Trees

Chapter 4 of textbook 1

Formal Definition

Tree is a sequence of nodes.

There is a starting node known as root node.

Every node other than the root has a parent node.

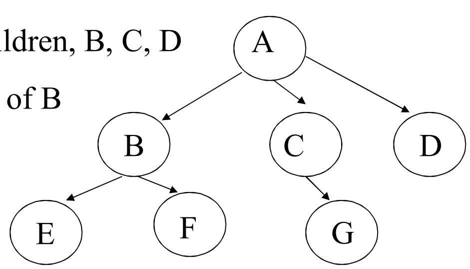
Nodes may have any number of children.

Nodes with no children are know as leaves

A has 3 children, B, C, D

A is parent of B

E is leaf



Some Terminologies:
Father = ancestor
Child= descendant
brother = sibling

Path to a node p is a sequence of nodes root, n_1, n_2, …..p such that n_1 is a child of root, n_2 is a child of n_1 ……

Path to E is ? A,B, E

How many paths are there to one node? Just one!

Depth of a node is the length of the unique path from the root to the node (not counting the node).

Root is at depth 0

Depth of E is ? 2

Leaves are nodes without any children.

D and E are leaf nodes.

Height of a non-leaf node is the length of the LONGEST path from the node to a leaf(not counting the leaf).

Height of a leaf is 0

Height of A is ? 2

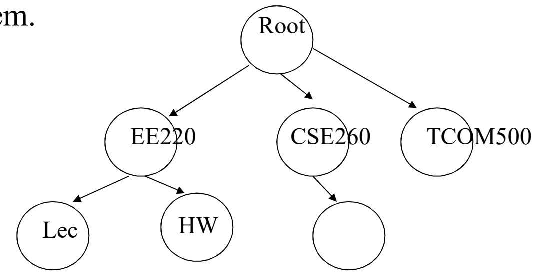
Application

Organization of a file system.

Root directory

EE220, CSE260, TCOM 500

EE220: Lecture Notes, HW



Every node has one or more elements:

Directory example: element of a node is the name of the corresponding directory

Implementation

```
typedef struct TreeNode *PtrToNode;

struct TreeNode
{
    ElementType Element;
    PtrToNode   FirstChild;
    PtrToNode   NextSibling;
}
```

Using pointers

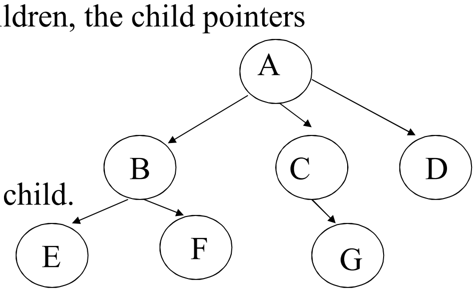
A node has a pointer to all its children

Since a node may have many children, the child pointers have a linked list.

A has a pointer to B, C, D each.

B has pointers to E and its other child.

E does not have any pointers.



```
typedef struct TreeNode *PtrToNode;

struct TreeNode
{
     ElementType Element;
     PtrToNode   FirstChild;
     PtrToNode   NextSibling;
}
```

Example

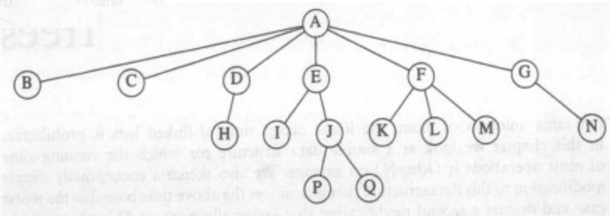


Figure 4.2 A tree

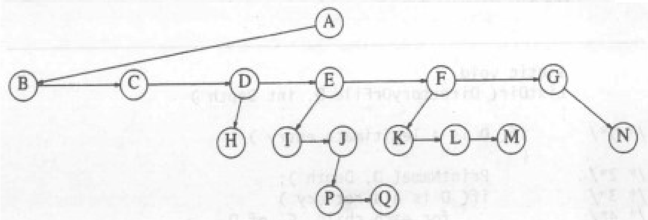


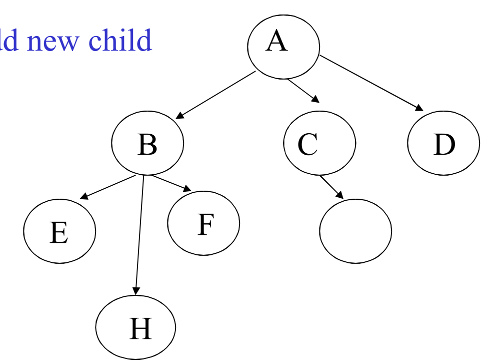
Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Addition of a child:

Create the new child node

Add a pointer to this child in the link list of its parent.

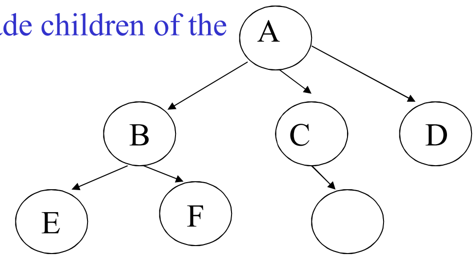
Want to add new child H to B

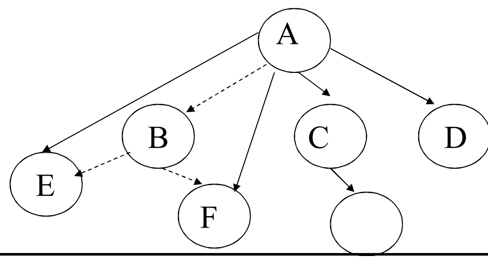


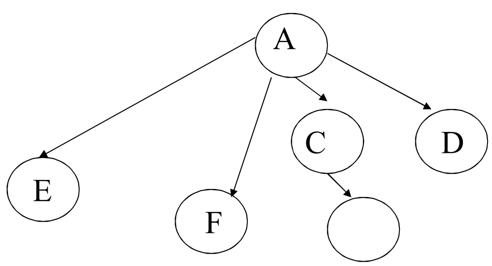
Deletion of a child B:

Children of B are first made children of the parent of B

Node B is deleted.



