Chapter 6
Trees

Data Structures and Algorithms

Luong The Nhan, Tran Giang Son
Faculty of Computer Science and Engineering
University of Technology, VNU-HCM

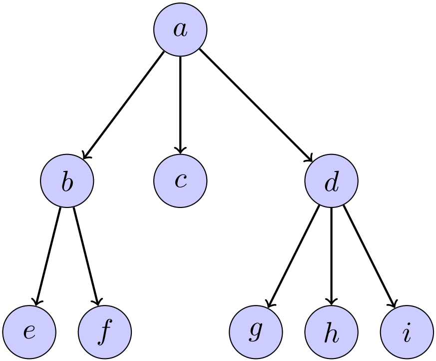
Contents

Basic Tree Concepts

Basic Tree Concepts

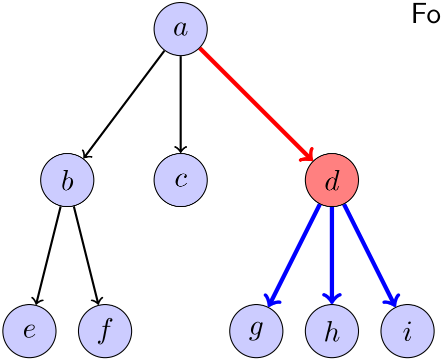
Definition

A tree (cây) consists of a finite set of elements, called nodes (nút), and a finite set of directed lines, called branches (nhánh), that connect the nodes.



Basic Tree Concepts

- Degree of a node (Bậc của nút): the number of branches associated with the node.
- Indegree branch (Nhánh vào): directed branch toward the node.
- Outdegree branch (Nhánh ra): directed branch away from the node.

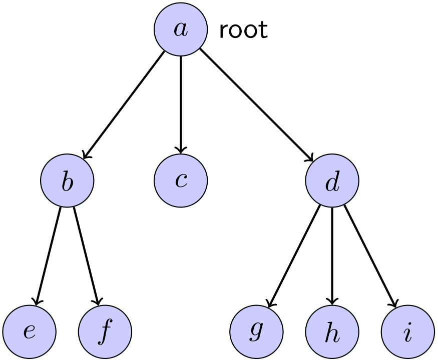


For the node d:

- **Degree** $= 4$
- **Indegree branches**: ad
  $\rightarrow$ indegree $= 1$
- **Outdegree branches**: dg, dh, di
  $\rightarrow$ outdegree $= 3$

Basic Tree Concepts

- The first node is called the root.
- indegree of the root = 0
- Except the root, the indegree of a node = 1
- outdegree of a node = 0 or 1 or more.



Basic Tree Concepts

Terms

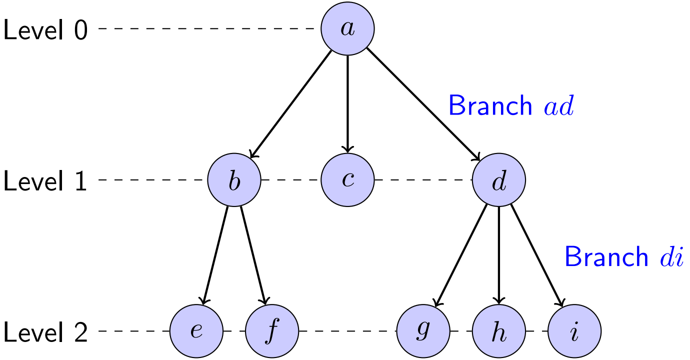
- A root (nút gốc) is the first node with an indegree of zero.
- A leaf (nút lá) is any node with an outdegree of zero.
- A internal node (nút nội) is not a root or a leaf.
- A parent (nút cha) has an outdegree greater than zero.
- A child (nút con) has an indegree of one.
  $\rightarrow$ a internal node is both a parent of a node and a child of another one.
- Siblings (nút anh em) are two or more nodes with the same parent.
- For a given node, an ancestor is any node in the path from the root to the node.
- For a given node, an descendent is any node in the paths from the node to a leaf.

