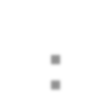
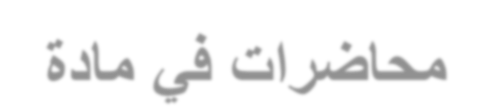
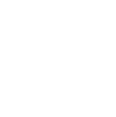
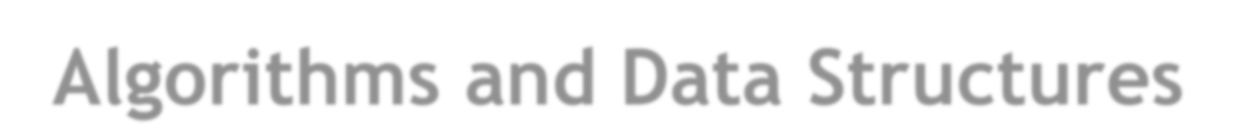
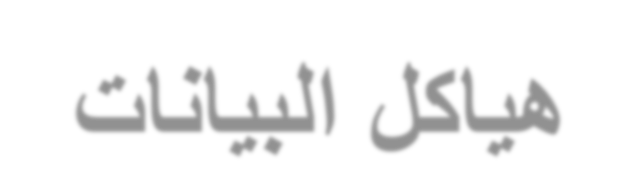
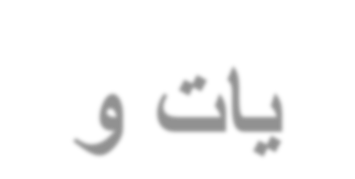
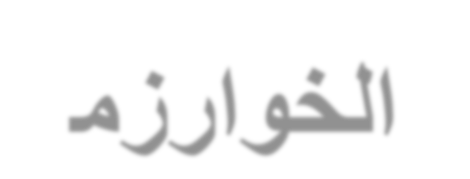
186



**Lecture # 9**

Binary Search Trees

(BST)

Outline

• Binary Search Trees (BST)

 BST property.

 Searching in a BST.

 Inorder traversal of BST.

 FindMin and FindMax.

 Insertion into a BST.

 Deletion a node.

Binary Search Trees (BST):

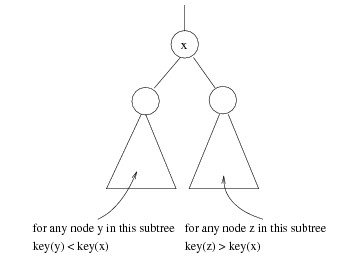
An application of Binary Trees

• **Binary Search Trees** (**BST**) are a type of Binary Trees with a special organization of data.

• Following we will discuss the property of the binary search tree (BST):

• **Binary search tree property**:

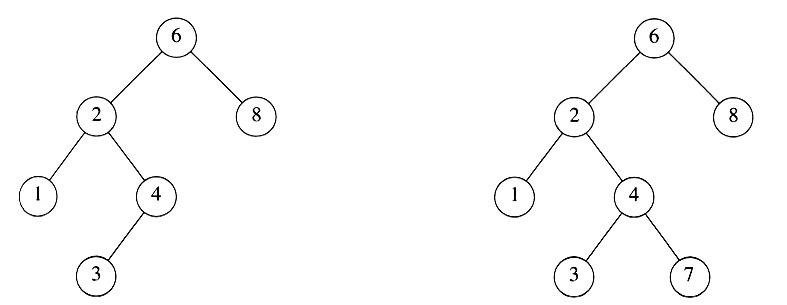
 For every node **X:**



 All the keys in its ***left subtree*** are ***smaller*** than the key value in **X**.

 All the keys in its ***right subtree*** are ***larger*** than the key value in **X**.