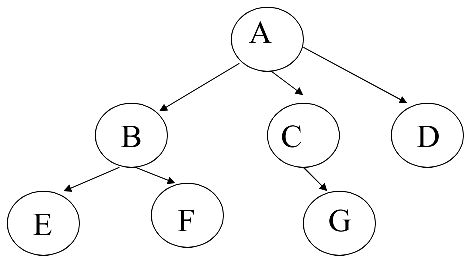




Deletion of the root:

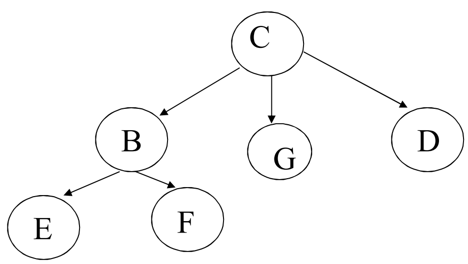
One of the children becomes the new root:

Other children of old root become the children of the new root



C becomes new root

B and D are children of C in addition to its original child



Tree Traversal (Tree Search)

Many algorithms involve walking through a tree, and performing some computation at each node

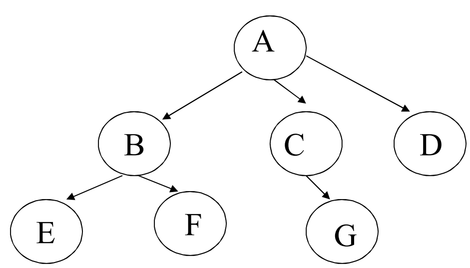
Walking through a tree is called a traversal

Depth-First Search (DFS):

- Pre-order
- Post-order
- In-order (applied for Binary Tree)

Breadth-First Search (BFS) or Level-order Search

DFS: Pre-order

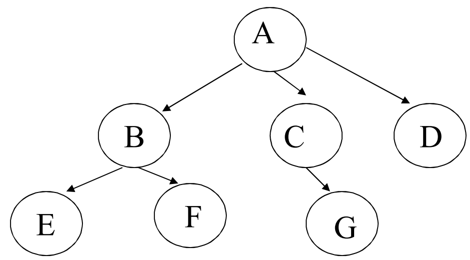


Pre-order:

First visit a node, then with its children

A->B->E->F->C->G->D

DFS: Post-order Example

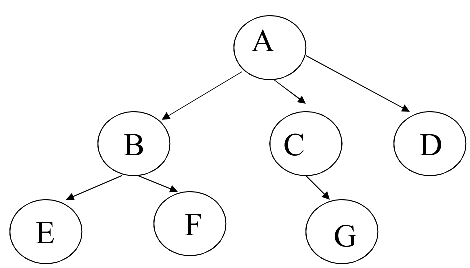


Post-order:

First visit its children, then return to the node.

E->F->B->G->C->D->A

BFS



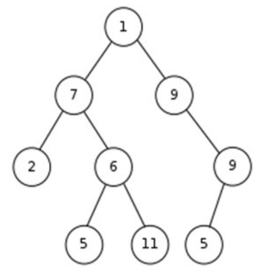
BFS:

First visit all nodes having the same level, then with its children

A -> B->C->D->E->F-G

Binary Trees

A node can have at most 2 children, leftchild and rightchild



What is the largest depth of a binary tree of N nodes? N - 1

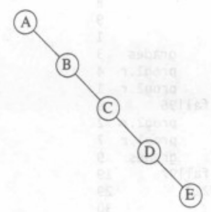


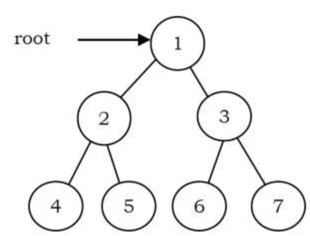
Figure 4.12 Worst-case binary tree

In-order Traversal

First visit the left subtree

Then visit the node

Then visit right subtree



4->2->5->1->6->3->7 in-order

1->2->4->5->3->6->7 pre-order

4->5->2->6->7->3->1 post-order

Quiz

Please give the order of nodes we visit:

DFS Pre-order node-left-right

DFS Post-order left-right-node

DFS In-order left-node-right

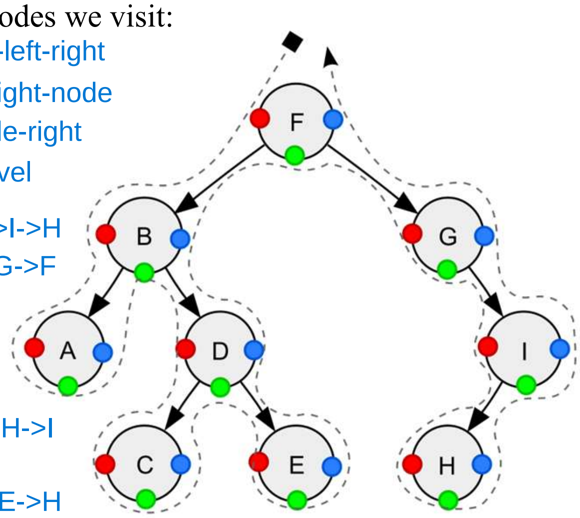
BFS (Level-order) level

1/ F->B->A->D->C->E->G->I->H

2/ A->C->E->D->B->H->I->G->F

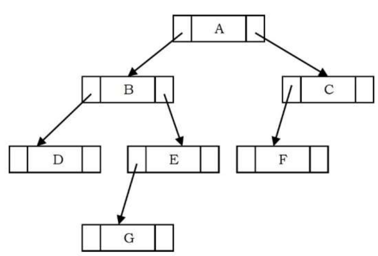
3/ A->B->C->D->E->F->G->H->I

4/ F->B->G->A->D->I->C->E->H



Implementation of a binary tree node

```
struct BinaryTreeNode
{
        ElementType Element;
        BinaryTreeNode * left;
        BinaryTreeNode * right;
};
```




```
typedef struct BinaryTreeNode* PtrToNode;
```