Basic Tree Concepts

Terms

- A path (đường đi) is a sequence of nodes in which each node is adjacent to the next one.
- The level (bậc) of a node is its distance from the root. $\rightarrow$ Siblings are always at the same level.
- The height (độ cao) of a tree is the level of the leaf in the longest path from the root plus 1.
- A subtree (cây con) is any connected structure below the root.

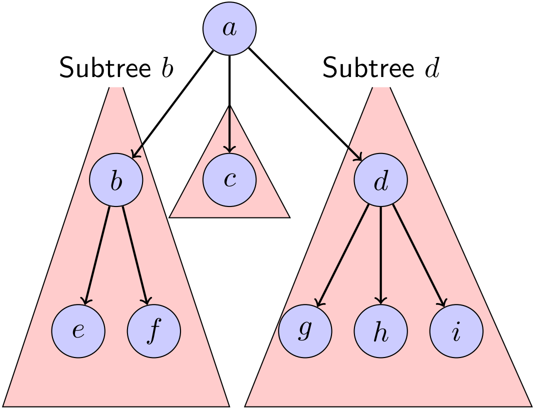
Basic Tree Concepts



- Parents: a, b, d
- Children: b, c, d, e, f, g, h, i
- Leaves: c, e, f, g, h, i
- Internal nodes: b, d
- Siblings: $\{b, c, d\}, \{e, f\}, \{g, h, i\}$
- Height $= 3$

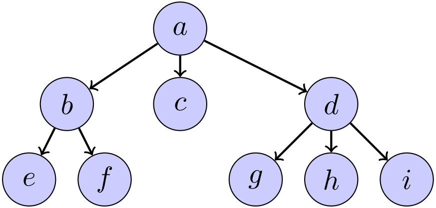
Luong The Nhan,
Tran Giang Son

Basic Tree Concepts



Tree representation

- organization chart



- indented list
  - a
    - b
      - e
      - f
    - c
    - d
      - g
      - h
      - i

- parenthetical listing

  $a\,(b\,(e\ f)\ c\ d\,(g\ h\ i))$

Applications of Trees

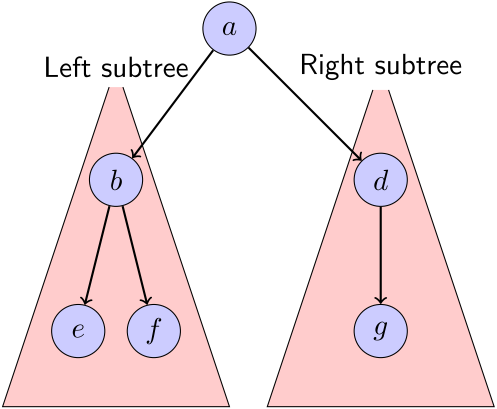
- Representing hierarchical data
- Storing data in a way that makes it easily searchable (ex: binary search tree)
- Representing sorted lists of data
- Network routing algorithms

Binary Trees

Luong The Nhan,
Tran Giang Son

Binary Trees

A binary tree node cannot have more than two subtrees.



Binary Trees Properties

- To store N nodes in a binary tree:
  - The minimum height: $H_{min} = \lfloor \log_2 N \rfloor + 1$
  - The maximum height: $H_{max} = N$

- Given a height of the binary tree, H:
  - The minimum number of nodes: $N_{min} = H$
  - The maximum number of nodes: $N_{max} = 2^H - 1$

Balance

The balance factor of a binary tree is the difference in height between its left and right subtrees.

$$B = H_L - H_R$$

Balanced tree:

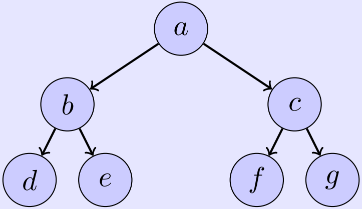
- balance factor is 0, -1, or 1
- subtrees are balanced

Binary Trees Properties

Complete tree

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

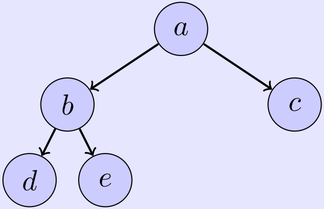
The last level is full.



