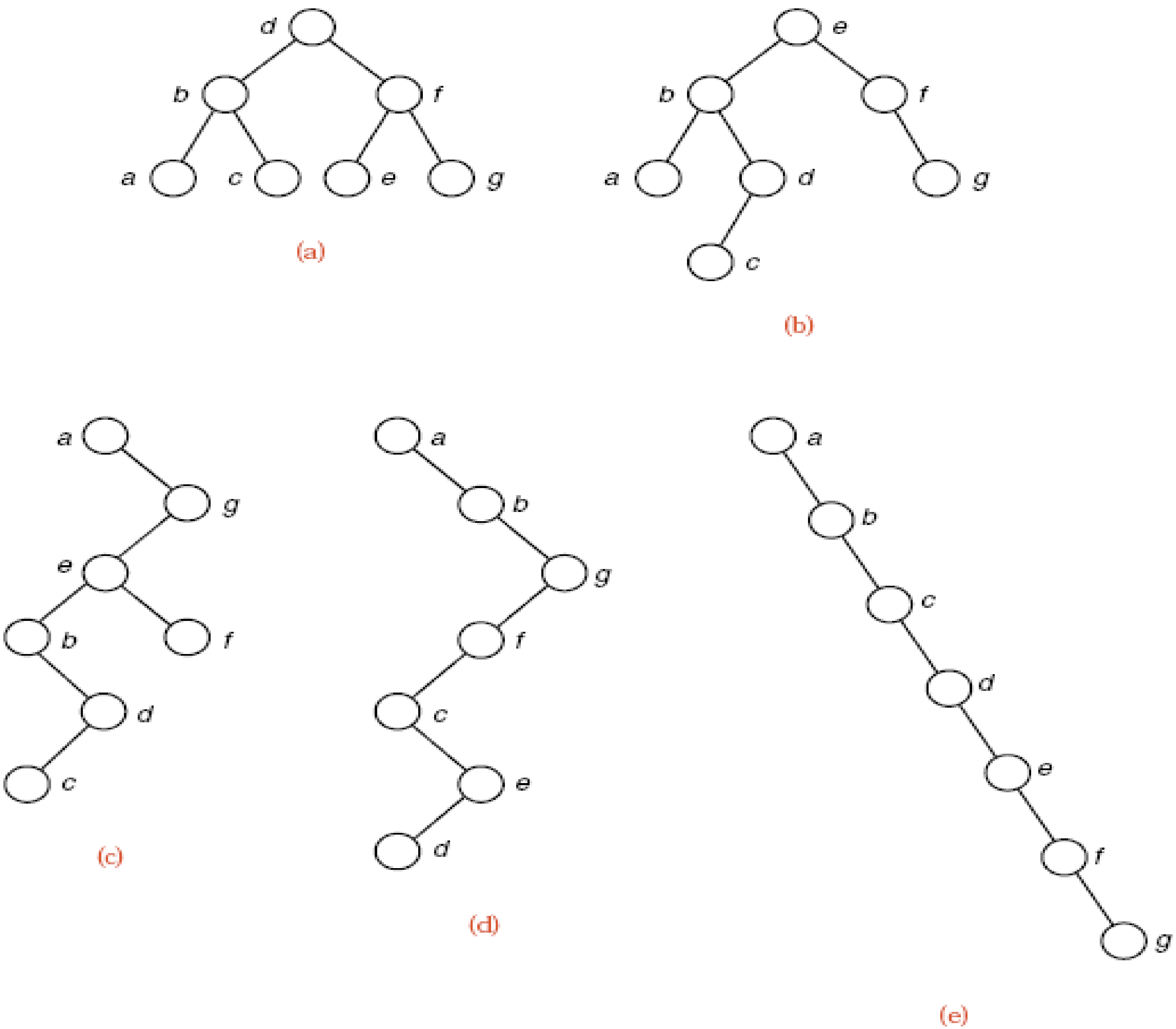
Return *success* or *notPresent*

Uses Auxiliary function recursive_Search or iterative_Search

1. pNode = recursive_Search(root, DataOut.key)
2. **if** (pNode is NULL)
    1. return *notPresent*
3. dataOut = pNode->data
4. return *success*

End Search

Binary Search Trees with the Same Keys

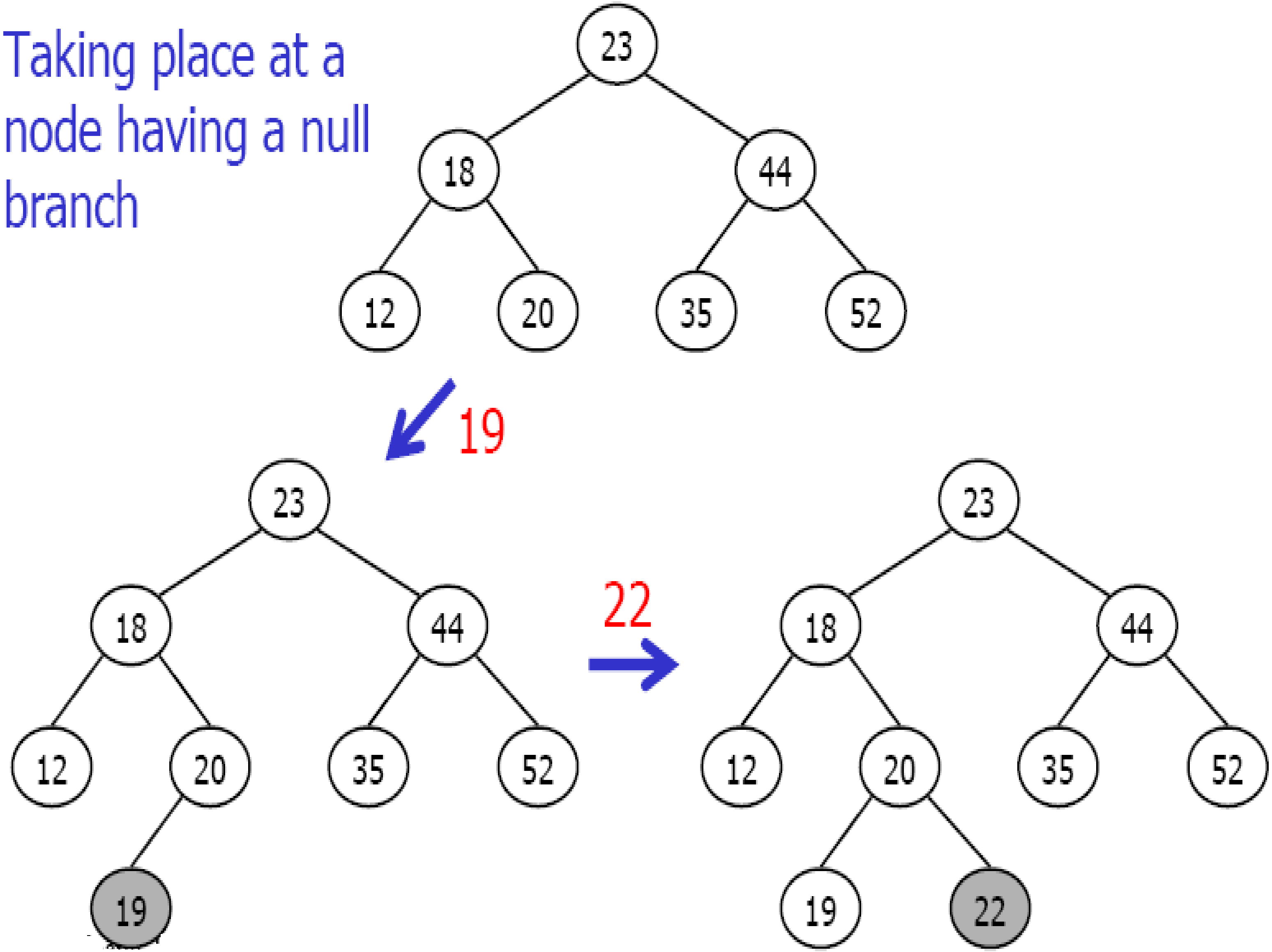


Search node in BST

- The same keys may be built into BST of many different shapes.
- Search in bushy BST with n nodes will do $O(\log n)$ comparisons of keys
- If the tree degenerates into a long chain, search will do $\Theta(n)$ comparisons on n vertices.
- The bushier the tree, the smaller the number of comparisons of keys need to be done.

Insert Node into BST

Taking place at a node having a null branch



Insert Node into BST

Question:

Can Insert method use recursive_Search or iterative_Search instead of recursive_Insert like that:

```
<ErrorCode> Insert (val DataIn <DataType>)

1.  pNode = recursive_Search (root, DataIn.key)
2.  if (pNode is NULL)
    1.  Allocate pNode
    2.  pNode->data = DataIn
    3.  return success
3.  else
    1.  return duplicate_error
End Insert
```

Insert Node into BST

Auxiliary functions for Insert:

recursive_Insert

iterative_Insert

Recursive Insert

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,
val DataIn <DataType>)

Inserts a new node into a BST.

Pre subroot points to the root of a tree/ subtree.
DataIn contains data to be inserted into the subtree.

Post If the key of DataIn already belongs to the subtree, *duplicate_error* is returned. Otherwise, DataIn is inserted into the subtree in such a way that the properties of a BST are preserved.

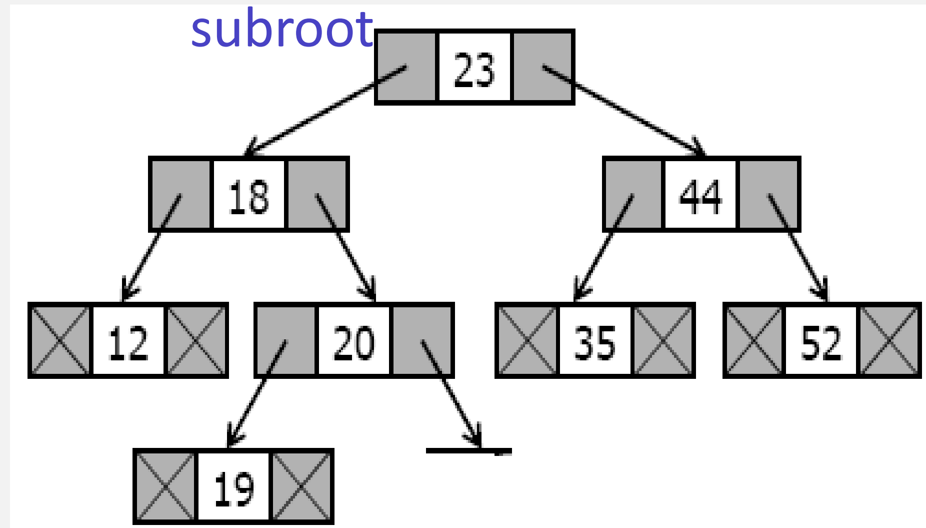
Return *duplicate_error* or *success*.