Implementation of Pre-order

Recursive function

```
void PreOrder(struct BinaryTreeNode *root){
    if(root) {
        printf("%d",root→data);
        PreOrder(root→left);
        PreOrder (root→right);
    }
}
```

Time complexity ?   2T(N/2) + 1

Implementation of Pre-order (2): Non-recursive function

```
Stack treeStack;
currNode = root;
While(!isEmpty(treeStack) || currNode != NULL)
{
  if(currNode !=NULL)
        {
        printf(currNode->data);
        push(treeStack, currNode);
        currNode = currNode->left;
        }
  else  {
         prevNode = pop(Stack);
         currNode = prevNode->right;
        }
}
```

Stack is used here

Implementation of Post-order

Recursive function

```
void PostOrder(struct BinaryTreeNode *root){
    if(root)        {
        PostOrder(root→left);
        PostOrder(root→right);
        printf("%d",root→data);
    }
}
```

Time complexity ?

Non-recursive function

Homework

```
Stack treeStack;
Stack auxStack; // Auxiliary stack to help with traversal
currNode = root;
Node* prevNode = NULL; // Previously traversed node

while (!isEmpty(treeStack) || currNode != NULL) {
    if (currNode != NULL) {
        push(treeStack, currNode);
        currNode = currNode->left;
    } else {
        currNode = top(treeStack);
        if (currNode->right != NULL && currNode->right != prevNode) {
            currNode = currNode->right;
        } else {
            printf("%d ", currNode->data);
            pop(treeStack);
            prevNode = currNode;
            currNode = NULL;
        }
    }
}
```

Implementation of In-order

Recursive function

```
void InOrder(struct BinaryTreeNode *root){
    if(root) {
        InOrder(root→left);
        printf("%d",root→data);
        InOrder(root→right);
    }
}
```

Non-recursive function

Homework

```
Stack treeStack;
currNode = root;
while (!isEmpty(treeStack) || currNode != NULL) {
    if (currNode != NULL) {
        push(treeStack, currNode);
        currNode = currNode->left;
    } else {
        currNode = pop(treeStack);
        printf("%d ", currNode->data);
        currNode = currNode->right;
    }
}
```

Implementation of BFS

```
void LevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q = CreateQueue();
    if(!root)
        return;
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //Process current node
        printf("%d", temp→data);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

Queue is used here

Binary Search Tree (BST)

Time complexity of searching in a **binary tree** is O(n).

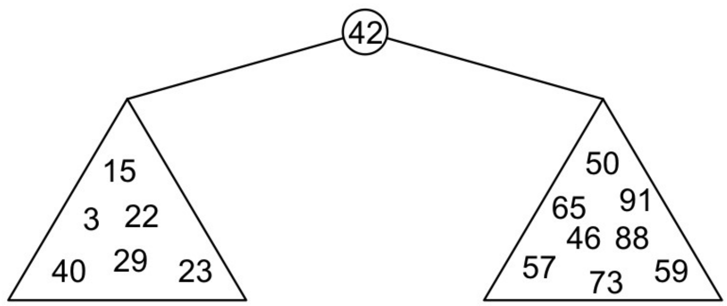
We introduce **a binary search tree** which is useful for searching with time complexity is O(log N) in average case.

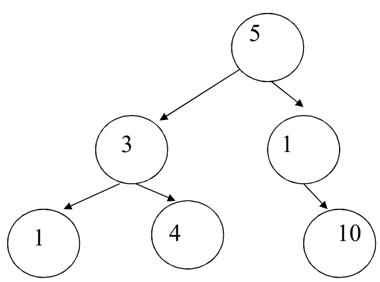
What is **a binary search tree**?

All elements in the left subtree of a node are smaller than the element of the node, and all elements in the right subtree of a node are larger.

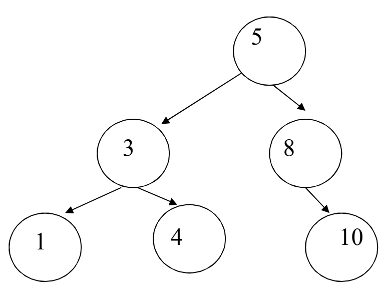
We will assume that in any binary tree, we are not storing duplicate values unless otherwise stated

BST



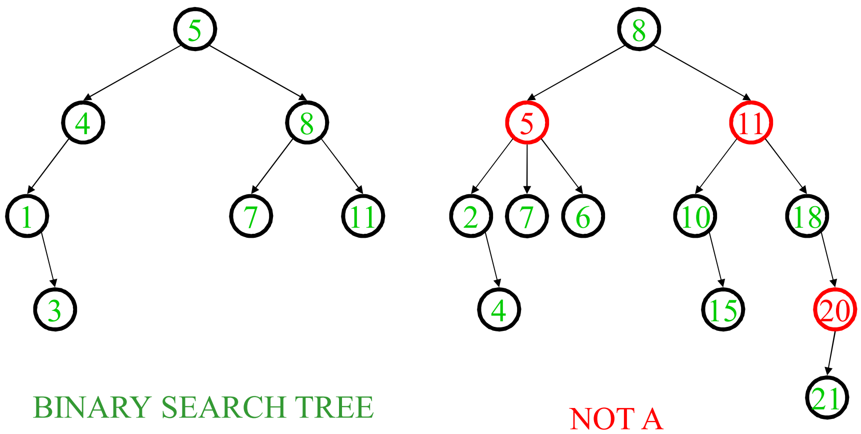


Not binary Search Tree



Binary Search Tree

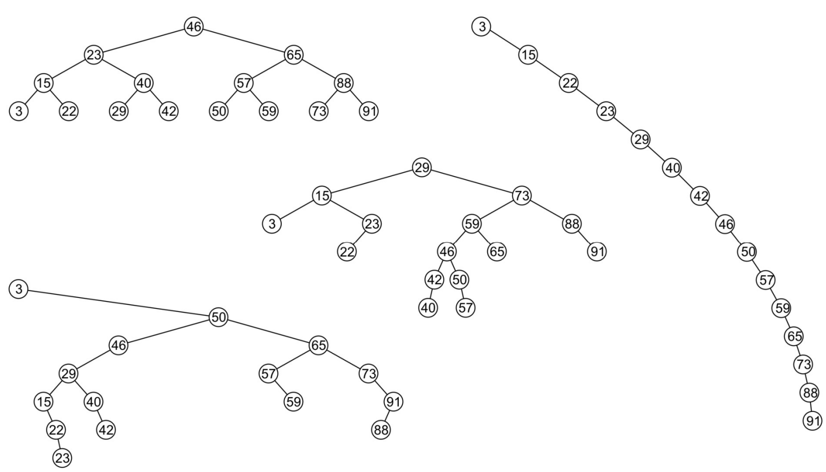
Another Examples



BINARY SEARCH TREE

NOT A
BINARY SEARCH TREE

All these binary search trees store the same data



Finding X in the Tree

Start from the root.

Each time we encounter a node, see if the element in the node equals the X. If yes stop.

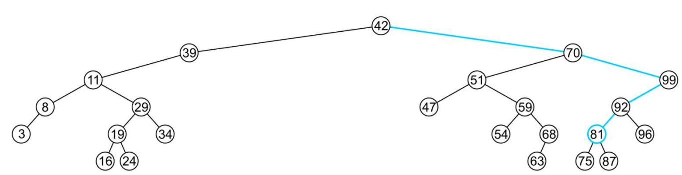
If X is less, go to the left subtree.

If it is more, go to the right subtree.

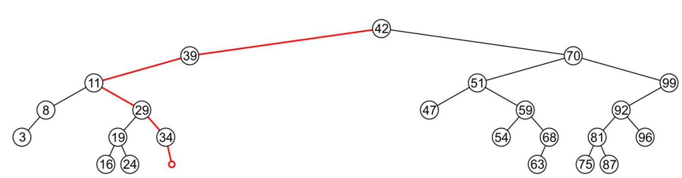
Conclude that X is not in the list if we reach a leaf node and the element in the node does not equal X.

To determine membership, traverse the tree based on the linear relationship:

– If a node containing the value is found, *e.g.*, 81, return 1(Found)



– If an empty node is reached, *e.g.*, 36, the object is not in the tree:



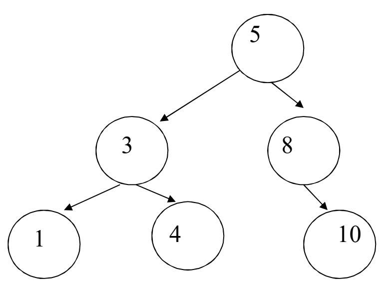
Recursive version of search