Nearly complete tree

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

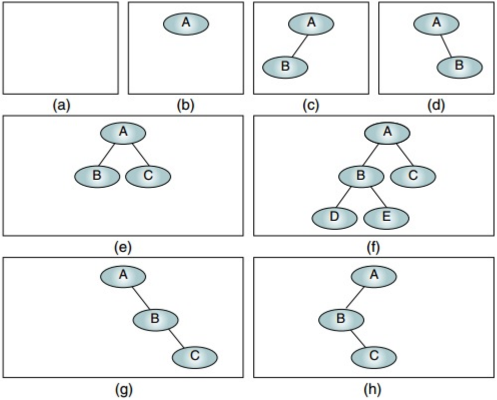
Nodes in the last level are on the left.



Binary Tree Structure

Definition

A binary tree 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.

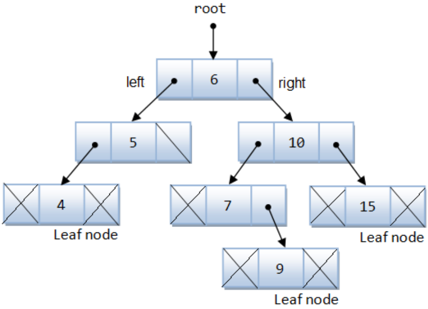


Binary Tree Structure: Linked implementation

```
node
  data <dataType>
  left <pointer>
  right <pointer>
end node
```

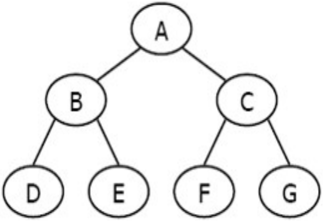
```
binaryTree
  root <pointer>
end binaryTree
```

```
// General dataTye:
dataType
  key <keyType>
  field1 <...>
  field2 <...>
  ...
  fieldn <...>
end dataType
```



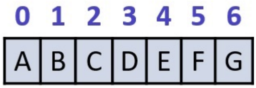
Binary Tree Structure: Array-based implementation

Suitable for complete tree, nearly complete tree.



Hình: Conceptual

```
binaryTree
  data <array of dataType>
end binaryTree
```



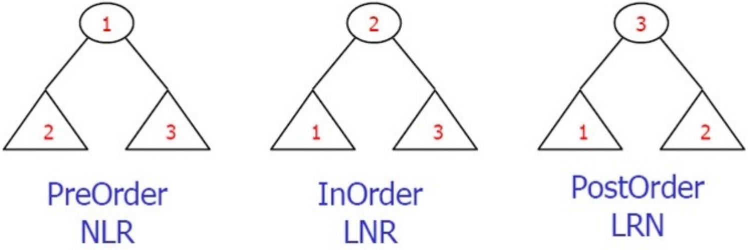
Hình: Physical

Binary Tree Traversals

- Depth-first traversal (duyệt theo chiều sâu): the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child, i.e. processes all of the descendents of a child before going on to the next child.
- Breadth-first traversal (duyệt theo chiều rộng): the processing proceeds horizontally from the root to all of its children, then to its children's children, i.e. each level is completely processed before the next level is started.