Uses recursive_Insert function.

Recursive Insert (cont.)

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,

val DataIn <DataType>)



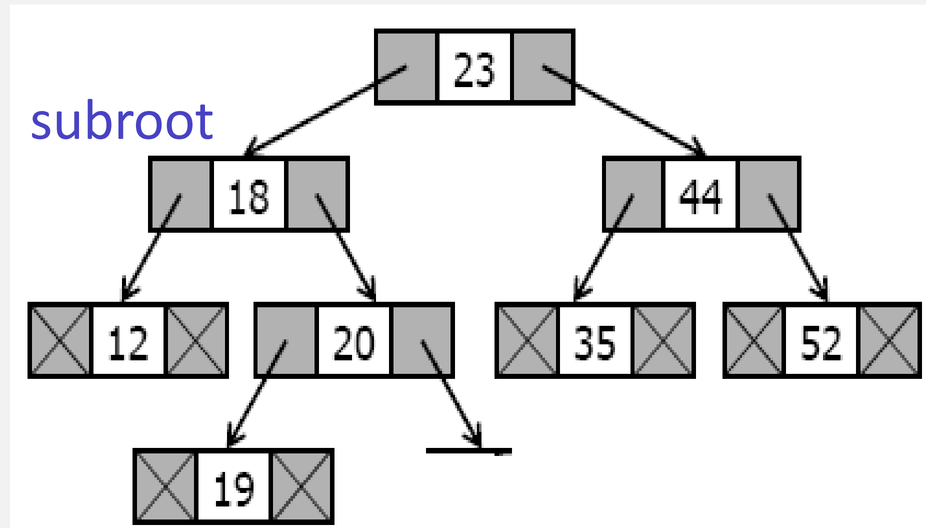
1. **if** (subroot is NULL)
    1. Allocate subroot
    2. subroot->data = DataIn
    3. return *success*
2. **else if** (DataIn.key < subroot->data.key)
    1. return recursive_Insert(subroot->left, DataIn)
3. **else if** (DataIn.key > subroot->data.key)
    1. return recursive_Insert(subroot->right, DataIn)
4. **else**
    1. return *duplicate_error*
5. End recursive_Insert

DataIn.key = 22

Recursive Insert (cont.)

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,

val DataIn <DataType>)



1. **if** (subroot is NULL)
   1. Allocate subroot
   2. subroot->data = DataIn
   3. return *success*

2. **else if** (DataIn.key < subroot->data.key)
   1. return recursive_Insert(subroot->left, DataIn)

3. **else if** (DataIn.key > subroot->data.key)
   1. return recursive_Insert(subroot->right, DataIn)

4. **else**
   1. return *duplicate_error*

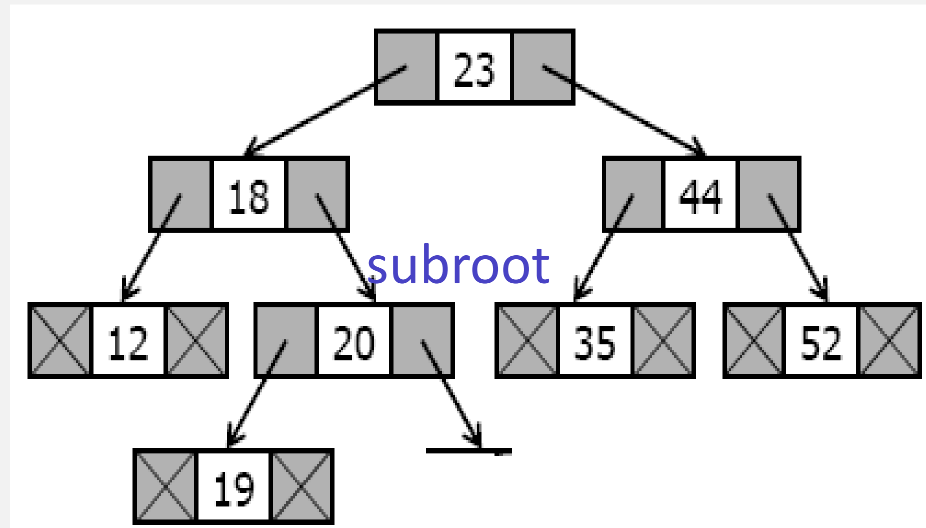
5. End recursive_Insert

DataIn.key = 22

Recursive Insert (cont.)

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,

val DataIn <DataType>)



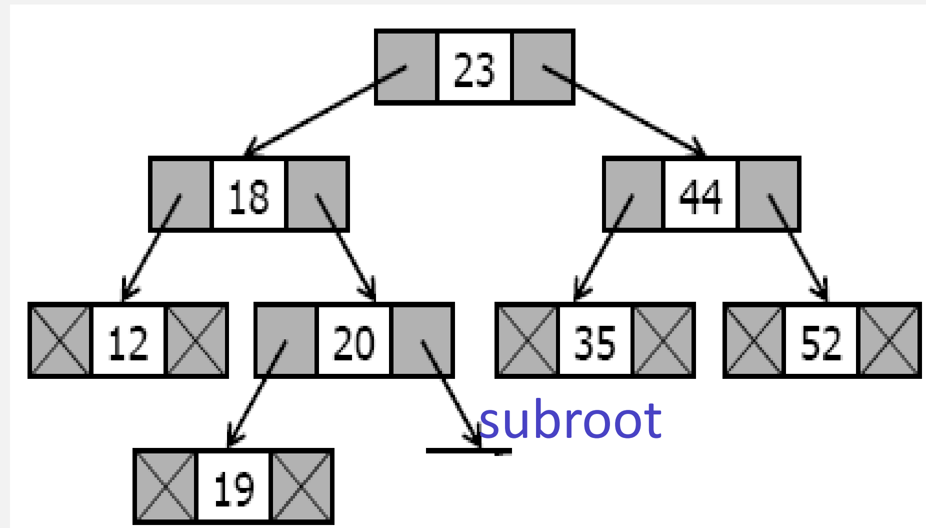
1. **if** (subroot is NULL)
    1. Allocate subroot
    2. subroot->data = DataIn
    3. return *success*
2. **else if** (DataIn.key < subroot->data.key)
    1. return recursive_Insert(subroot->left, DataIn)
3. **else if** (DataIn.key > subroot->data.key)
    1. return recursive_Insert(subroot->right, DataIn)
4. **else**
    1. return *duplicate_error*
5. End recursive_Insert

DataIn.key = 22

Recursive Insert (cont.)

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,

val DataIn <DataType>)



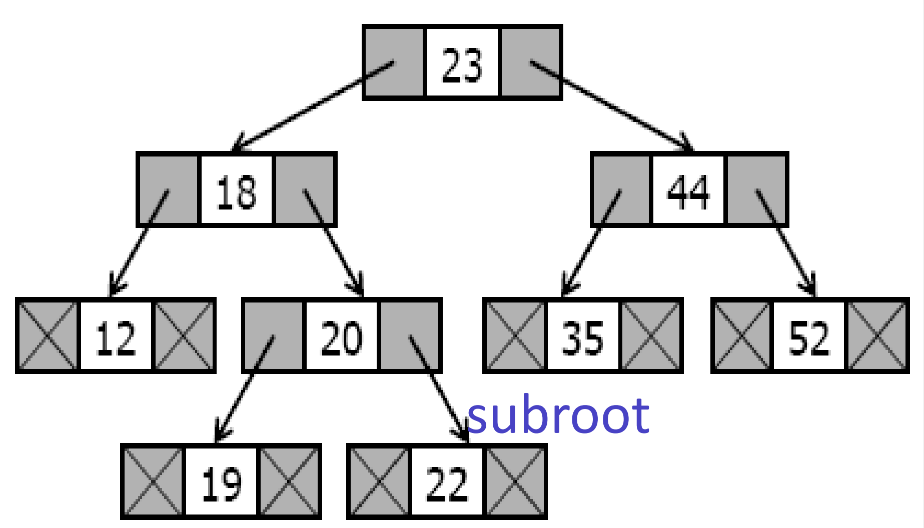
1. **if** (subroot is NULL)
    1. Allocate subroot
    2. subroot->data = DataIn
    3. return *success*
2. **else if** (DataIn.key < subroot->data.key)
    1. return recursive_Insert(subroot->left, DataIn)
3. **else if** (DataIn.key > subroot->data.key)
    1. return recursive_Insert(subroot->right, DataIn)
4. **else**
    1. return *duplicate_error*
5. End recursive_Insert

DataIn.key = 22

Recursive Insert (cont.)

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,

val DataIn <DataType>)



1. **if** (subroot is NULL)
   1. Allocate subroot
   2. subroot->data = DataIn
   3. return *success*
2. **else if** (DataIn.key < subroot->data.key)
   1. return recursive_Insert(subroot->left, DataIn)
3. **else if** (DataIn.key > subroot->data.key)
   1. return recursive_Insert(subroot->right, DataIn)
4. **else**
   1. return *duplicate_error*
5. End recursive_Insert

DataIn.key = 22

Iterative Insert

<ErrorCode> **iterative_Insert** (ref subroot <pointer>,
val DataIn <DataType>)

Inserts a new node into a BST.

Pre subroot is NULL or points to the root of a subtree. DataIn contains data to be inserted into the subtree.

Post If the key of DataIn already belongs to the subtree, *duplicate_error* is returned. Otherwise, DataIn is inserted into the subtree in such a way that the properties of a BST are preserved.

Return *duplicate_error* or *success*.

Iterative Insert (cont.)

<ErrorCode> **iterative_Insert** (ref subroot <pointer>,
val DataIn <DataType>)

1. **if** (subroot is NULL)
   1. Allocate subroot
   2. subroot->data = DataIn
   3. return *success*

2. **else**

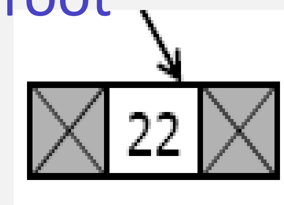
DataIn.key = 22

Iterative Insert (cont.)