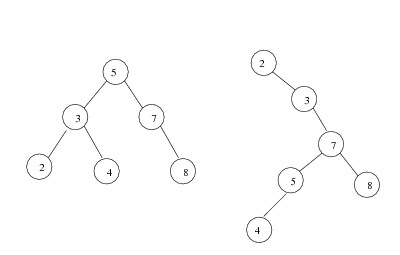
Binary Search Trees (cont…)



A binary search tree **Not** a binary search tree

Binary Search Trees (cont…)

The same set of keys may have different BSTs



BST Example

• Given the following sequence of numbers,

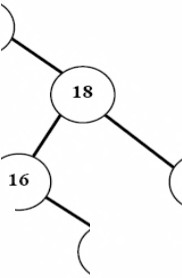
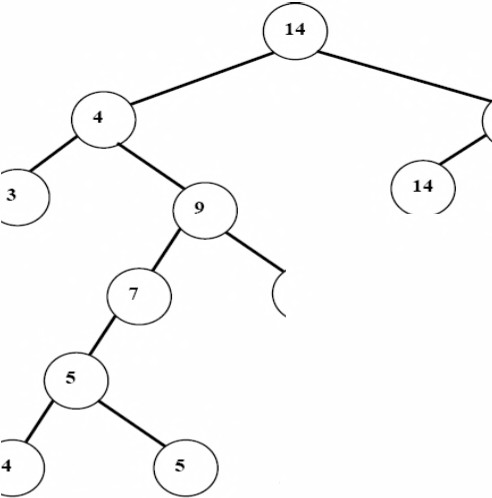
**14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5**

• The following binary search tree can be constructed.

BST Example (cont…)

15

17



20

*ee*

*9*

*A Binary Search Tr*

Binary Search Tree Class

**class BinarySearchTree**

**{**

**public:**

**BinarySearchTree(); // Constructor**

**~BinarySearchTree(); // Destructor**

**void destroy\_tree(); void insert(int x); void inorder();**

**void postorder();**

**void preorder();**

|  |  |  |
| --- | --- | --- |
| **BinaryNode** | **\*** | **Find(int x);** |
| **BinaryNode** | **\*** | **FindMin();** |
| **BinaryNode** | **\*** | **FindMax();** |

Binary Search Tree Class (cont…)

**private:**

**void destroy\_tree(BinaryNode \*t); void insert(int x, BinaryNode \*t); void inorder(BinaryNode \*t);**

**void postorder(BinaryNode \*t);**

**void preorder(BinaryNode \*t);**

|  |  |  |
| --- | --- | --- |
| **BinaryNode** | **\*** | **Find(int x, BinaryNode \*t);** |
| **BinaryNode** | **\*** | **FindMin(BinaryNode \*t);** |
| **BinaryNode** | **\*** | **FindMax(BinaryNode \*t);** |

**BinaryNode \*root;**

**};**

Constructor and Destructor

**BinarySearchTree::BinarySearchTree()**

**{**

**root = NULL;**

**}**

**BinarySearchTree::~BinarySearchTree()**

**{**

**destroy\_tree();**

**}**

destroy\_tree

**void BinarySearchTree::destroy\_tree(BinaryNode \*t)**

**{**

**if(t != NULL)**

**{**

**destroy\_tree(t->left); destroy\_tree(t->right); delete t;**

**}**

**}**

**void BinarySearchTree::destroy\_tree()**

**{**

**destroy\_tree(root);**

**}**

Searching in a BST

• To search for a number ***b***:

1. Compare ***b*** with the root;

 If (***b*** = root), return the node.

 If (***b*** < root), go left.

 If (***b*** > root), go right.

2. Repeat step **1**, comparing ***b*** with the new node we are at.

3. Repeat until either the node is found or we reach a non-existing node.

Searching in a BST (cont…)

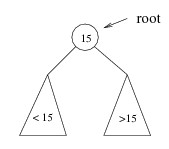
• If we are searching for a **key=15**, then we are done.