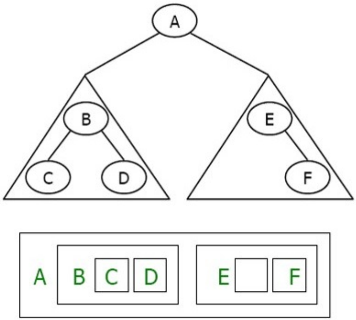
Depth-first traversal

- Preorder traversal
- Inorder traversal
- Postorder traversal

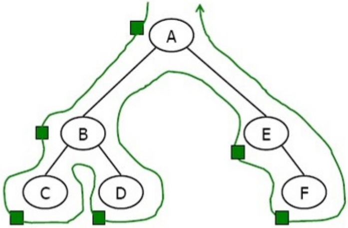


Preorder traversal (NLR)

In the preorder traversal, the root is processed first, before the left and right subtrees.



Processing order



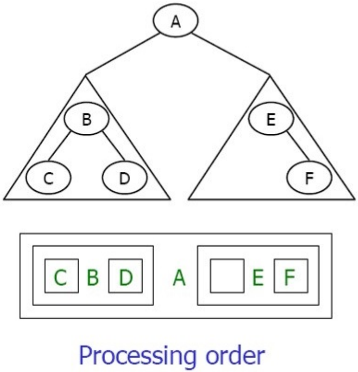
Walking order

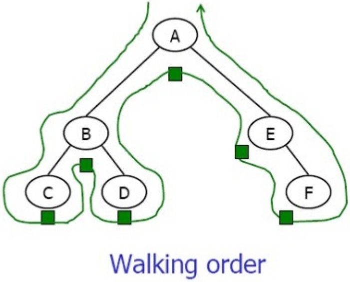
Preorder traversal (NLR)

Algorithm preOrder(val root <pointer>)
Traverse a binary tree in node-left-right sequence.
Pre: `root` is the entry node of a tree or subtree
Post: each node has been processed in order
if *root is not null* **then**
    process(root)
    preOrder(root->left)
    preOrder(root->right)
end
Return
End preOrder

Inorder traversal (LNR)

In the inorder traversal, the root is processed between its subtrees.



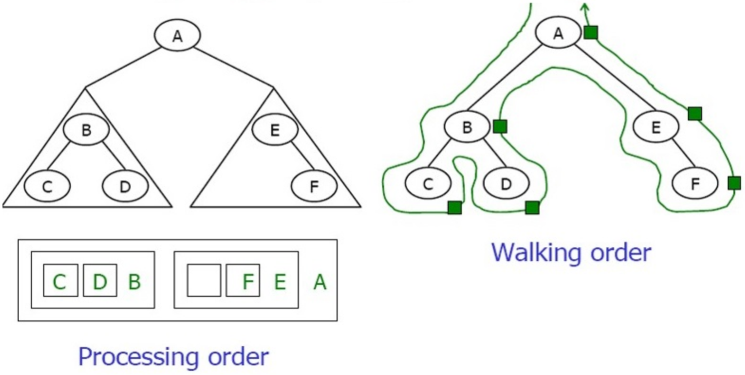


Inorder traversal (LNR)

Algorithm inOrder(val root <pointer>)
Traverse a binary tree in left-node-right sequence.
Pre: `root` is the entry node of a tree or subtree
Post: each node has been processed in order
if *root is not null* **then**
    inOrder(root->left)
    process(root)
    inOrder(root->right)
end
Return
End inOrder

Postorder traversal (LRN)

In the postorder traversal, the root is processed after its subtrees.

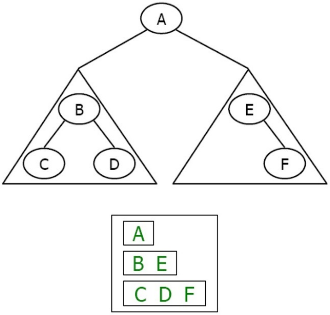


Postorder traversal (LRN)

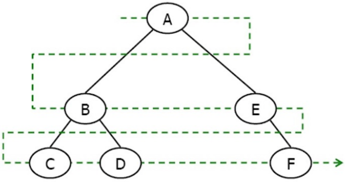
Algorithm postOrder(val root <pointer>)
Traverse a binary tree in left-right-node sequence.
Pre: `root` is the entry node of a tree or subtree
Post: each node has been processed in order
if *root is not null* **then**
    postOrder(root->left)
    postOrder(root->right)
    process(root)
end
Return
End postOrder

Breadth-First Traversals

In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level.