<ErrorCode> **iterative_Insert** (ref subroot <pointer>,
val DataIn <DataType>)

1. **if** (subroot is NULL)
   1. Allocate subroot
   2. subroot->data = DataIn
   3. return *success*
2. **else**

subroot

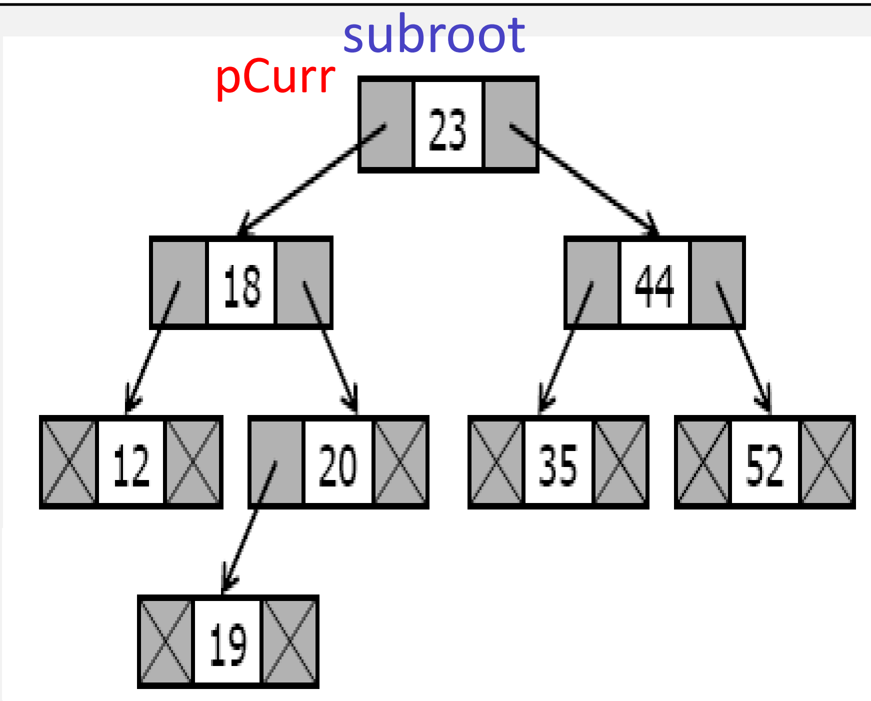


DataIn.key = 22

Iterative Insert (cont.)

2. else
   1. pCurr = subroot
   2. **loop** (pCurr is not NULL)
      1. **if** (pCurr->data.key = DataIn.key)
         1. return *duplicate_error*
      2. parent = pCurr
      3. **if** (DataIn.key < parent->data.key)
         1. pCurr = parent -> left
      4. **else**
         1. pCurr = parent -> right
   3. **if** (DataIn.key < parent->data.key)
      1. Allocate parent->left
      2. parent->left.data = DataIn
   4. **else**
      1. Allocate parent->right
      2. parent->right.data = DataIn
   5. return *success*

End Iterative_Insert

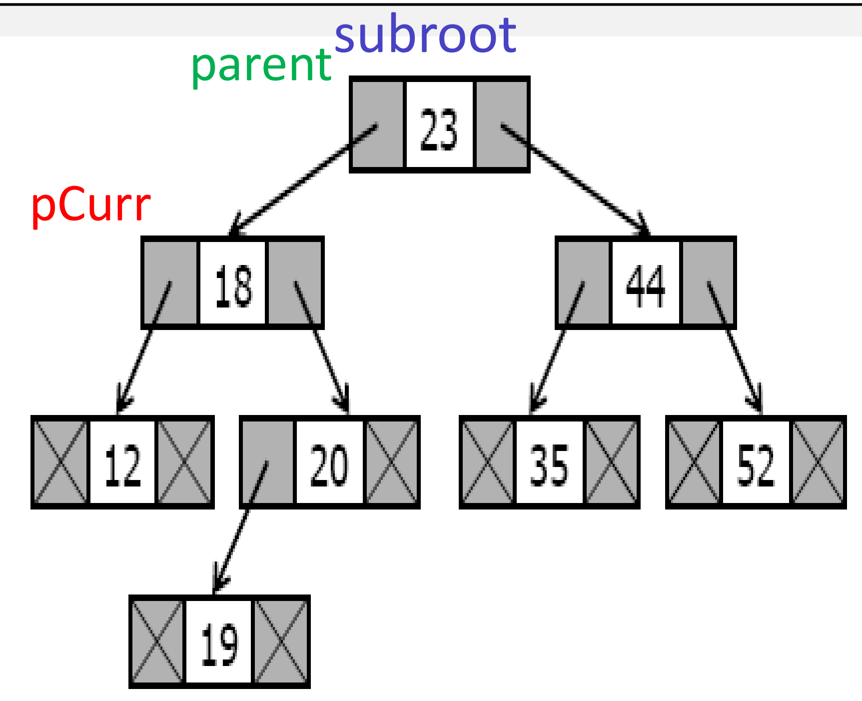


DataIn.key = 22

Iterative Insert (cont.)

2. else
   1. pCurr = subroot
   2. **loop** (pCurr is not NULL)
      1. **if** (pCurr->data.key = DataIn.key)
         1. return *duplicate_error*
      2. parent = pCurr
      3. **if** (DataIn.key < parent->data.key)
         1. pCurr = parent -> left
      4. **else**
         1. pCurr = parent -> right
   3. **if** (DataIn.key < parent->data.key)
      1. Allocate parent->left
      2. parent->left.data = DataIn
   4. **else**
      1. Allocate parent->right
      2. parent->right.data = DataIn
   5. return *success*

End Iterative_Insert

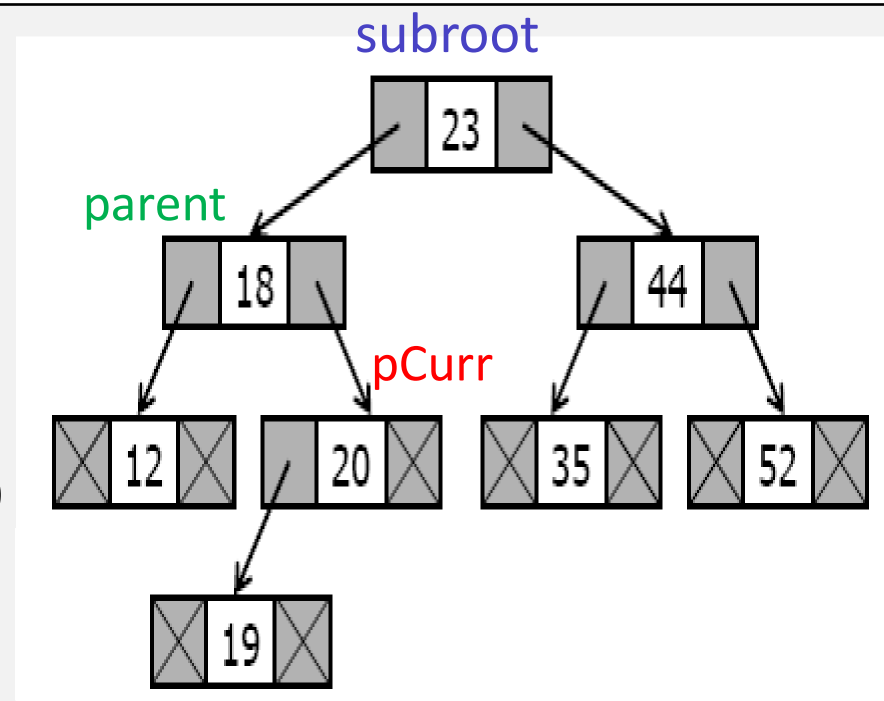


DataIn.key = 22

Iterative Insert (cont.)

2. else
    1. pCurr = subroot
    2. **loop** (pCurr is not NULL)
        1. **if** (pCurr->data.key = DataIn.key)
            1. return *duplicate_error*
        2. parent = pCurr
        3. **if** (DataIn.key < parent->data.key)
            1. pCurr = parent -> left
        4. **else**
            1. pCurr = parent -> right
    3. **if** (DataIn.key < parent->data.key)
        1. Allocate parent->left
        2. parent->left.data = DataIn
    4. **else**
        1. Allocate parent->right
        2. parent->right.data = DataIn
    5. return *success*

End Iterative_Insert

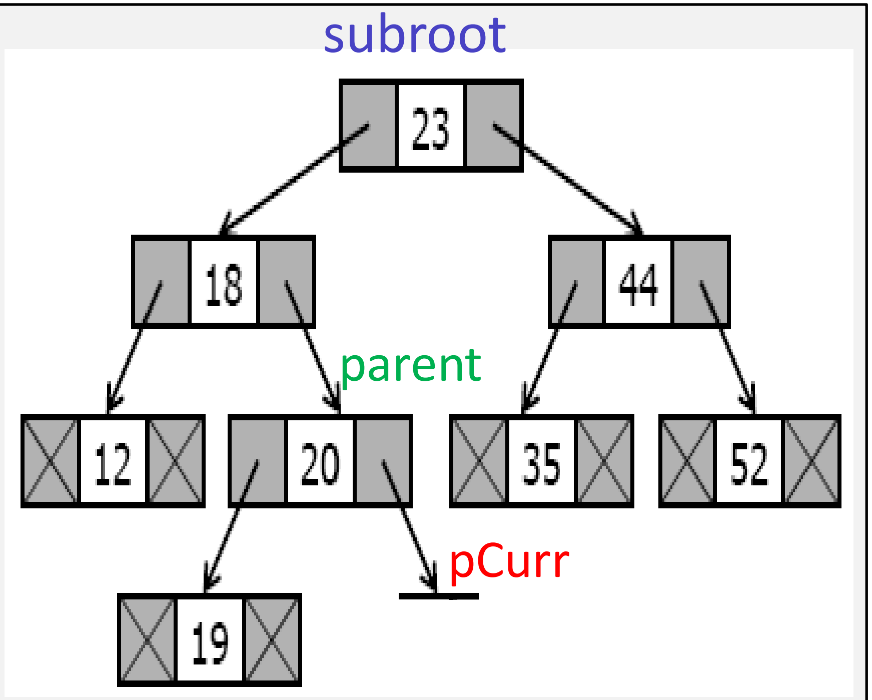


DataIn.key = 22

Iterative Insert (cont.)