• If we are searching for a **key<15**, then we should search in the

***left subtree***.

• If we are searching for a **key>15**, then we should search in the

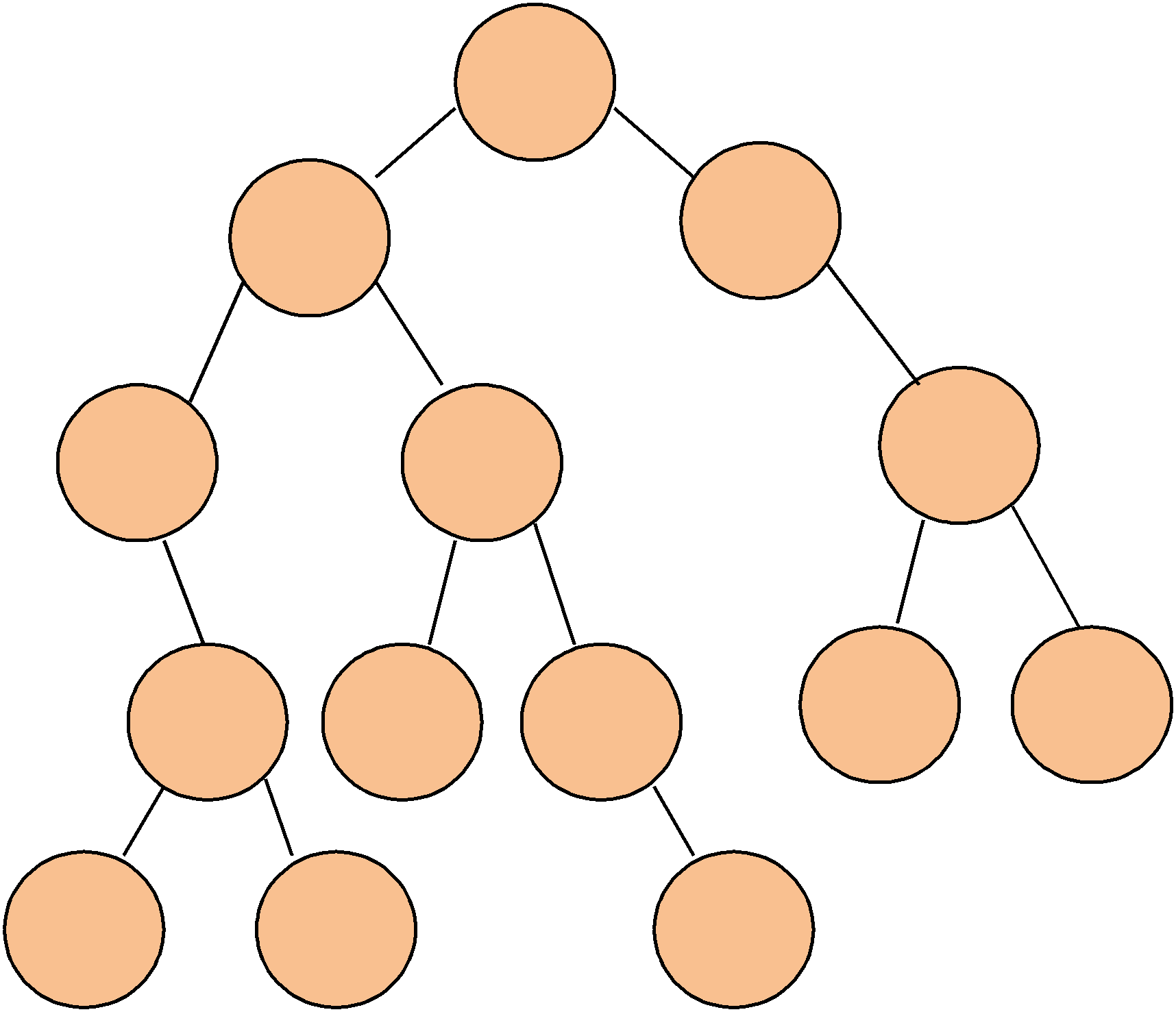
***right subtree***.



Searching in a BST (cont…)

• Try it with ***b***=12, and also with ***b***=17.

15



8 20

2 11 27

6 10 12

22 30

3 7 14

Searching (Find)

• Find **X:**

 Return a pointer to the node that has key **X**.

 Or NULL if there is no such node.

• **Time complexity: O(h)**, where **h** is the height of the tree.