Processing order



Walking order

Breadth-First Traversals

Algorithm breadthFirst(val root <pointer>)
Process tree using breadth-first traversal.

Pre: `root` is node to be processed
Post: tree has been processed

currentNode = root
bfQueue = createQueue()

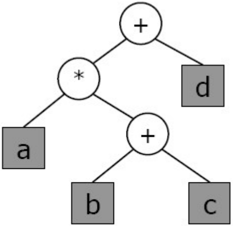
Breadth-First Traversals

```
while currentNode not null do
    process(currentNode)
    if currentNode->left not null then
        enqueue(bfQueue, currentNode->left)
    end
    if currentNode->right not nul then
        enqueue(bfQueue, currentNode->right)
    end
    if not emptyQueue(bfQueue) then
        currentNode = dequeue(bfQueue)
    else
        currentNode = NULL
    end
end
destroyQueue(bfQueue)
End breadthFirst
```

Expression Trees

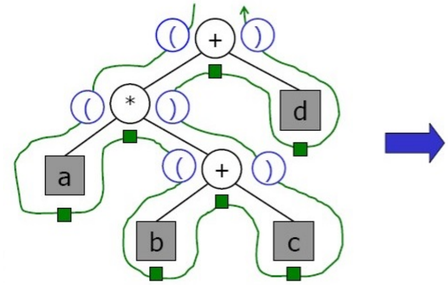
Expression Trees

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



a * (b + c) + d

Infix Expression Tree Traversal



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

Infix Expression Tree Traversal

Algorithm infix(val tree <pointer>)
Print the infix expression for an expression tree.
Pre: `tree` is a pointer to an expression tree
Post: the infix expression has been printed

```
if tree not empty then
    if tree->data is an operand then
        print (tree->data)
    else
        print (open parenthesis)
        infix (tree->left)
        print (tree->data)
        infix (tree->right)
        print (close parenthesis)
    end
end
End infix
```

Postfix Expression Tree Traversal

Algorithm postfix(val tree <pointer>)
Print the postfix expression for an expression tree.
Pre: `tree` is a pointer to an expression tree
Post: the postfix expression has been printed

```
if tree not empty then
    postfix (tree->left)
    postfix (tree->right)
    print (tree->data)
end
End postfix
```

Prefix Expression Tree Traversal

Algorithm prefix(val tree <pointer>)
Print the prefix expression for an expression tree.
Pre: `tree` is a pointer to an expression tree
Post: the prefix expression has been printed

```
if tree not empty then
    print (tree->data)
    prefix (tree->left)
    prefix (tree->right)
end
End prefix
```

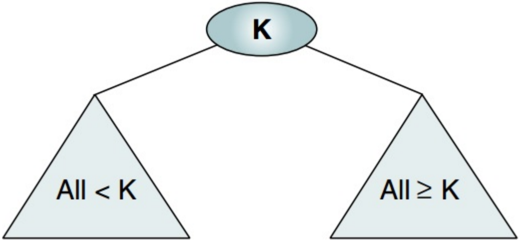
Binary Search Trees

Binary Search Trees

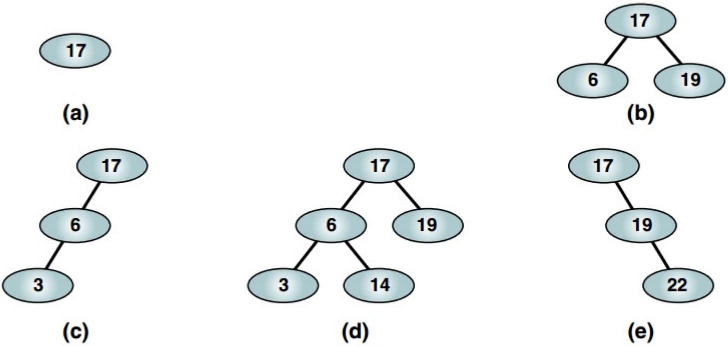
Definition

A binary search tree is a binary tree with the following properties:

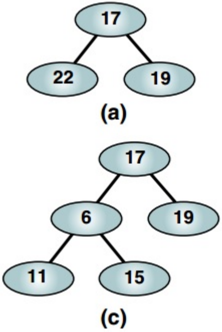
1. All items in the left subtree are less than the root.
2. All items in the right subtree are greater than or equal to the root.
3. Each subtree is itself a binary search tree.



Valid Binary Search Trees



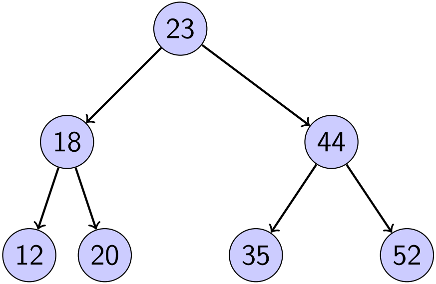
Invalid Binary Search Trees



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).

Binary Search Tree Traversals



- Preorder traversal: 23, 18, 12, 20, 44, 35, 52
- Postorder traversal: 12, 20, 18, 35, 52, 44, 23
- Inorder traversal: **12, 18, 20, 23, 35, 44, 52**

The inorder traversal of a binary search tree produces an ordered list.

Binary Search Tree Search

Find Smallest Node

Algorithm findSmallestBST(val root <pointer>)

This algorithm finds the smallest node in a BST.

Pre: `root` is a pointer to a nonempty BST or subtree

Return address of smallest node

if *root->left null* **then**

  return root

end

return findSmallestBST(root->left)

End findSmallestBST

Binary Search Tree Search

Find Largest Node

Algorithm findLargestBST(val root <pointer>)

This algorithm finds the largest node in a BST.

Pre: `root` is a pointer to a nonempty BST or subtree

Return address of largest node returned

if *root->right null* **then**

  return root

end

return findLargestBST(root->right)

End findLargestBST

Binary Search

Recursive Search

Algorithm searchBST(val root <pointer>, val target <keyType>)
Search a binary search tree for a given value.

Pre: `root` is the root to a binary tree or subtree
`target` is the key value requested

Return the node address if the value is found
`null` if the node is not in the tree

Luong The Nhan,
Tran Giang Son

Binary Search

Recursive Search

```
if root is null then
    return null
end

if target < root->data.key then
    return searchBST(root->left, target)
else if target > root->data.key then
    return searchBST(root->right, target)
else
    return root
end

End searchBST
```

Binary Search

Iterative Search

Algorithm iterativeSearchBST(val root <pointer>, val target <keyType>)
Search a binary search tree for a given value using a loop.

Pre: `root` is the root to a binary tree or subtree
`target` is the key value requested

Return the node address if the value is found
`null` if the node is not in the tree