2. else
    1. pCurr = subroot
    2. **loop** (pCurr is not NULL)
        1. **if** (pCurr->data.key = DataIn.key)
            1. return *duplicate_error*
        2. parent = pCurr
        3. **if** (DataIn.key < parent->data.key)
            1. pCurr = parent -> left
        4. **else**
            1. pCurr = parent -> right
    3. **if** (DataIn.key < parent->data.key)
        1. Allocate parent->left
        2. parent->left.data = DataIn
    4. **else**
        1. Allocate parent->right
        2. parent->right.data = DataIn
    5. return *success*

End Iterative_Insert

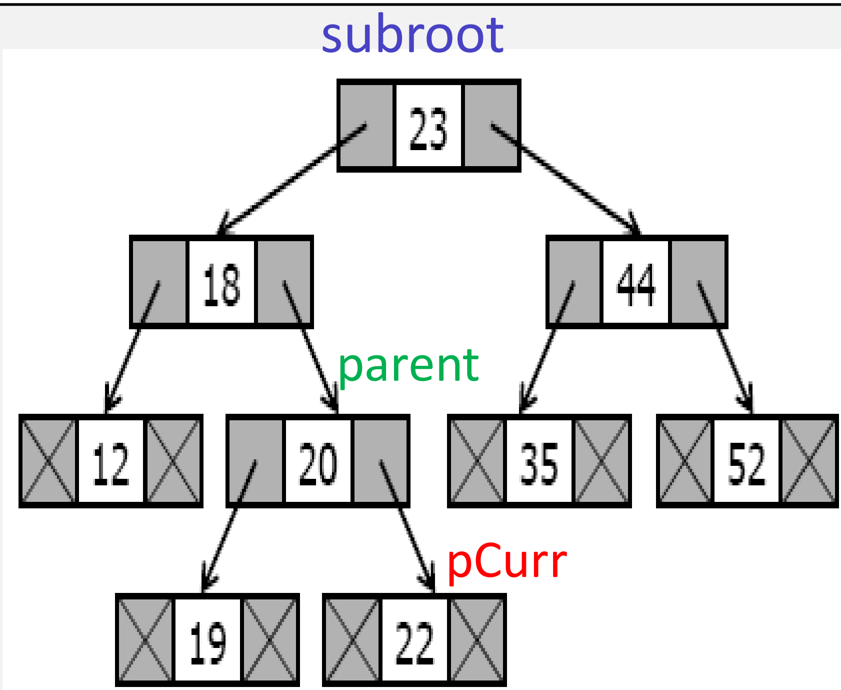


DataIn.key = 22

Iterative Insert (cont.)

2. else
    1. pCurr = subroot
    2. **loop** (pCurr is not NULL)
        1. **if** (pCurr->data.key = DataIn.key)
            1. return *duplicate_error*
        2. parent = pCurr
        3. **if** (DataIn.key < parent->data.key)
            1. pCurr = parent -> left
        4. **else**
            1. pCurr = parent -> right
    3. **if** (DataIn.key < parent->data.key)
        1. Allocate parent->left
        2. parent->left.data = DataIn
    4. **else**
        1. Allocate parent->right
        2. parent->right.data = DataIn
    5. return *success*

End Iterative_Insert



DataIn.key = 22

Insert Node into BST

<ErrorCode> **Insert** (val DataIn <DataType>)

Inserts a new node into a BST.

Post If the key of DataIn already belongs to the BST, *duplicate_error* is returned. Otherwise, DataIn is inserted into the tree in such a way that the properties of a BST are preserved.

Return *duplicate_error* or *success*.

Uses recursive_Insert or iterative_Insert function.

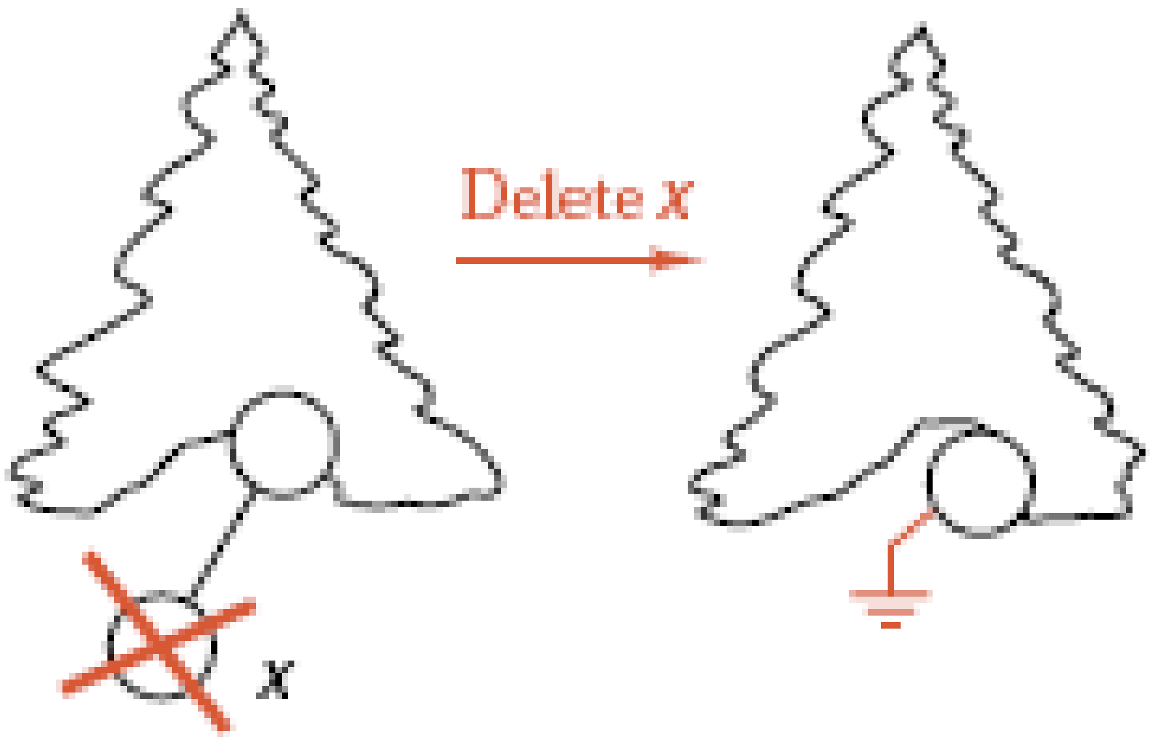
1. return recursive_Insert (root, DataIn)

End Insert

Insert Node into BST

- Insertion a new node into a random BST with n nodes takes O(log n) steps.
- Insertion may take n steps when BST degenerates to a chain.
- If the keys are inserted in sorted order into an empty tree, BST becomes a chain.

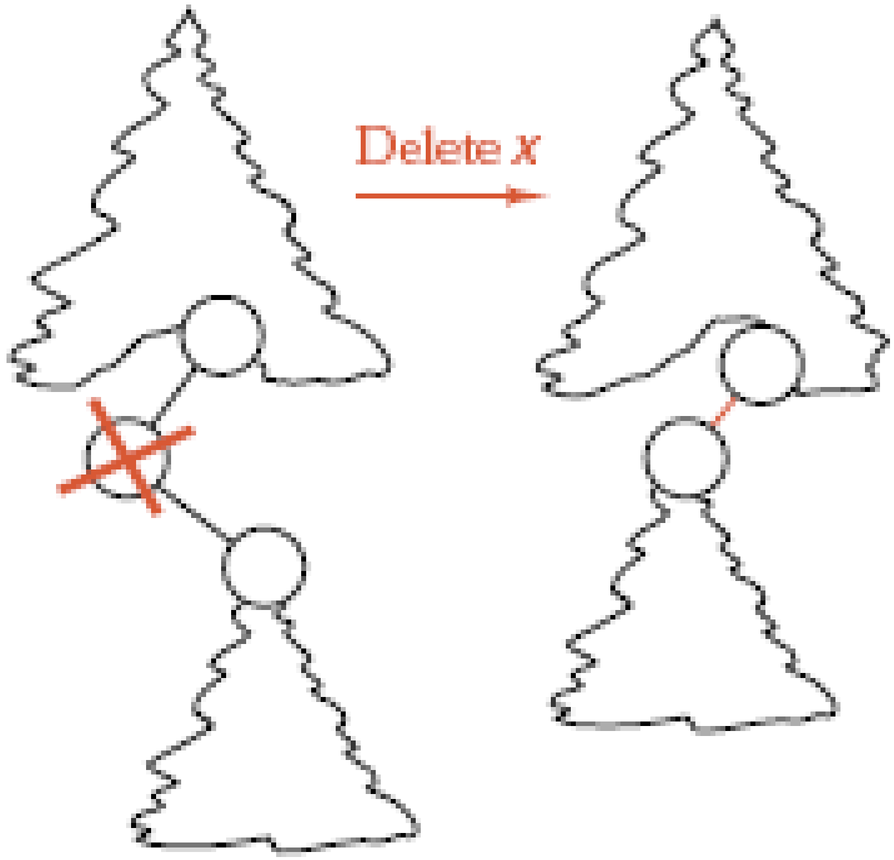
Delete node from BST



- Deletion of a leaf:

  Set the deleted node's parent link to NULL.