• ***The height of a tree*** is the longest path from the root to a leaf.

Searching (Find) (cont…)

**BinaryNode \* BinarySearchTree::Find(int x, BinaryNode \*t)**

**{**

**if (t == NULL)**

**return NULL;**

**else if (x < t->element)**

**return Find(x, t->left);**

**else if (x > t->element)**

**return Find(x, t->right);**

**else**

**}**

**return t; // match (the node is found)**

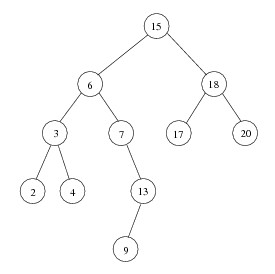
**BinaryNode \* BinarySearchTree::Find(int x)**

**{**

**return Find(x, root);**

**}**

Inorder Traversal of BST



• Inorder traversal of BST prints out all the keys in **sorted order**.

**Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20**



FindMin / FindMax

• **Goal:** return the node containing the smallest (largest) key in the tree.

• **Algorithm:**

 Start at the root.

 Go left (right) as long as there is a left (right) child.

 The stopping point is the smallest (largest) element.

• **Time complexity: O(h)**, where **h** is the height of the tree.

FindMin

**BinaryNode \* BinarySearchTree::FindMin(BinaryNode \*t)**

**{**

**if (t == NULL)**

**return NULL;**

**if (t->left == NULL)**

**return t;**

**return FindMin(t->left);**

**}**

**BinaryNode \* BinarySearchTree::FindMin()**

**{**

**return FindMin(root);**

**}**

FindMax

**BinaryNode \* BinarySearchTree::FindMax(BinaryNode \*t)**

**{**

**if (t == NULL)**

**return NULL;**

**if (t->right == NULL)**

**return t;**

**return FindMax(t->right);**

**}**

**BinaryNode \* BinarySearchTree::FindMax()**

**{**

**return FindMax(root);**

**}**

Insertion into a BST

• Proceed down the tree as you would with a **Find**.

 If **X** is found, do nothing (or update something).

 Otherwise, insert **X** at the last spot on the path traversed.

• **Time complexity: O(h)**, where **h** is the height of the tree.

Insertion into a BST (cont…)

• **Example:** insert(13)