Binary Search

Iterative Search

while *(root is not NULL) AND (root->data.key <> target)* **do**

    if *target* $<$ *root->data.key* **then**

        root = root->left

    else

        root = root->right

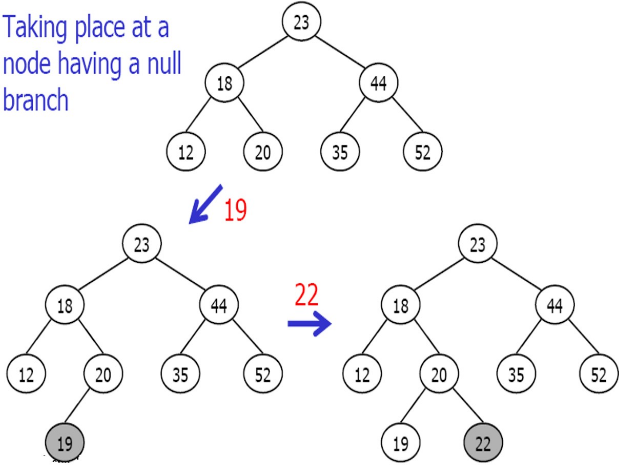
    end

end

return root

End iterativeSearchBST

Insert Node into BST



All BST insertions take place at a leaf or a leaflike node (a node that has only one null branch).

Insert Node into BST: Iterative Insert

Algorithm iterativeInsertBST(ref root <pointer>, val new <pointer>)
Insert node containing new data into BST using iteration.

Pre: `root` is address of first node in a BST
`new` is address of node containing data to be inserted

Post: new node inserted into the tree

Insert Node into BST: Iterative Insert

```
if root is null then
    root = new
else
    pWalk = root
    while pWalk not null do
        parent = pWalk
        if new->data.key < pWalk->data.key then
            pWalk = pWalk->left
        else
            pWalk = pWalk->right
        end
    end
    if new->data.key < parent->data.key then
        parent->left = new
    else
        parent->right = new
    end
end
End iterativeInsertBST
```

Insert Node into BST: Recursive Insert

Algorithm recursiveInsertBST(ref root <pointer>, val new <pointer>)
Insert node containing new data into BST using recursion.

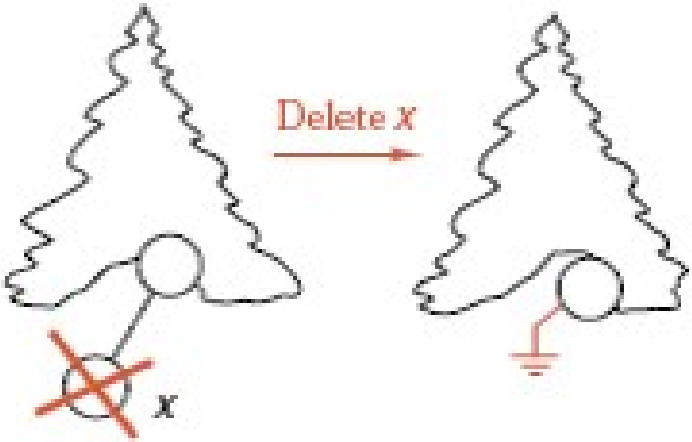
Pre: `root` is address of current node in a BST
`new` is address of node containing data to be inserted

Post: new node inserted into the tree

Insert Node into BST: Recursive Insert

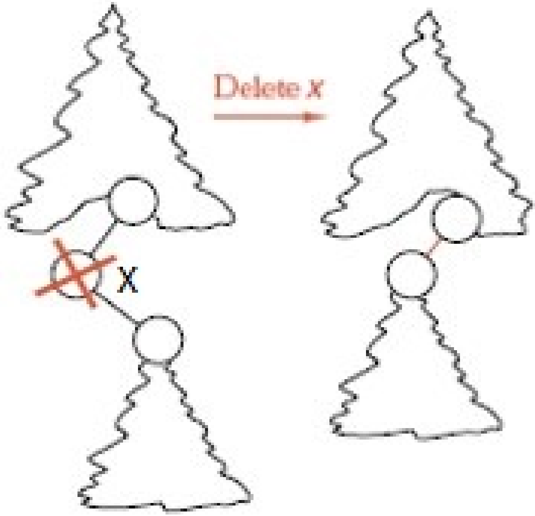
if *root is null* **then**
    root = new
else
    if *new->data.key < root->data.key* **then**
        recursiveInsertBST(root->left, new)
    else
        recursiveInsertBST(root->right, new)
    end
end
Return
End recursiveInsertBST

Delete node from BST



Deletion of a leaf: Set the deleted node's parent link to NULL.