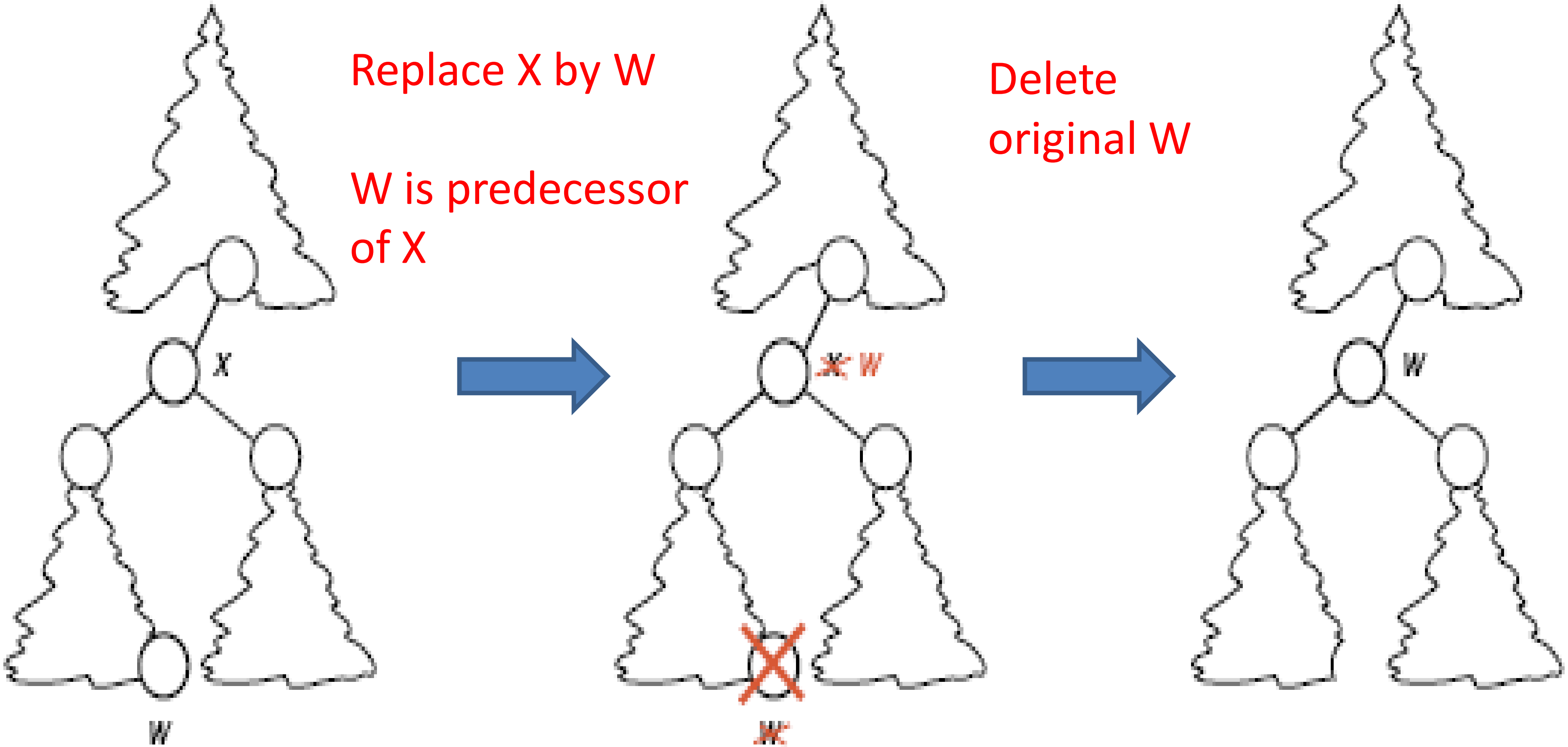
Delete node from BST



- Deletion of a node having only right subtree or left subtree:

  Attach the subtree to the deleted node's parent.

Delete node from BST

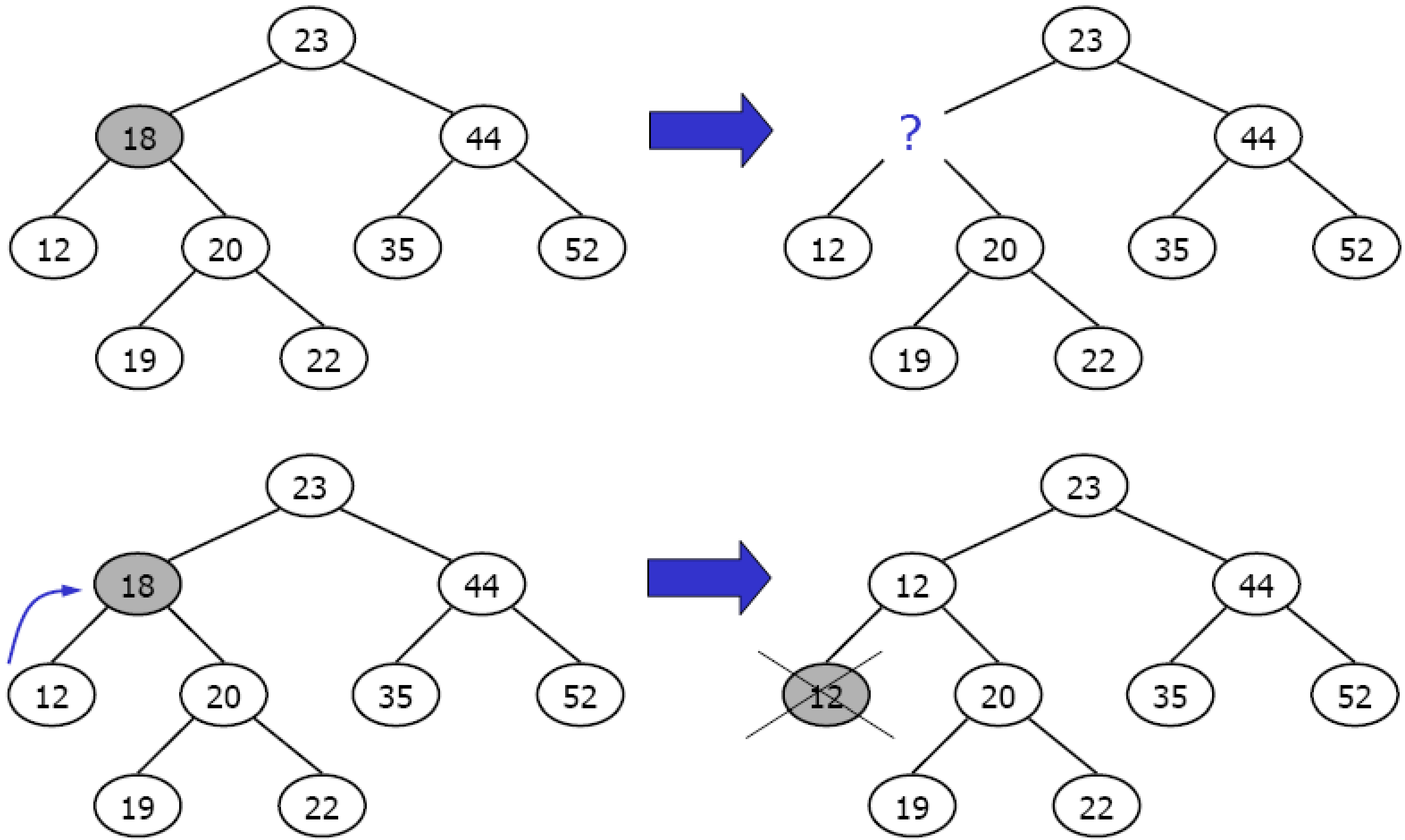


- Deletion of a node having both subtrees:

  Replace the deleted node by its predecessor or by its successor, recycle this node instead.

Delete node from BST

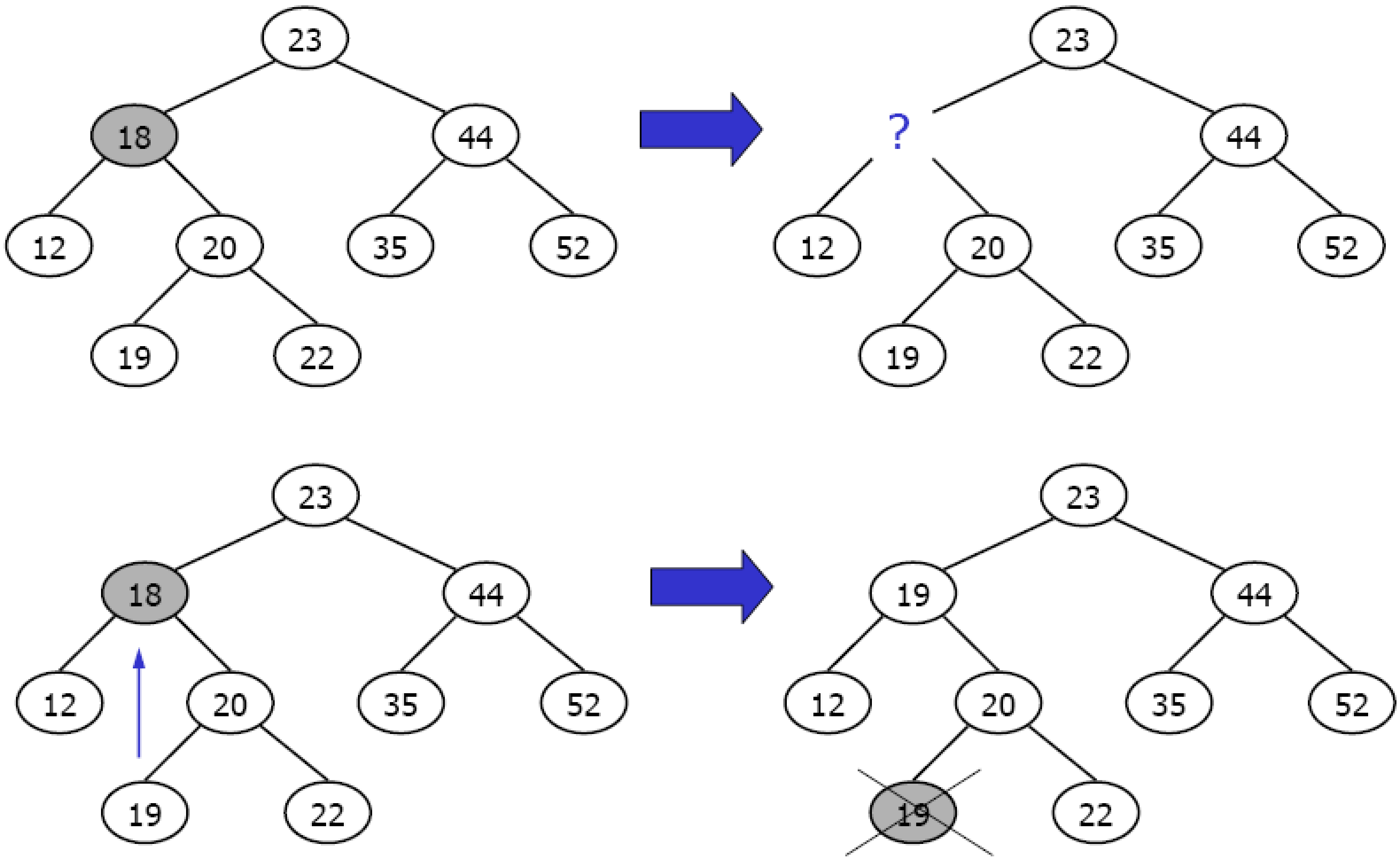
- Node having both subtrees



Using largest node in the left subtree

Delete node from BST

- Node having both subtrees



Using smallest node in the right subtree

Delete node from BST

Auxiliary functions for Insert:

recursive_Delete

iterative_Delete

Recursive Delete

<ErrorCode> **recursive_Delete** (ref subroot <pointer>,
val key <KeyType>)

Deletes a node from a BST.

Pre subroot is NULL or points to the root of a subtree. Key contains value needs to be removed from BST.

Post If key is found, it will be removed from BST.

Return *notFound* or *success*.

Uses recursive_Delete and RemoveNode functions.

Recursive Delete (cont.)