```
Search(root, X)
  {
     node = root;
     If (node = NULL) return NOT FOUND;
      Else If (node->element == X) return FOUND;
      Else If (X < node->element) Search(node->leftchild, X);
      Else If (X > node->element) Search(node->rightchild, X);
  }
```

Complexity: O(d), d is the depth,

Average case d= log N

Worse case d = N



Search for 10

Sequence Traveled:

5, 8, 10

Found!

Search for 3.5

Sequence Traveled:

5, 3, 4

Not found!

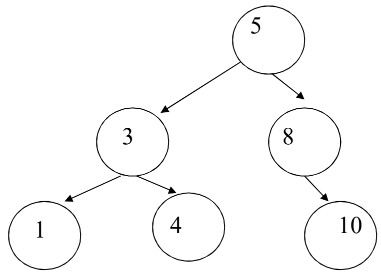
Quiz: Non-recursive version of search

```
TreeNode* searchIterative(TreeNode* root, int key) {
    while (root != NULL && root->data != key) {
        if (key < root->data) {
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return root;
}
```

Find Min and Find Max

Find Min: start at the root and go left as long as there is a left child. The stopping leaf is the smallest element.
Find Max: start at the root and go right as long as there is a right child. The stopping leaf is the greatest element.

Complexity: O(d)



Travel 5, 3, 1

Return 1;

Travel 5, 8, 10

Return 10;

```
// Recursive function to find the minimum value in the BST
TreeNode* findMinRecursive(TreeNode* root) {
    if (root == NULL) {
        return NULL;
    } else if (root->left == NULL) {
        return root;
    } else {
        return findMinRecursive(root->left);
    }
}

// Recursive function to find the maximum value in the BST
TreeNode* findMaxRecursive(TreeNode* root) {
    if (root == NULL) {
        return NULL;
    } else if (root->right == NULL) {
        return root;
    } else {
        return findMaxRecursive(root->right);
    }
}
```

Quiz: implementation

Provide the recursive version and non-recursive version of Find Min and Find Max

```
Find_Min(root):
T = root;
If(T != NULL)
        while(T->left != NULL)
                T=T->left;
return(T);
```