Delete node from BST

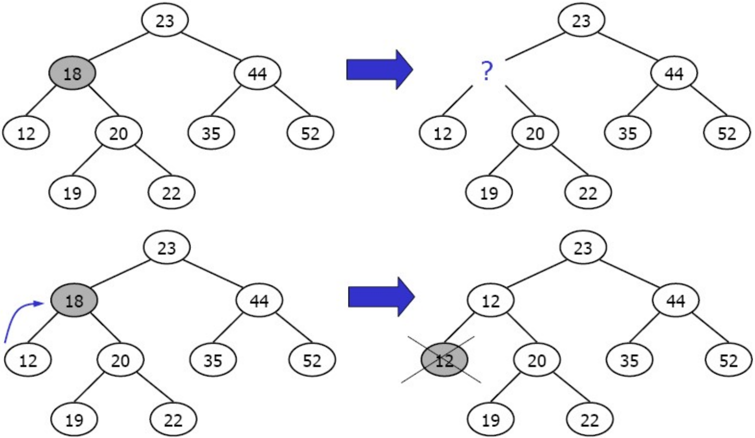


Deletion of a node having only right subtree or left subtree: Attach the subtree to the deleted node's parent.

Delete node from BST

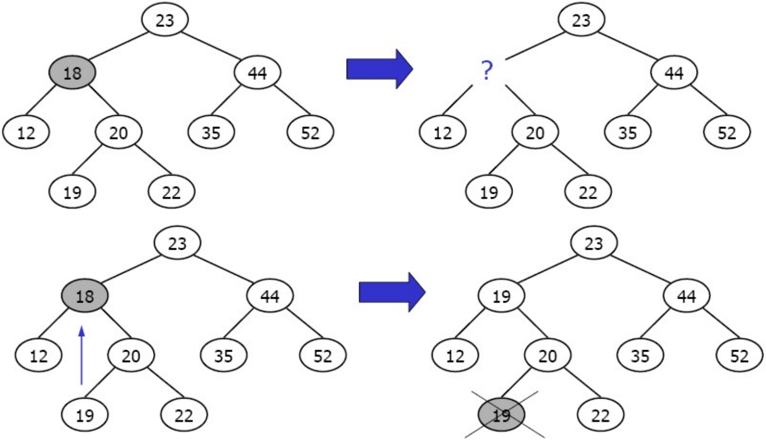
Deletion of a node having both subtrees:
Replace the deleted node by its predecessor or by its successor, recycle this node instead.

Delete node from BST



Using largest node in the left subtree

Delete node from BST



Using smallest node in the right subtree

Delete node from BST

Algorithm deleteBST(ref root <pointer>, val dltKey <keyType>)
Deletes a node from a BST.

Pre: `root` is pointer to tree containing data to be deleted
`dltKey` is key of node to be deleted

Post: node deleted and memory recycled
if `dltKey` not found, `root` unchanged

Return `true` if node deleted, `false` if not found

Delete node from BST

if *root is null* **then**
    return false
end
if *dltKey < root->data.key* **then**
    return deleteBST(root->left, dltKey)
else if *dltKey > root->data.key* **then**
    return deleteBST(root->right, dltKey)

Delete node from BST

else

// Deleted node found – Test for leaf node

if *root->left is null* **then**

dltPtr = root

root = root->right

recycle(dltPtr)

return true

else if *root->right is null* **then**

dltPtr = root

root = root->left

recycle(dltPtr)

return true

Delete node from BST

```
else
    // ...
    else
        // Deleted node is not a leaf.
        // Find largest node on left subtree
        dltPtr = root->left
        while dltPtr->right not null do
            dltPtr = dltPtr->right
        end
        // Node found. Move data and delete leaf node
        root->data = dltPtr->data
        return deleteBST(root->left, dltPtr->data.key)
    end
end
End deleteBST
```