<ErrorCode> **recursive_Delete** (ref subroot <pointer>,
val key <KeyType>)

1. **if** (subroot is NULL)
    1. return *notFound*
2. **else if** (key < subroot->data.key)
    1. return recursive_Delete(subroot->left, key)
3. **else if** (key > subroot->data.key)
    1. return recursive_Delete(subroot->right, key)
4. **else**
    1. RemoveNode(subroot)
    2. return *success*

End recursive_Delete

Delete Node from BST

<ErrorCode> **Delete** (val key <KeyType>)

Deletes a node from a BST.

Pre subroot is NULL or points to the root of a subtree. Key contains value needs to be removed from BST.

Post If key is found, it will be removed from BST.

Return *notFound* or *success*.

Uses recursive_Delete and RemoveNode functions.

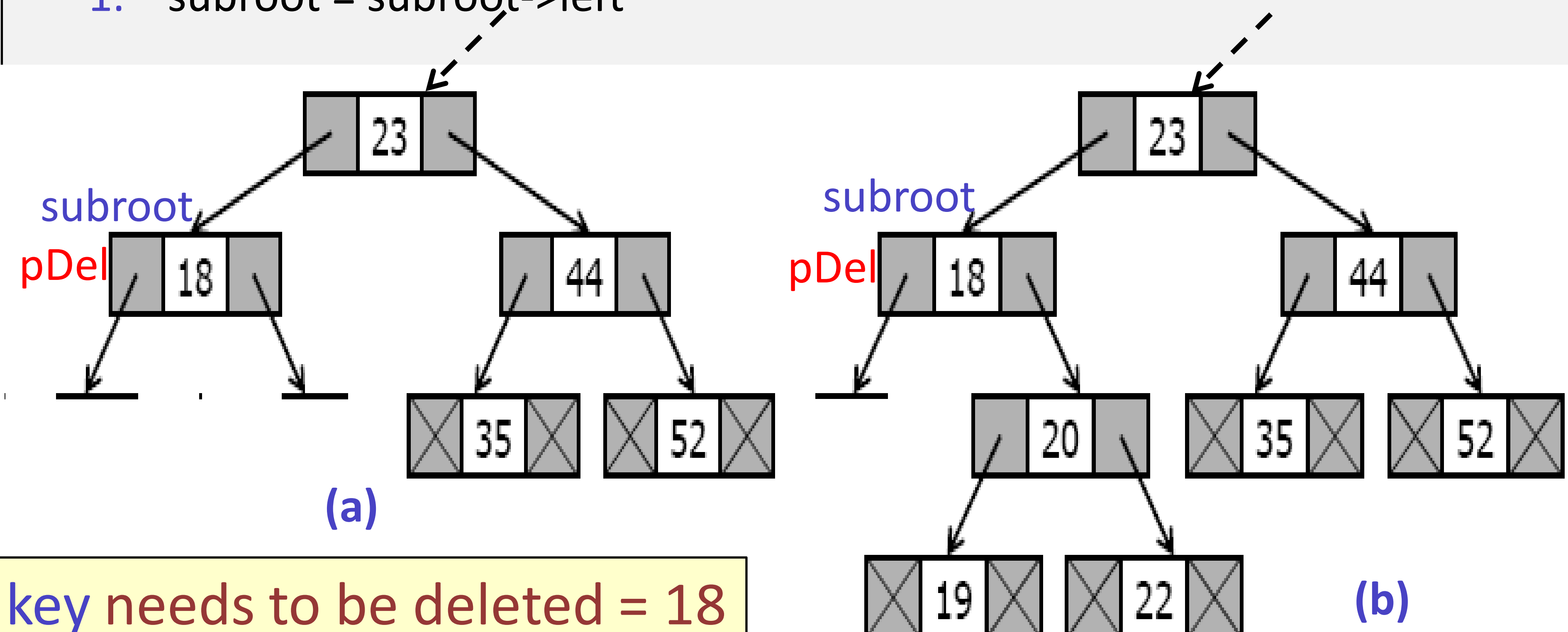
1. return recursive_Delete (root, key)

End Delete

Auxiliary Function RemoveNode

<void> **RemoveNode** (ref subroot <pointer>, val key <KeyType>)

1. pDel = subroot // remember node to delete at end.
2. **if** (subroot ->left is NULL) // leaf node or node having only right subtree.
    1. subroot = subroot->right **// (a) and (b)**
3. **else if** (subroot->right is NULL) // node having only left subtree.
    1. subroot = subroot->left

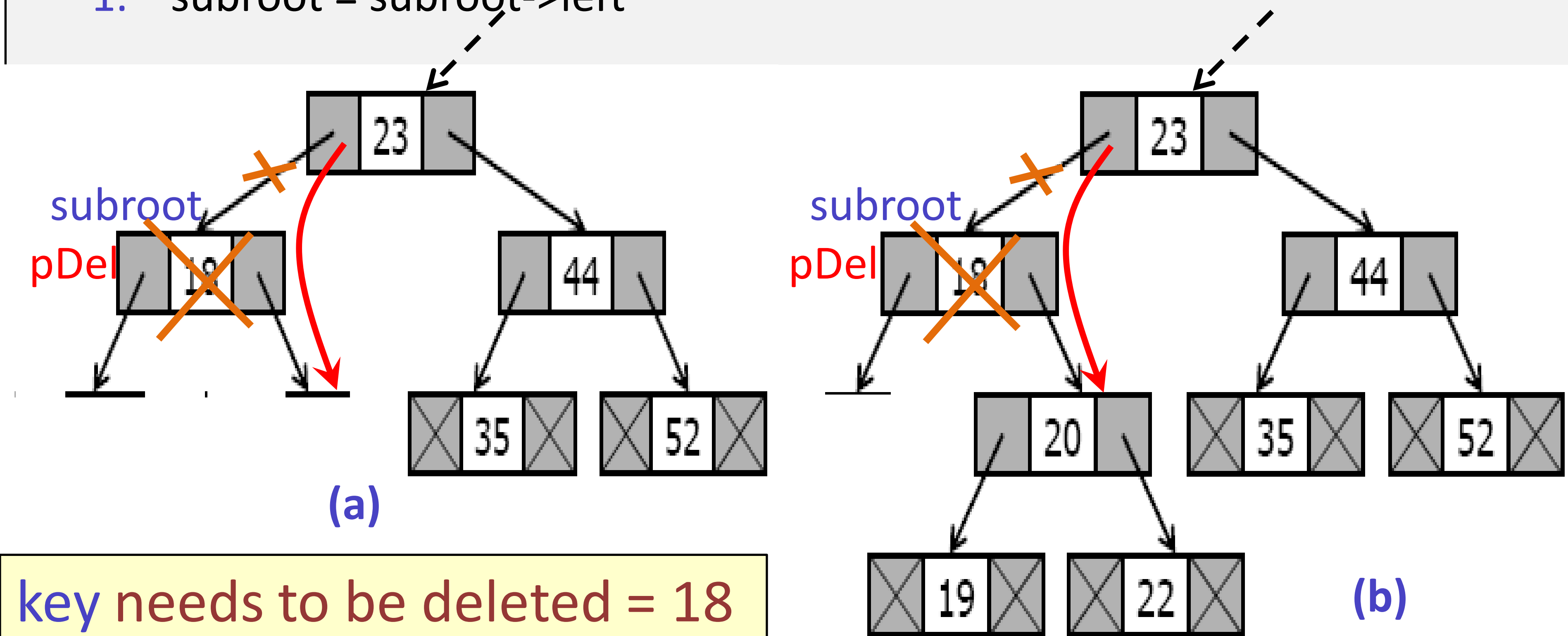


key needs to be deleted = 18

Auxiliary Function RemoveNode

<void> **RemoveNode** (ref subroot <pointer>, val key <KeyType>)

1. pDel = subroot // remember node to delete at end.
2. **if** (subroot ->left is NULL) // leaf node or node having only right subtree.
    1. subroot = subroot->right **// (a) and (b)**
3. **else if** (subroot->right is NULL) // node having only left subtree.
    1. subroot = subroot->left

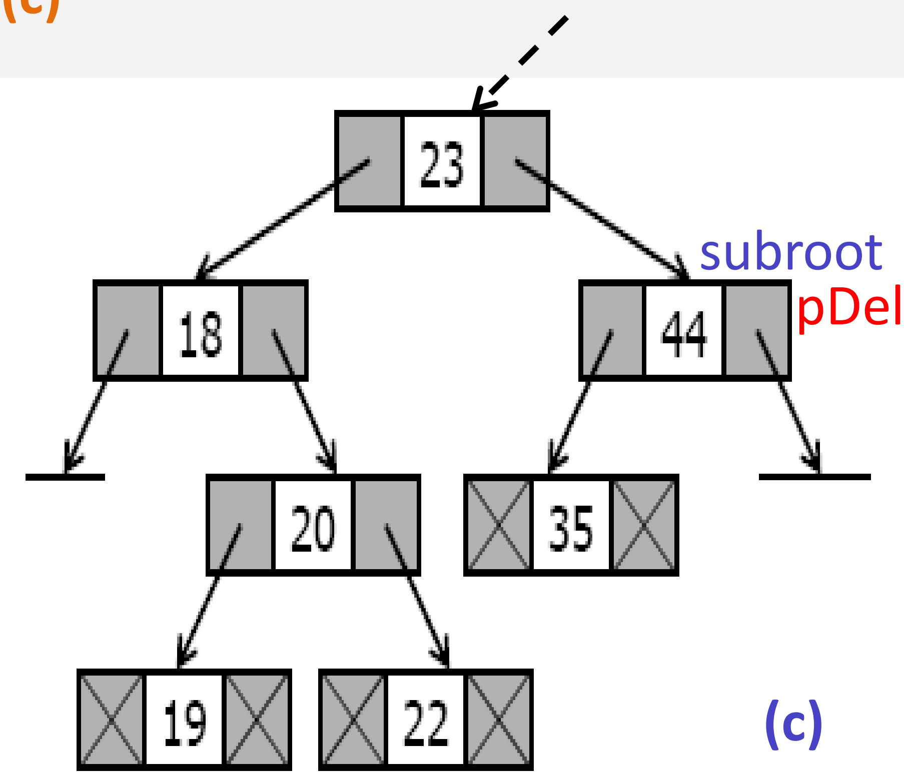


key needs to be deleted = 18

Auxiliary Function RemoveNode

<void> **RemoveNode** (ref subroot <pointer>, val key <KeyType>)

1. pDel = subroot // remember node to delete at end.
2. **if** (subroot ->left is NULL) // leaf node or node having only right subtree.
    1. subroot = subroot->right
3. **else if** (subroot->right is NULL) // node having only left subtree.
    1. subroot = subroot->left **// (c)**