```
Find_Min(root):
T = root;
If(T != NULL)
        if (T-> left ! = NULL)
                Find_Min(T-left);
return T;
```

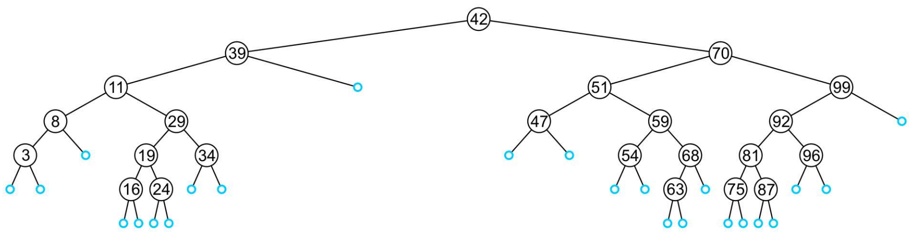
```
// Function to create a new tree node
TreeNode* createNode(int data) {
    TreeNode* newNode =
(TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Non-recursive function to find the minimum value in the
BST
TreeNode* findMinIterative(TreeNode* root) {
    if (root == NULL) {
        return NULL;
    }
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
```

Insertion

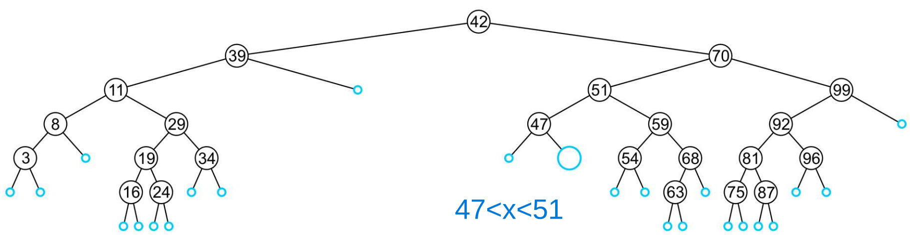
An insertion will be performed at a leaf node:

- Any empty node is a possible location for an insertion



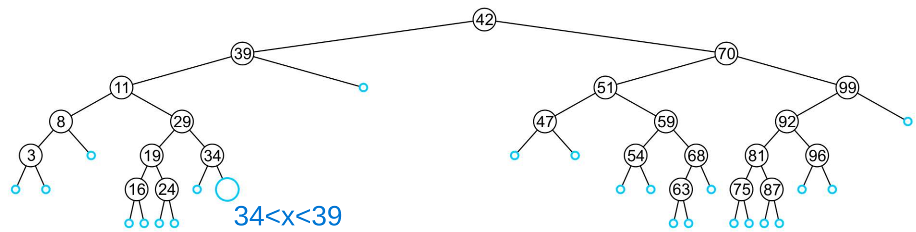
Insertion

For example, this node may hold 48, 49, or 50



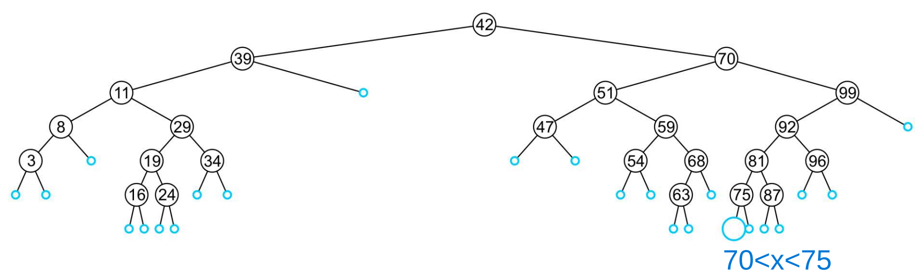
Insertion

An insertion at this location must be 35, 36, 37, or 38



Insertion

This empty node may hold values from 71 to 74



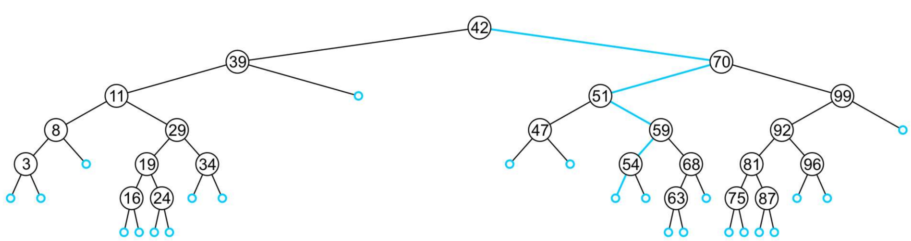
Insertion Algorithm

Like find, we will step through the tree
- If we find the object already in the tree, we will return
  - The object is already in the binary search tree (no duplicates)
- Otherwise, we will arrive at an empty node
- The object will be inserted into that location

Insertion: example 1

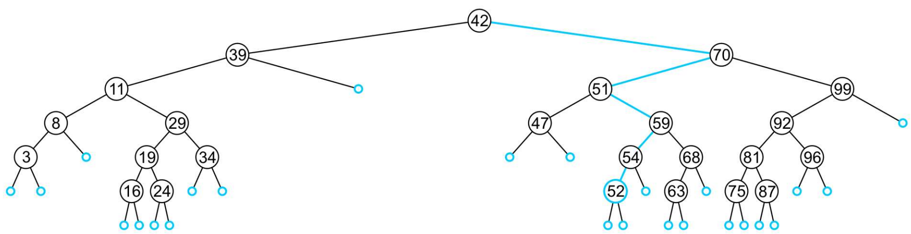
In inserting the value 52, we traverse the tree until we reach an empty node

– The left sub-tree of 54 is an empty node



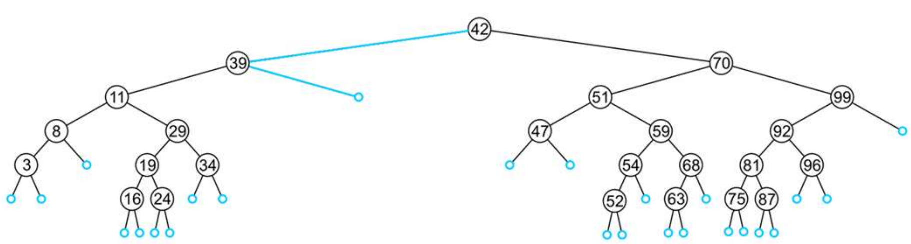
Insertion: example 1

A new leaf node is created and assigned to the member variable `left` of `54`



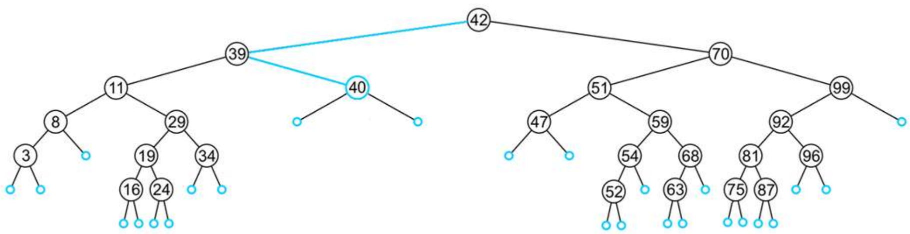
Insertion: Example 2

In inserting 40, we determine the right sub-tree of 39 is an empty node



Insertion: example 2

A new leaf node storing 40 is created and assigned to the member pointer `right of 39`