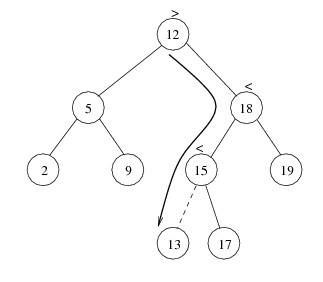
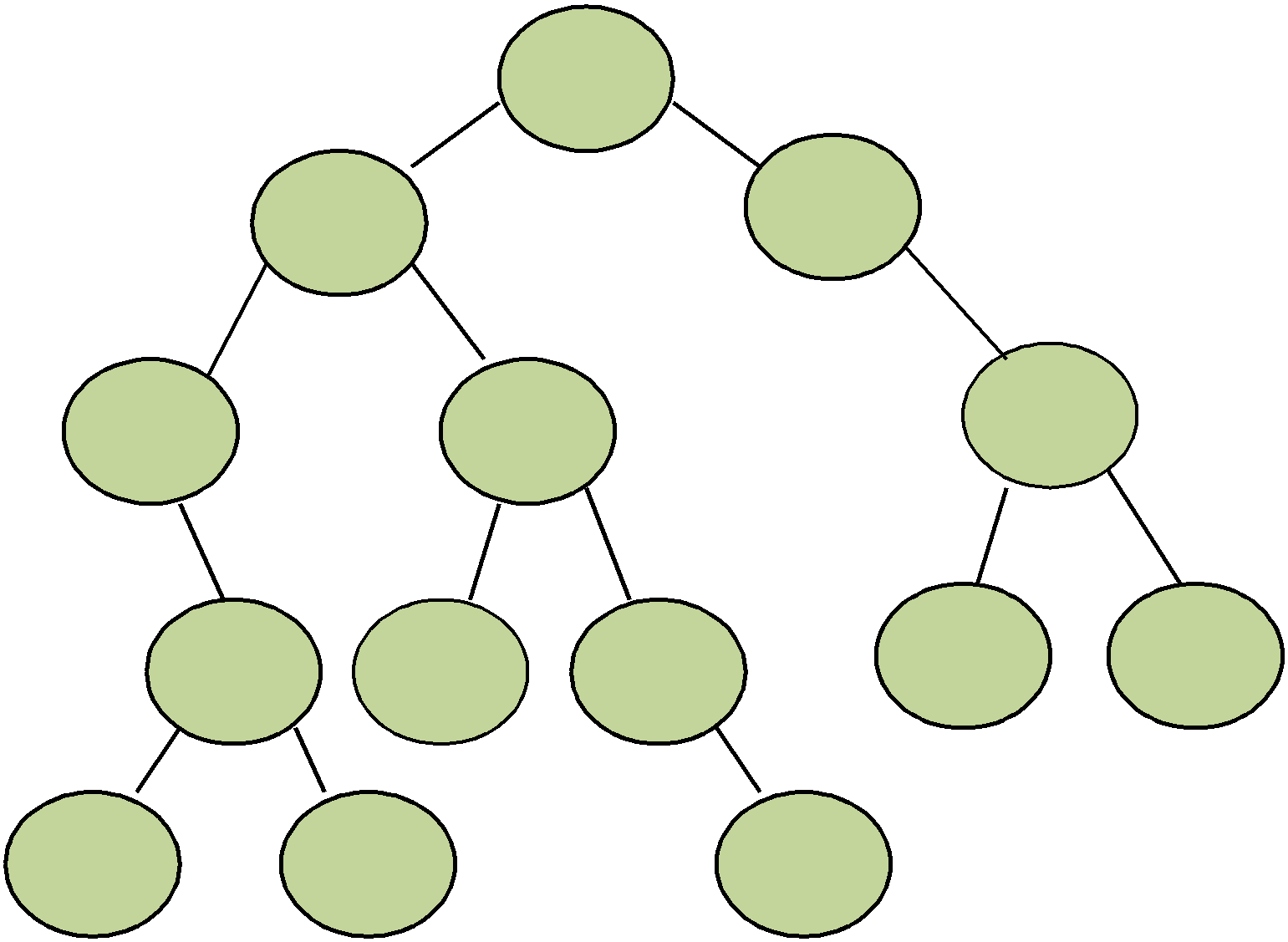