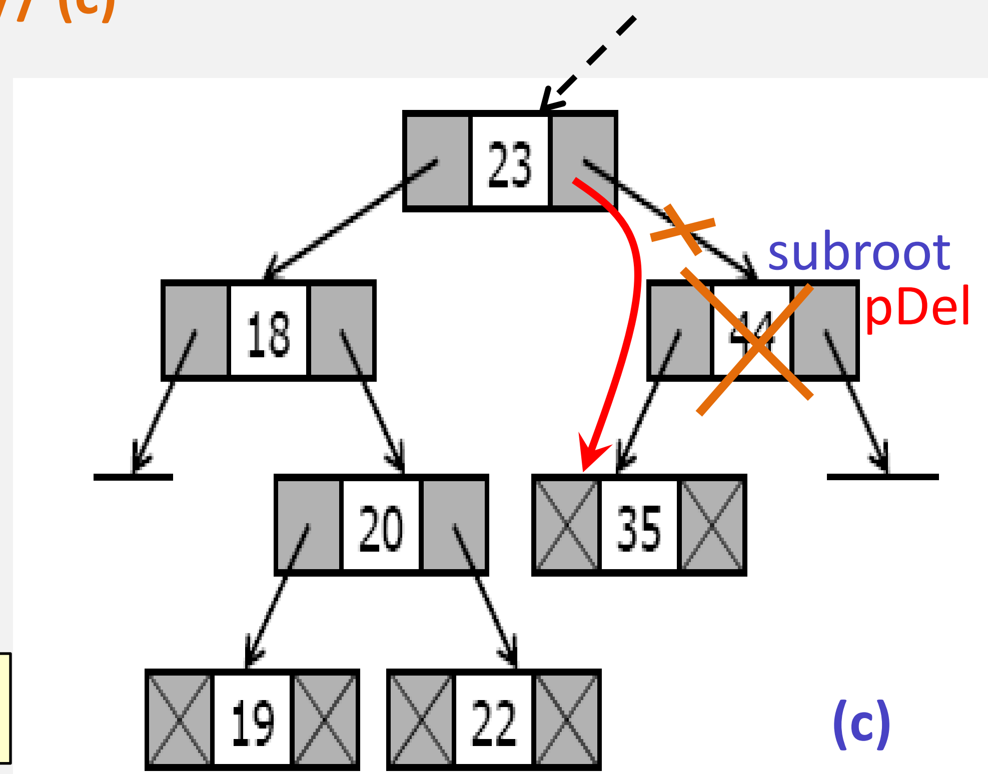


key needs to be deleted = 44

Auxiliary Function RemoveNode

<void> **RemoveNode** (ref subroot <pointer>, val key <KeyType>)

1. pDel = subroot // remember node to delete at end.
2. **if** (subroot ->left is NULL) // leaf node or node having only right subtree.
    1. subroot = subroot->right
3. **else if** (subroot->right is NULL) // node having only left subtree.
    1. subroot = subroot->left **// (c)**



key needs to be deleted = 44

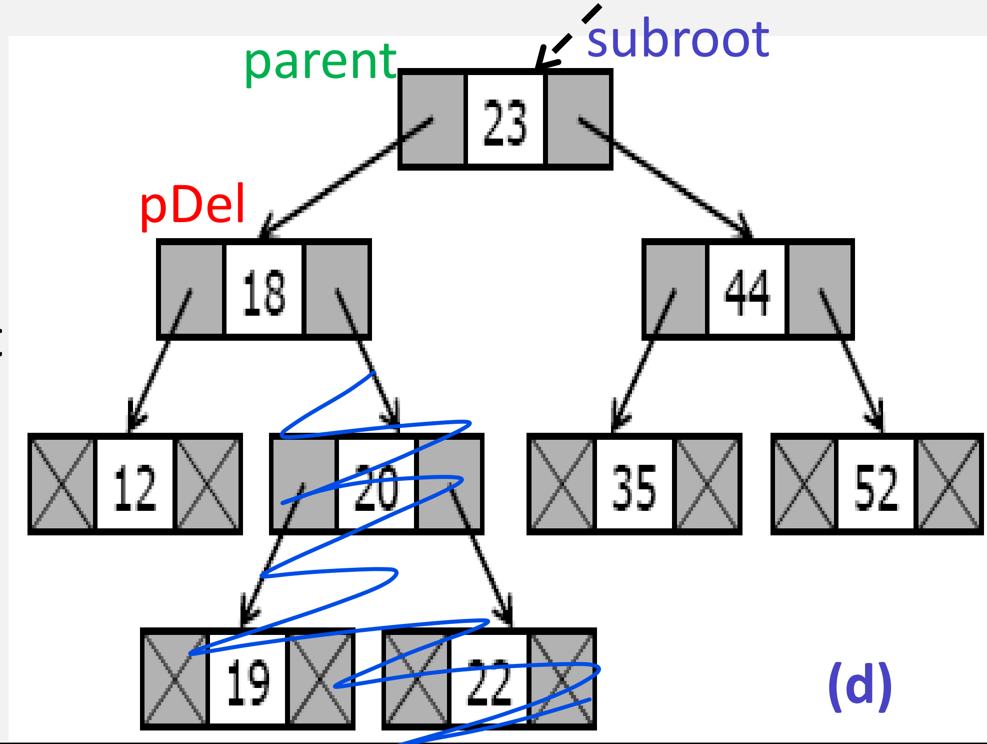
Auxiliary Function RemoveNode (cont.)

4. else // node having both subtrees. **(d)**

1. parent = subroot
2. pDel = parent ->left // move left to find the predecessor.
3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
    1. parent = pDel
    2. pDel = pDel->right
4. subroot->data = pDel->data
5. if (parent = subroot)
    1. parent->left = pDel->left
6. else
    1. parent->right = pDel->left
7. recycle pDel

End RemoveNode

key needs to be deleted = 23



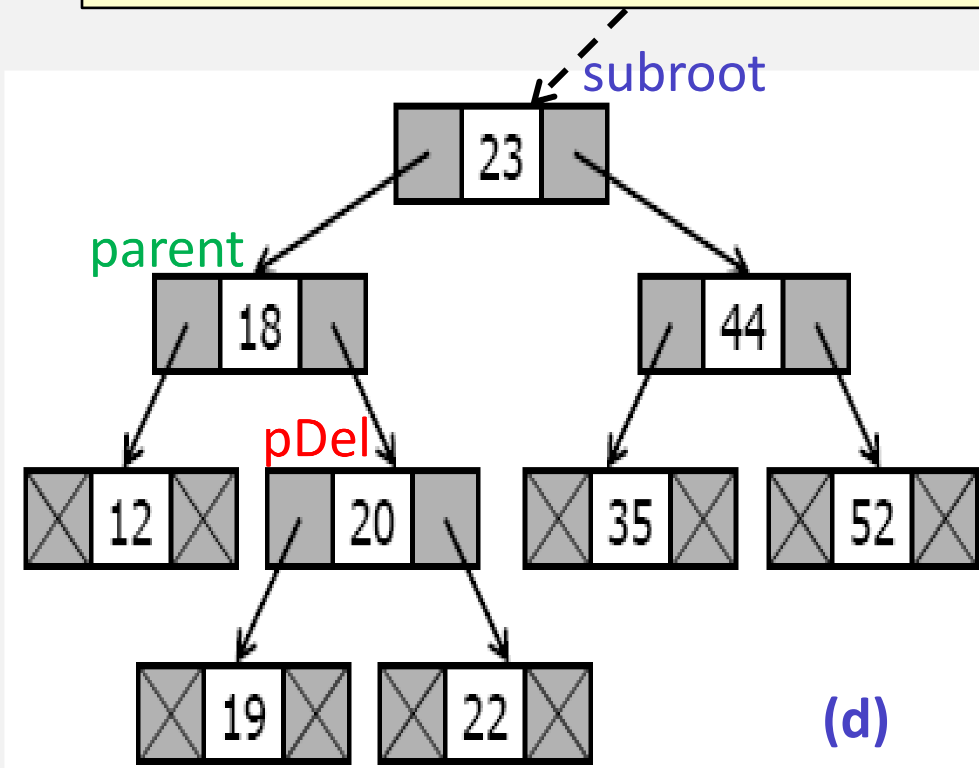
(d)

Auxiliary Function RemoveNode (cont.)

1. **1.** else // node having both subtrees. **(d)**
    1. parent = subroot
    2. pDel = parent ->left // move left to find the predecessor.
    3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
        1. parent = pDel
        2. pDel = pDel->right
    4. subroot->data = pDel->data
    5. if (parent = subroot)
        1. parent->left = pDel->left
    6. else
        1. parent->right = pDel->left
    7. recycle pDel

End RemoveNode

key needs to be deleted = 23

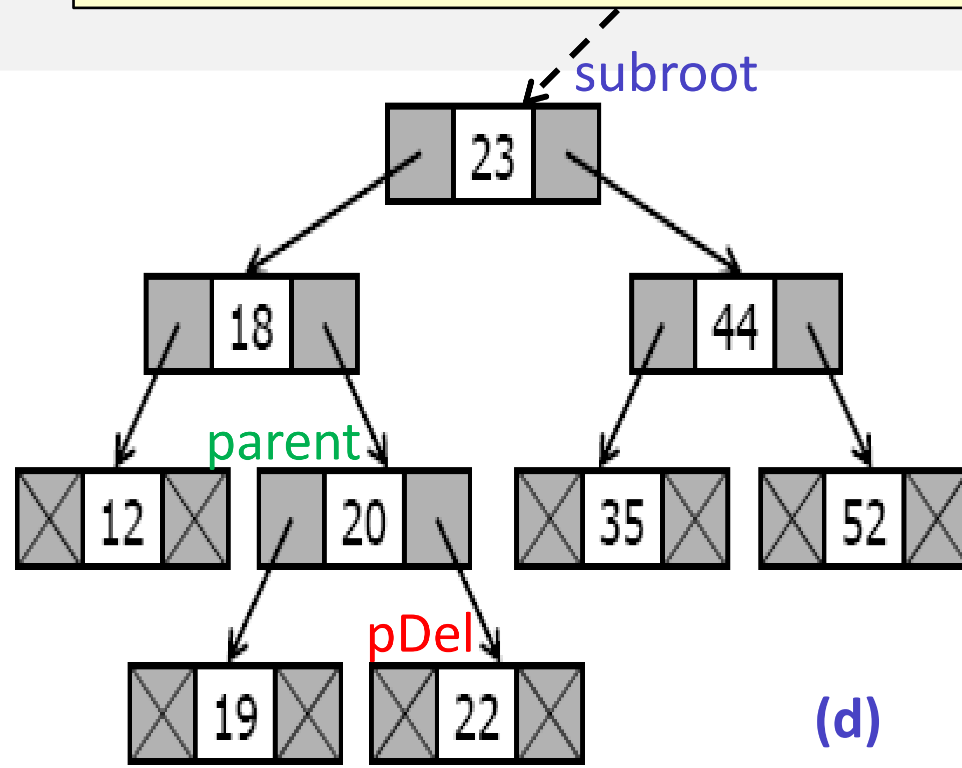


Auxiliary Function RemoveNode (cont.)