```
        SearchTree
        Insert( ElementType X, SearchTree T )
        {
/* 1*/      if( T == NULL )
            {
                /* Create and return a one-node tree */
/* 2*/          T = malloc( sizeof( struct TreeNode ) );
/* 3*/          if( T == NULL )
/* 4*/              FatalError( "Out of space!!!" );
                else
                {
/* 5*/              T->Element = X;
/* 6*/              T->Left = T->Right = NULL;
                }
            }
            else
/* 7*/      if( X < T->Element )
/* 8*/          T->Left = Insert( X, T->Left );
            else
/* 9*/      if( X > T->Element )
/*10*/          T->Right = Insert( X, T->Right );
            /* Else X is in the tree already; we'll do nothing */

/*11*/      return T;  /* Do not forget this line!! */
        }
```

Figure 4.22 Insertion into a binary search tree

Complexity: O(d)

Quiz: Insertion

Blackboard example:

– In the given order, insert these objects into an initially empty binary search tree:

31 45 36 14 52 42 6 21 73 47 26 37 33 8

– What values could be placed:
  - To the left of 21? 15->20
  - To the right of 26? 27->30
  - To the left of 47? 46

– How would we determine if 40 is in this binary search tree?

– Which values could be inserted to increase the height of the tree?

Erase (Deletion)

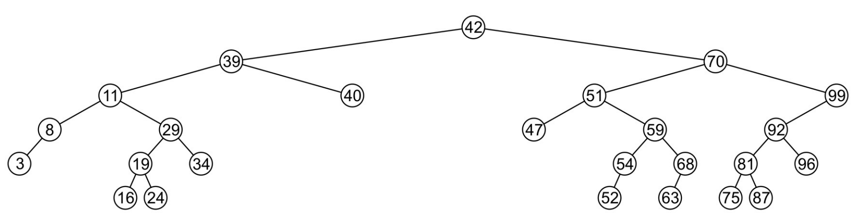
A node being erased is not always going to be a leaf node

There are three possible scenarios:

– The node is a leaf node,

– It has exactly one child, or

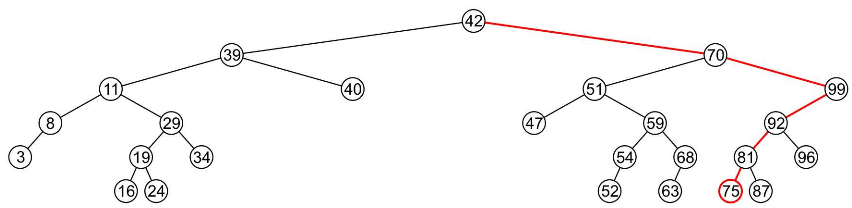
– It has two children (it is a full node)



Erase

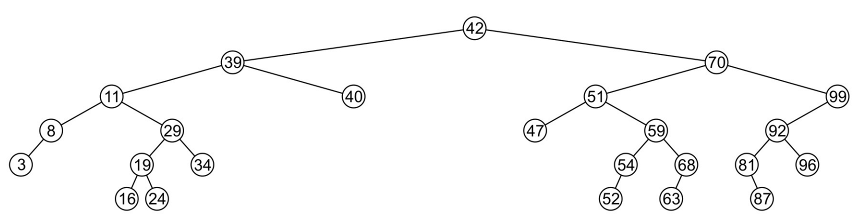
A leaf node simply must be removed and the appropriate member variable of the parent is set to `NULL`

– Consider removing 75



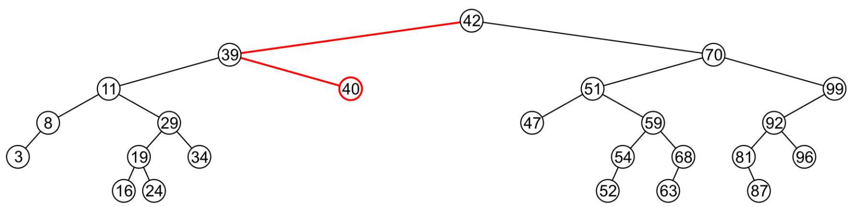
Erase

The node is deleted and `left pointer` of 81 is set to `NULL`