Illustration of Insertion

Before inserting 25 After inserting 25

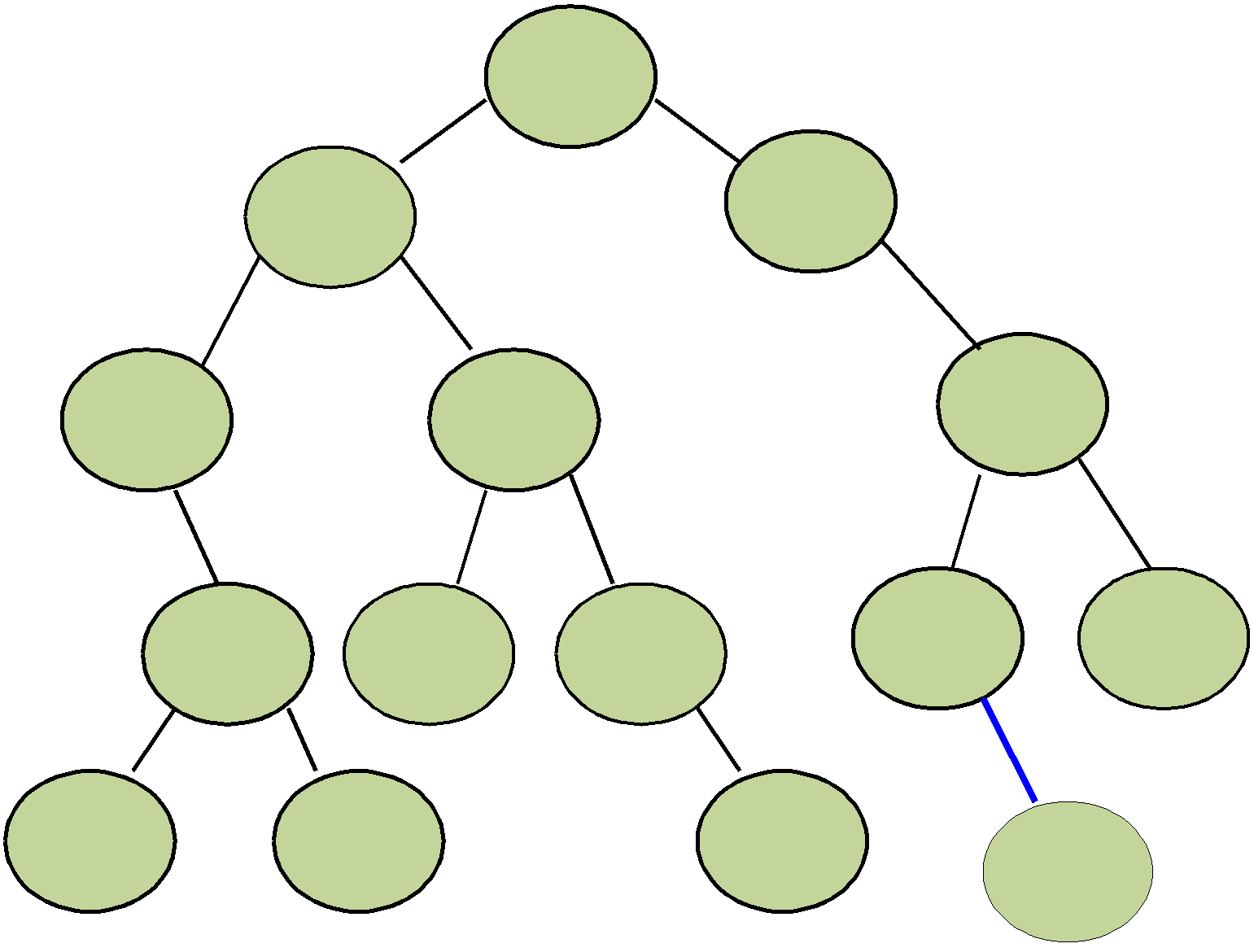
15



8 20

15

8 20



2 11 27

2 11 27

6 10 12

22 30

6 10 12

22 30

3 7 14

3 7 14 25

insert(int x, BinaryNode \*t)

|  |  |  |
| --- | --- | --- |
| **void** | **BinarySearchTree::insert(int x, BinaryNode** | **\*t)** |
| **{** | **BinaryNode\* newNode = new BinaryNode;** |  |

**if(x < t->element)**

**{**

**if(t->left != NULL)**

**insert(x, t->left);**

**else {**

**}**

**t->left = newNode; newNode->element = x; newNode->left = NULL; newNode->right = NULL; }**

**else if(x > t->element)**

**{**

**if(t->right != NULL)**

**insert(x, t->right);**

**else {**

**}**

**}**

**t->right = newNode; newNode->element = x; newNode->left = NULL; newNode->right = NULL; }**

insert(int x)

**void BinarySearchTree::insert(int x)**

**{**

**if(root != NULL)**

**insert(x, root);**

**else**

**{**

**}**

**}**

**root = new BinaryNode;**

**root->element = x; root->left = NULL; root->right = NULL;**

Deletion

• When we delete a node, we need to consider how we take care of the children of the deleted node.

 This has to be done such that the property of the binary search tree is maintained.

• **Time complexity: O(h)**, where **h** is the height of the tree.