1. else // node having both subtrees. **(d)**
    1. parent = subroot
    2. pDel = parent ->left // move left to find the predecessor.
    3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
        1. parent = pDel
        2. pDel = pDel->right
    4. subroot->data = pDel->data
    5. if (parent = subroot)
        1. parent->left = pDel->left
    6. else
        1. parent->right = pDel->left
    7. recycle pDel

End RemoveNode

key needs to be deleted = 23

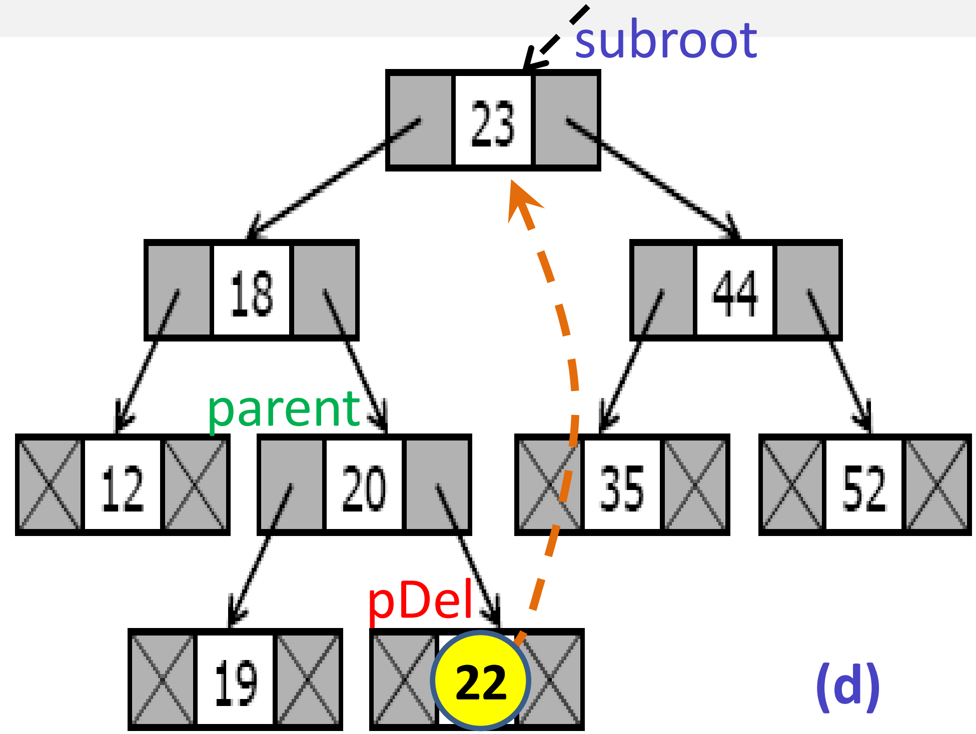


Auxiliary Function RemoveNode (cont.)

1. else // node having both subtrees. **(d)**
    1. parent = subroot
    2. pDel = parent ->left // move left to find the predecessor.
    3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
        1. parent = pDel
        2. pDel = pDel->right
    4. subroot->data = pDel->data
    5. if (parent = subroot)
        1. parent->left = pDel->left
    6. else
        1. parent->right = pDel->left
    7. recycle pDel

End RemoveNode

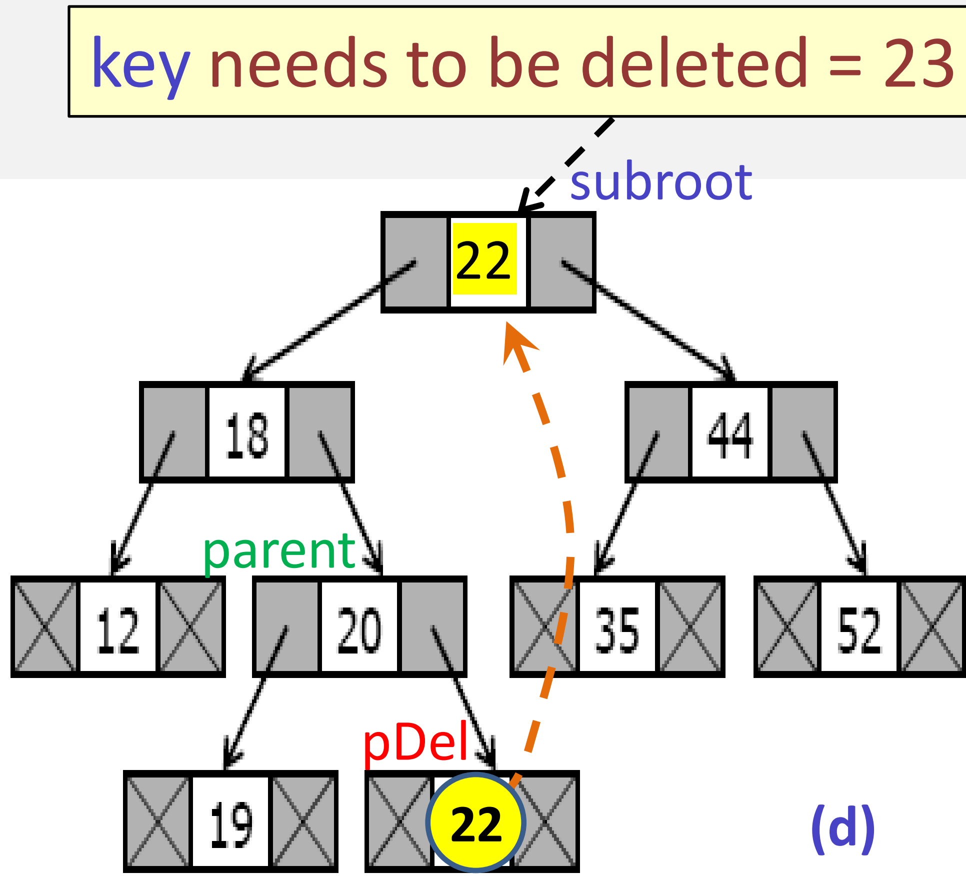
key needs to be deleted = 23



Auxiliary Function RemoveNode (cont.)

1. else // node having both subtrees. **(d)**
    1. parent = subroot
    2. pDel = parent ->left // move left to find the predecessor.
    3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
        1. parent = pDel
        2. pDel = pDel->right
    4. subroot->data = pDel->data
    5. if (parent = subroot)
        1. parent->left = pDel->left
    6. else
        1. parent->right = pDel->left
    7. recycle pDel

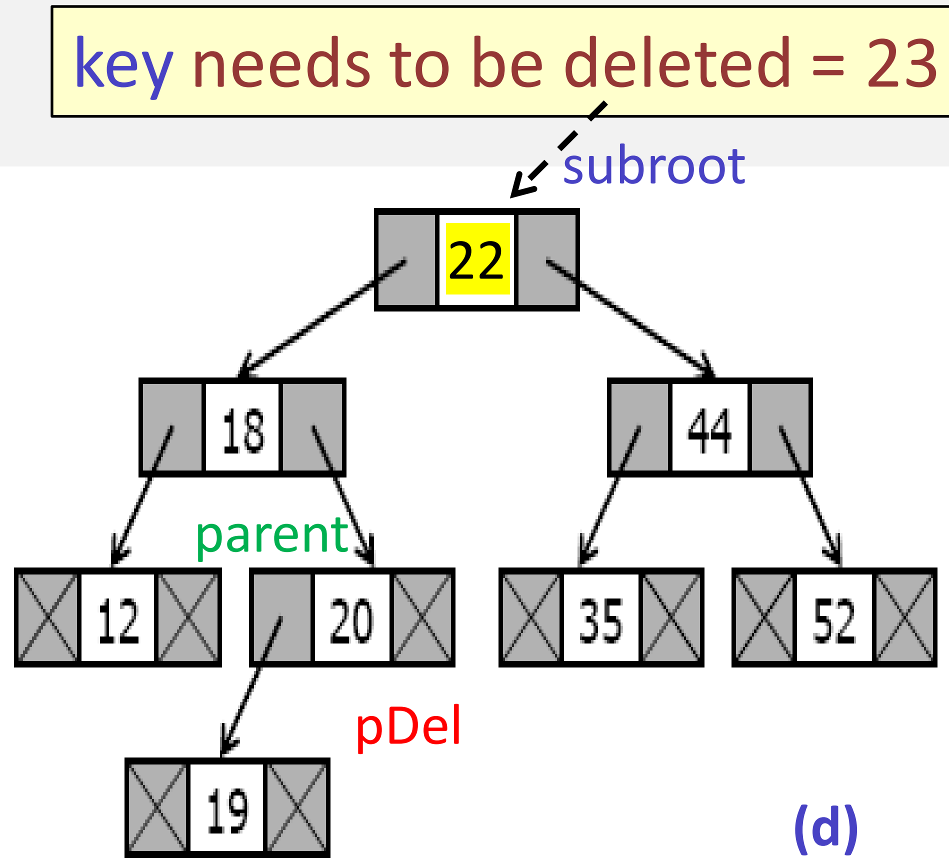
End RemoveNode



Auxiliary Function RemoveNode (cont.)

1. else // node having both subtrees. **(d)**
    1. parent = subroot
    2. pDel = parent ->left // move left to find the predecessor.
    3. **loop** (pDel->right is not NULL) // pDel is not the predecessor
        1. parent = pDel
        2. pDel = pDel->right
    4. subroot->data = pDel->data
    5. if (parent = subroot)
        1. parent->left = pDel->left
    6. else
        1. parent->right = pDel->left
    7. recycle pDel

End RemoveNode



Performance of random BST

- The average number of nodes visited during a search of average BST with n nodes approximately 2 ln2 = (2 ln 2) (lg n)$\approx 1.39 \lg n$

- The average BST requires approximately 2 ln2 ≈ 1.39 times as many comparisons as a completely balanced tree.