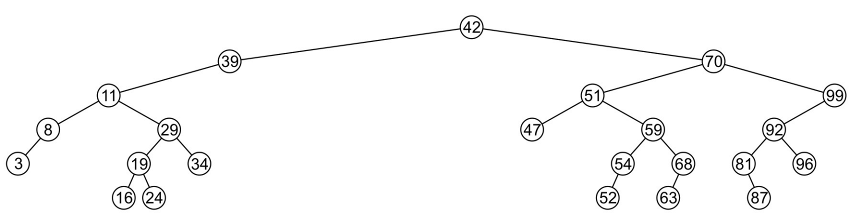
Erase

Erasing the node containing 40 is similar



Erase

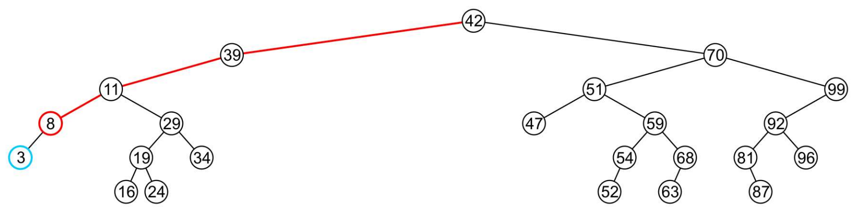
The node is deleted and `right` `pointer` of 39 is set to `NULL`



Erase

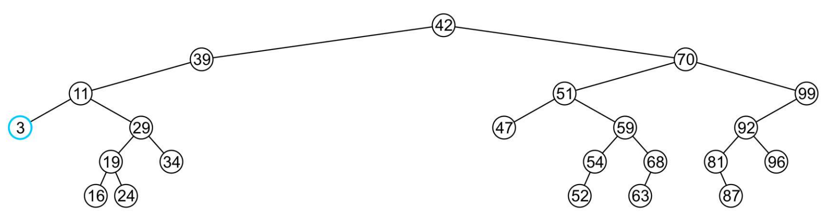
If a node has only one child, we can simply promote the sub-tree associated with the child

– Consider removing 8 which has one left child



Erase

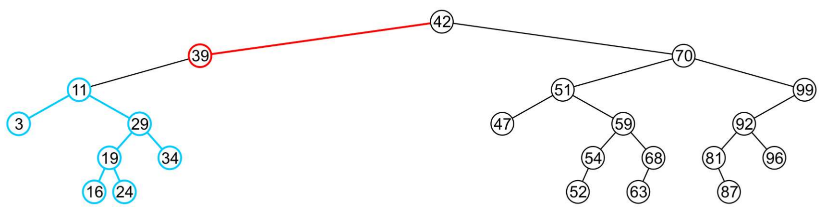
The node 8 is deleted and the `left pointer` of 11 is updated to point to 3.



Erase

There is no difference in promoting a single node or a sub-tree

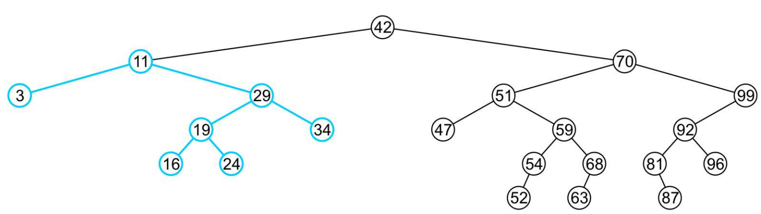
– To remove 39, it has a single child 11



Erase

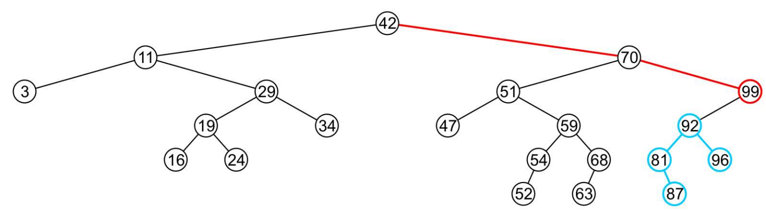
The node containing 39 is deleted and `left_node` of 42 is updated to point to 11

– Notice that order is still maintained



Erase

Consider erasing the node containing 99

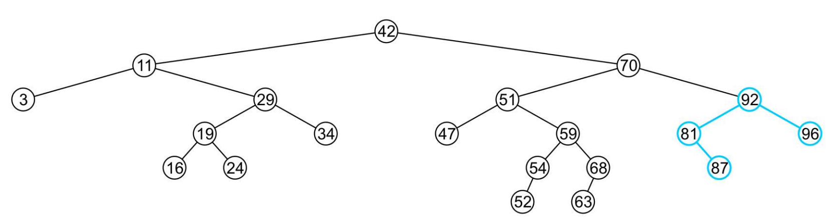


Erase

The node is deleted and the left sub-tree is promoted:

– The member variable `right pointer` of 70 is set to point to 92.

– Again, the order of the tree is maintained.

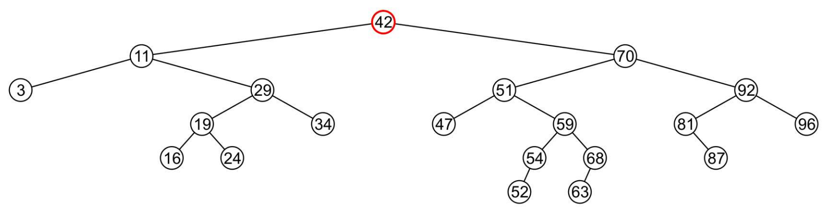


Erase

Finally, we will consider the problem of erasing a full node, *e.g.*, 42

We will perform two operations:

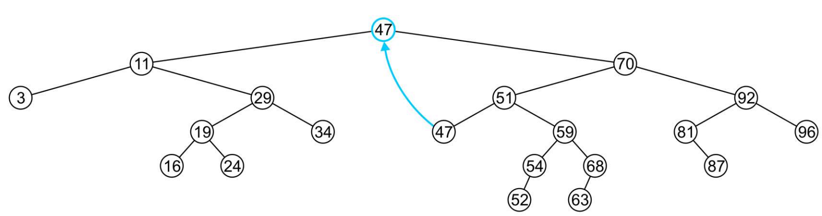
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree



Erase

In this case, we replace 42 with 47

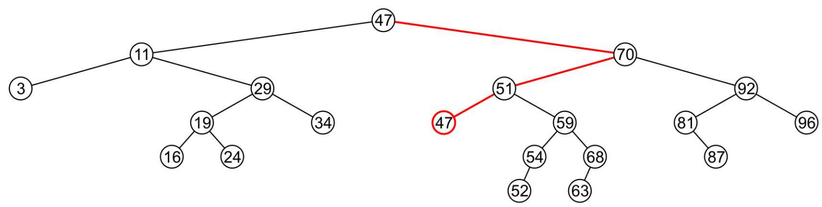
- We temporarily have two copies of 47 in the tree



Erase

We now recursively erase 47 from the right sub-tree

– We note that 47 is a leaf node in the right sub-tree

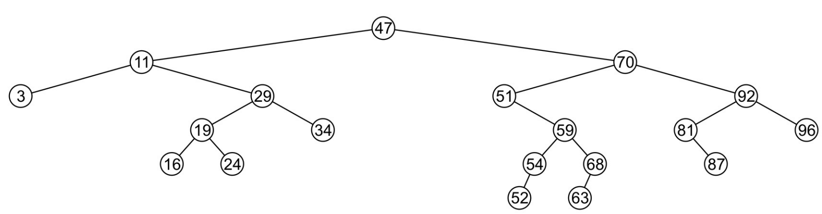


Erase

Leaf nodes are simply removed and `left pointer` of 51 is set to `NULL`

– Notice that the tree is still sorted:

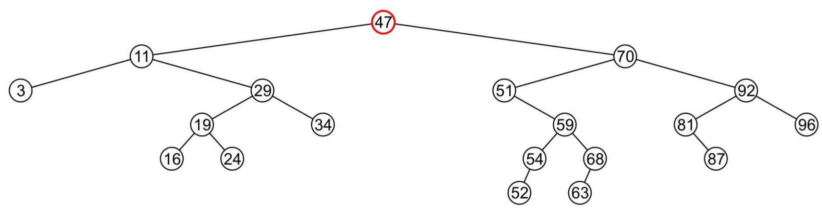
47 was the least object in the right sub-tree