Deletion under Different Cases

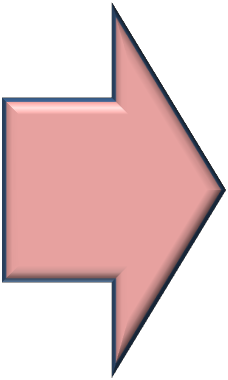
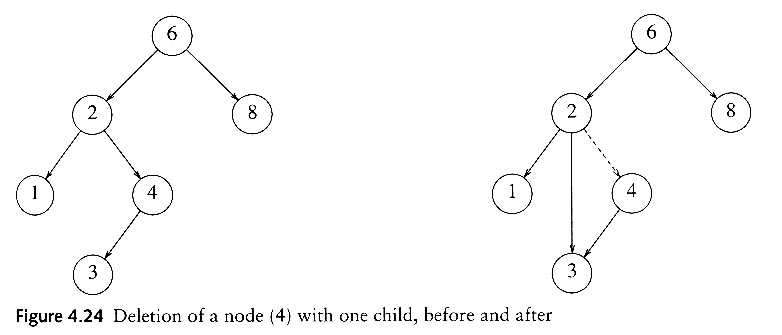
 **Case 1: the node is a leaf**

 Delete it immediately.

 **Case 2: the node has one child**

 Adjust a pointer from the parent to bypass that node.

Deletion Case 2



Deletion under Different Cases

(cont…)

 **Case 3: the node has 2 children (or 2 subtrees)**

 Replace the key of that node with the **maximum** element at the **left subtree**.

• Delete that maximum element.

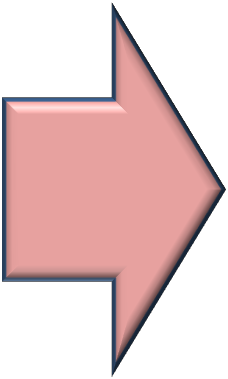
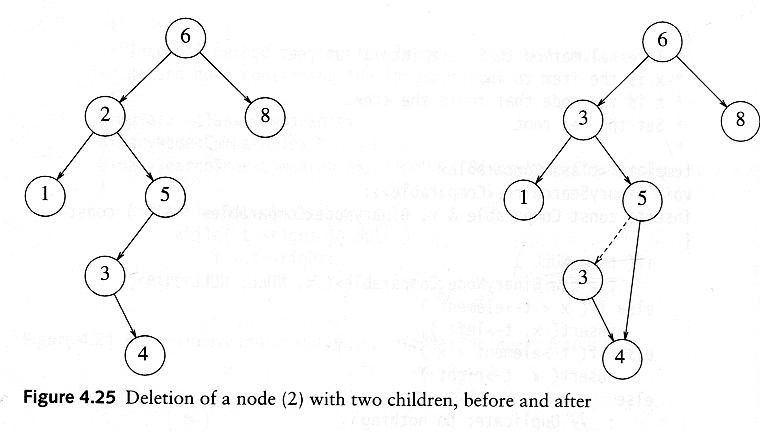
**OR**

 Replace the key of that node with the **minimum** element

at the **right subtree**.

• Delete that minimum element.

Deletion Case 3



Using **BinarySearchTree**

**int main(void) { BinarySearchTree bst; bst.insert(5); bst.insert(3); bst.insert(9); bst.insert(4);**

**cout << "Inorder:"; bst.inorder(); cout << endl;**

**cout << "Postorder:"; bst.postorder(); cout << endl; cout << "Preorder:"; bst.preorder(); cout << endl; int a;**

**cout << "\n Enter your value to search: ";**

**cin >> a;**

**if(bst.search(a) == NULL)**

**cout << "\n The value " << a << " is not found.\n";**

**else**

**cout << "\n The value " << a << " is found.\n";**

Using **BinarySearchTree** (cont…)

**if(bst.FindMin() != NULL)**

**cout << "\n The minimum value in the tree is: "**

**<< bst.FindMin()->element;**

**else**

**cout << "\n Error... \n";**

**if(bst.FindMax() != NULL)**

**cout << "\n The maximum value in the tree is: "**

**<< bst.FindMax()->element;**

**else**

**cout << "\n Error... \n";**

**}**

Result