Erase

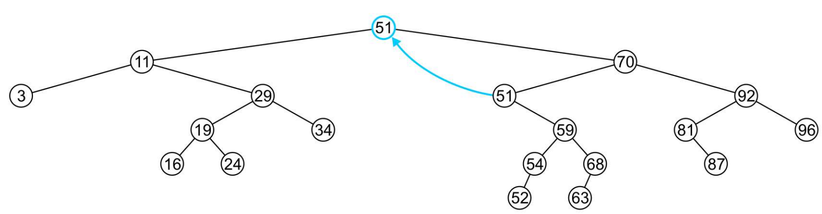
Suppose we want to erase the root 47 again:

- We must copy the minimum of the right sub-tree
- We could promote the maximum object in the left sub-tree and achieve similar results



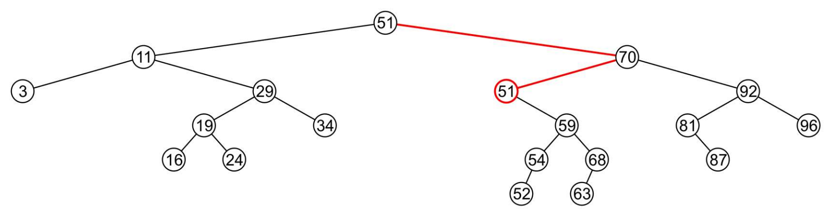
Erase

We copy 51 from the right sub-tree



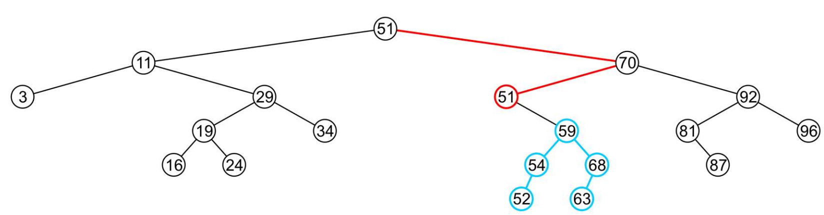
Erase

We must proceed by delete 51 from the right sub-tree



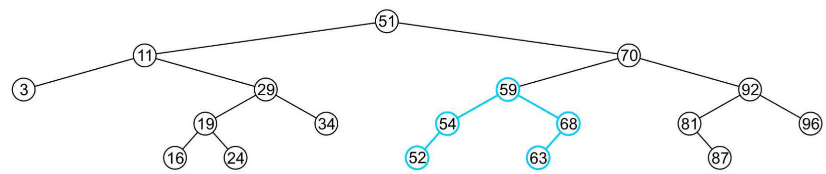
Erase

In this case, the node storing 51 has just a single child



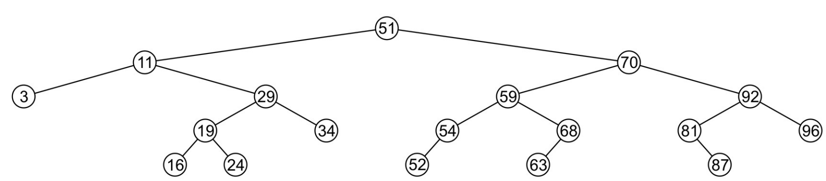
Erase

We delete the node containing 51 and assign the member variable `left pointer` of 70 to point to 59.



Erase

Note that after seven removals, the remaining tree is still correctly sorted

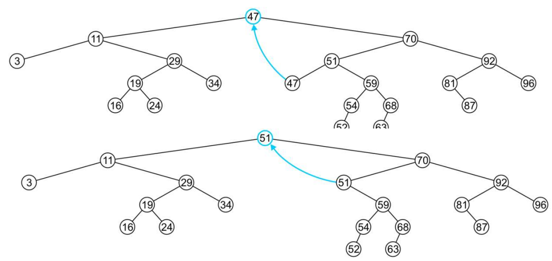


Erase

In the two examples of removing a full node, we promoted:

- A node with no children
- A node with right child

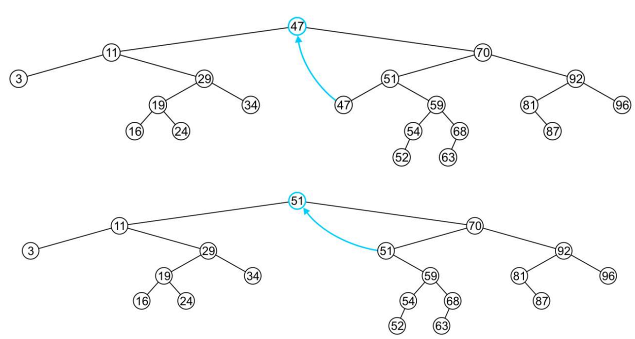
Is it possible, in removing a full node, to promote a child with two children?



Erase

Recall that we promoted the minimum value in the right sub-tree

- If that node had a left sub-tree, that sub-tree would contain a smaller value



Pseudo Code

```
Delete(node) {
If a node is childless, then
    {
        node->parent->ptr_to_node = NULL
        free node;
    }
If a node has one child
    {
        node->parent->child = node->child;
        free node;
    }
```

```
If a node has 2 children,
    {
        minnode = findmin(rightsubtree)->key;
        node->key = minnode->key;
        delete(minnode);
    }
}
```

Complexity? O(d)

Quiz: Erase

Blackboard example:

- In the binary search tree generated previously:
  - Erase 47
  - Erase 21
  - Erase 45
  - Erase 31
  - Erase 36

Next week

- Online Assignment 2
  - Duration :1h
  - Content: Linked List, Stack, Queue
- AVL Tree